А. Л. Марченко

Python

БОЛЬШАЯ КНИГА ПРИМЕРОВ

Издательство Московского университета
2023

УДК 004.438
ББК 32.973.2
М30

Марченко, А. Л.
М30 Python: большая книга примеров / А. Л. Марченко. — Москва : Издательство Московского университета, 2023. — 361, [1] с. — (Электронное издание сетевого распространения).

ISBN 978-5-19-011853-7 (e-book)

Большая книга примеров основывается на описании языка программирования Python (https://docs.python.org/) и множества материалов из различных интернет-источников. Основное предназначение книги — формирование представления о языке на основе его описания и примеров его применения.

Книга может быть использована в качестве учебного пособия.

УДК 004.438
ББК 32.973.2

ISBN 978-5-19-011853-7 (e-book)

Языки. Общие представления

Язык - это инструмент для представления (кодирования) ИНФОРМАЦИИ. Информация первична: сам по себе как инструмент язык представляет ограниченный в деле создания, описания и сравнения возможностей языков. Языков множество: одна и та же информация одновременно может быть представлена многими различными разговорными языками естественного происхождения, языками жестов, языками отдельных областей знаний (язык математики и логики), языками программирования. В зависимости от языка передаваемая информация может быть закодирована и передана лицам, соответственно владеющим английским, французским и другими языками. Существуют подмножества языков (языков жестов), предназначенных для лётчиков и палубной команды на борту авианосца (aircraft handling officers) в составе (список прилагается):

- Команда катапульты и сдерживающих сетей. (Catapult and arresting gear crews).
- Авиатехники. (Air wing maintenance personnel).
- Команда осуществляющая чек-проверку самолетов (Air wing quality control personnel).
- Погрузчики (Cargo-handling personnel).
- Команда оборудования наземной поддержки. (Ground support equipment (GSE) troubleshooters).
- Команда отвечающая за улавливающий трос (Hook runners).
- Фотографы и операторы. (Photographer's mates).
- Команда обслуживающая посадку вертолетов. (Helicopter landing signal enlisted personnel (LSE).
- Инспекция по качеству. (Quality Assurance (QA))
- Инспектор самолетов эскадрильи. (Squadron plane inspectors)
- Служба наземных сигналов. (Landing signal officer (LSO))
- Офицеры по воздушному транспорту. (Air transfer officers (ATO))
- Команда работающая с сжиженным кислородом. (Liquid oxygen (LOX) crews)
- Смотрящий за безопасностью на палубе. (Safety observers)
- Медики. (Medical personnel)
- Оружейники. (Ordnancemen)
- Спасатели. (Crash and salvage crews)
- Палубные взрывотехники. (Explosive ordnance disposal (EOD))
- Пожарные. (Firefighter)
- Авиа регулировщики-стажеры. (Plane handlers (Trainees))
- Смотрители такелажа/чернорабочие. (Chocks and chains – entry-level flight-deck workers under the yellowshirts)
- Авиалифтеры. (Aircraft elevator operators)
- Палубные водители. (Tractor drivers)

- Связисты. (Messengers and phone talkers)
- Заправщики. (Aviation fuel handlers)
- Механики эскадрильи. (Air wing plane captains: squadron personnel that prepare aircraft for flight)
- Инспектор финальной проверки. (Final checker (inspector)).
- Палубные регулировщики. (Traffic controller)

Например, инспектор финальной проверки (Final checker inspector) жестами сообщает лётчику о состоянии самолёта при его подготовке к полёту, о ситуации на взлётной палубе, даёт "добро" на взлет самолета, жестами непосредственно руководит взлётом. Лётчик жестами подтверждает, что он понимает сигналы инспектора и старается выполнять его команды. О возможно возникающих проблемах на борту самолёта лётчик жестами (тот же язык но со стороны лётчика) сообщает об этом палубной команде. Также лётчик сообщает о готовности к полёту, и непосредственно перед взлётом благодарит персонал палубной команды. Соблюдение правил языка (грамотное общение) позволяет поддерживать порядок на ограниченной по размерам палубе авианосца большому количеству самолётов, находящихся в разном состоянии и степени готовности к полёту.

Также существуют языки (языки программирования), которые позволяют закодировать и донести информацию от программиста до компьютера. В этом случае цель программиста — «внятно и понятно» изъясняться с компьютером через специальные выполняемые компьтером программы (интерпретаторы и компиляторы) так, чтобы компьютер "понимал" программиста, а программисту в ходе общения (решения задачи) были бы понятны сообщения компьютера для данной задачи, платформы и операционной системы. Для многих конкретных случаев (задач и систем) хороши определённые языки, а всего их более 1 000. Языков программирования так много, потому что программисты постоянно находятся в поисках новых инструментов и возможностей, чтобы упростить и сделать более эффективным процесс разработки программного обеспечения. Поэтому создаётся много узкоспециализированных языков, написанных специально под определённую задачу и область знания, а широко известные универсальные языки программирования совершенствуются и обновляются.Изучить их все невозможно, однако можно успешно работать с несколькими языками или даже с одним из них.

Примеры трёх программ (скриптов), «понятных» (хотелось, чтобы это было именно так) программисту при описании задачи и компьютеру в результате определённых регламентированных работ по созданию и выполнению кода. На различных языках программирования выводятся на экран надписи «Hello, world!»

Язык python

```
#print("Hello, world!")Pythonusing System;
# This is a sample Python script.
# Press Shift+F10 to execute it or replace it with your code.
# Press Double Shift to search everywhere for classes, files, tool windows,
# actions, and settings.
# def print_hi(name):
def print_hi(WRLD):
    # Use a breakpoint in the code line below to debug your script.
    #print(f'Hi, {name}')  # Press Ctrl+F8 to toggle the breakpoint.
    print(f'Hi... {WRLD}')  # Press Ctrl+F8 to toggle the breakpoint.
# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    #print_hi('PyCharm')
    print_hi('Hello, world')
```

Язык C#

```
package demo

fun main(args : Array)

 {

 println("Hello, world!")

 }
```

Язык C++

```
#include <iostream>
using namspace std;
// class definition
class Message
{
 public:
    void display()
    {
      cout << “Hello, world!”;
    }
}

int main()
 {
     message m;
     m.display();     // create an object
     return 0;        // call function
 }
```

■ Структура языка

Подлежащая кодированию информация является основным условием создания и выполнения программ: невозможно что-либо описывать при отсутствии информации, подлежащей описанию. Если нечего описывать — не будет языка.

При этом недостаточно ПРОСТО закодировать информацию на языке программирования. Программный код (результат описания) должен быть кратким, легко читаемым и максимально понятным.

При написании больших программ очень быстро наступает момент, когда основное время тратится на её повторное чтение и модификацию (отладку), а не на разработку. Поддерживать большие объёмы кода сложно. Одной из задач программирования является управление сложностью кода. Для упрощения работы с кодом программа разбивается на модули, классы, методы. В управлении сложностью также помогает правильное именование переменных.

■ Выражения python

Далее описываются важные особенности языка программирования python, которые касаются работы с кодом.

До начала этапа программирования информация, которая должна быть закодирована, представляет собой множество слов естественного языка. Задание по написанию программы понято, представлено в виде схем на основе связного текста, описано на естественном языке (русском, английском, французском, и т.д.).

Этот текст представляет собой множество понятий, объединённых по правилам грамматики языка в связные осмысленные предложения. Корректное описание задачи на естественном языке является одним из условий её успешного перевода на язык программирования.

Здесь не обсуждается грамматика естественного языка. Важно, что на основе грамматики из исходных понятий формируется описание задачи, возможно, графическое представление алгоритма (блок-схема), и её решения.

Далее исходное описание на естественном языке, возможно, представленное в виде схем (тоже язык!) переводится программистами на язык программирования (ещё один язык) и в результате серии переводов позволяет получить готовый к выполнению программный код (это работа компьютера).

Перевод, соответствующий описанию задачи, производится по правилам языка программирования на основе грамматики этого языка.

Аналог предложения естественного языка в python - выражение. Результат перевода. Основа любого программного кода. Выражение состоит из множества операндов (переменных и констант), собранных на основе правил грамматики python из исходных символов языка. Операнды, в свою очередь, (опять же по правилам грамматики!) объединяются в предложения (выражения) с помощью операторов и команд. Команды — это операнды с ранее определёнными значениями. Обычно это имена функций. В выражении команды обеспечивают выполнение определенных действий.

Примеры:

```
[python]length = 5


breadth = 2
```

```
area = length * breadth

print('Площадь равна', area)

print('Периметр равен' , 2 * (length + breadth))[/python]
```

Здесь операторами являются '=' (присвоить значение), символы сложения и умножения (математика), команда 'print' (вывод на экран). Операнды – это переменные length, breadth и area, а также их значения (2 и 5). Выражения – последовательности операндов произвольной длинны, связанные одним или несколькими операторами.

Имена

Для переменных и констант в python можно придумать множество названий, но существует несколько правил, которые нужно соблюдать при написании кода.

Имена python

Правила для переменных в python:

- Имя переменной должно начинаться с буквы (test) или символа подчеркивания (_test);
- Имя переменной не может начинаться с цифры;
- Имя переменной может содержать только буквенно-цифровые символы и подчеркивание (Az, 0-9 и _);
- Имена переменных чувствительны к регистру (test, Test и TEST — три разные переменные);
- Имя переменной не должно совпадать с зарезервированным ключевым словом;

Список всех зарезервированных в python ключевых слов можно посмотреть так:

```
#    Это первая программа на языке python
import keyword
print(keyword.kwlist)
```

В результате выполнения этой программы (скрипта) выводится список следующего содержания (приводится с комментариями):

False	ложь.
True	правда.
None	"пустой" объект.
and	логическое И.
with / as	менеджер контекста.
assert	условие - возбуждает исключение, если условие ложно.
break	выход из цикла.
class	пользовательский тип, состоящий из методов и атрибутов.
continue	переход на следующую итерацию цикла.
def	определение функции.
del	удаление объекта.
elif	в противном случае, если.
else	см. for/else или if/else.
except	перехватить исключение.
finally	вместе с инструкцией try, выполняет инструкции независимо от того, было ли исключение или нет.
for	цикл for.

from импорт нескольких функций из модуля.

global позволяет сделать значение переменной, присвоенное ей внутри функции, доступным и за пределами этой функции.

if если.

import импорт модуля.

in проверка на вхождение.

is ссылаются ли 2 объекта на одно и то же место в памяти.

lambda определение анонимной функции.

nonlocal позволяет сделать значение переменной, присвоенное ей внутри функции, доступным в объемлющей инструкции.

not логическое НЕ.

or логическое ИЛИ.

pass ничего не делающая конструкция.

raise возбудить исключение.

return вернуть результат.

try выполнить инструкции, перехватывая исключения.

while цикл while.

yield определение функции-генератора.

■ Информация о модуле keyword...

keyword.kwlist список всех доступных ключевых слов.

keyword.iskeyword(строка) является ли строка ключевым словом.

■ Нотация в Python

Нотация в Python: CamelCase или under_score?

Нотация — это соглашение об именовании. Наиболее популярные нотации в программировании — camel case и under score.

camelCase (еще называется "верблюжья нотация") — это стиль, в котором слова пишутся слитно, а каждое слово начинается с прописной (большой) буквы. Имеется два подвида этого стиля: lowerCamelCase (все слова кроме первого начинаются с прописной буквы)

и

UpperCamelCase (все слова, в том числе и первое пишутся с большой буквы).

under_score (snake_case) — при использовании этого стиля в качестве разделителя используется символ подчеркивания '_', а все слова начинаются со строчной (маленькой) буквы;

Существует также стиль kebab-case, который похож на under_score, но вместо знака подчеркивания '_' в нем используется тире '-'.

В Python преимущественно используется нотация under_score

Одно из правил Zen of Python (PEP20) — "читаемость имеет значение". Нотация under_score удобна для чтения и соответствует этому правилу.

При этом under_score — не единственная нотация, рекомендуемая к использованию в python. Вот руководство по именованию (тип, варианты нотаций), основанное на рекомендациях автора языка python Гвидо ван Россума:

Type	Public	Internal
Packages	lower_with_under	
Modules	lower_with_under	_lower_with_under
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_with_under	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected)
Method Names	lower_with_under	_lower_with_under() (protected)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

■ Как выбирать имена в Python

Основной принцип хорошего именования — имена должны быть содержательными (полностью отражать своё назначение). Перед тем, как дать имя переменной, функции или классу, желательно ответить на вопросы:

- Почему эта переменная (функция, класс и т.п.) существует?
- Что она делает? (или какие данные в ней хранятся?)
- Как используется?

Пример плохого именования:

```
"""
Объявление функции area
"""
def area(side1, side2):
    return side1 * side2

"""
Применение функции area
"""
```

```
d = area(4, 5)
```

В данном примере переменная d ничего не обозначает. По названию функции area не понятно, что она делает, какие параметры принимает и что возвращает.

Пример хорошего именования:

```
"""
Объявление функции get_rectangle_area
"""
def get_rectangle_area(length, width):
    return length * width

"""
Применение функции get_rectangle_area
"""

area = get_rectangle_area(4, 5)
```

В примере понятно что делает функция get_rectangle_area (получив длину и ширину, функция вычисляет и возвращает площадь прямоугольника). Результат записывается в переменную area (площадь).

Имя содержательное понятное, легко читается и не требует дополнительных комментариев.

Еще несколько рекомендаций по выбору имён в Python:

Рекомендуется подбирать имена на английском языке. Не нужно использовать русские имена для переменных и писать их транслитом. Важно знать английский язык.

```
def proverka_zdorovya():  # плохо
    pass

def health_check():  # хорошо
    pass
```

Для названия функций надо использовать глагол (например "купить", "получить", "распечатать"), для переменных, модулей и классов надо использовать существительное (например "список", "покупатели", "товары").

Для названия функций не следует использовать существительные:

```
def speed_calculator(distance, time):  # неправильно (калькулятор скорости)
    pass

def calculate_speed(distance, time):  # правильно (рассчитать скорость)
    pass
```

Имена функций должны быть как можно короче. Функция выполняет всего одно действие и именно оно должно отображаться в её имени.

```
def create_text_utf8_pipe_from_input():  # плохо
    pass
```

```
def create_pipe():  # хорошо
    pass
```

Перед именами булевских переменных (и методов, возвращающих тип bool), добавлять префикс "is".

Там где не получается использовать "is", можно использовать "has" или "can". Пример:

```
has_children()
...
```

Нужно использовать удобопроизносимые имена. Удобно работать со словами. Часть мозга специализируется на обработке слов — нужно использовать эту область мозга.

```
# пример плохого именования (имена трудно произносить вслух)
class GlobalFRTEV:  # плохо
    def get_any_mdhd(self):  # плохо
        pass

    def get_any_mdhl(self):  # плохо
        pass

    def get_any_mdhf(self):  # плохо
        pass
```

Имена будут использоваться в обсуждении с коллегами. Будет сложно рассказывать про баг в методе get_any_mdhf класса GlobalFRTEV.

Не рекомендуется использовать однобуквенные переменные. Их можно использовать исключительно для локальных переменных в небольших методах.

```
r = requests.get('https://pythonchik.ru')  # плохо

response = requests.get('https://pythonchik.ru')  # хорошо
```

Не нужно разрабатывать и использовать свою систему кодирования имен. Такие имена трудно произносить и в них можно сделать опечатку. К тому же, каждый новый программист должен будет изучать эту новую систему кодирования.

```
tm1_word = "Hello"  # плохо
```

Также при выборе имени файла и кодировании имён не следует использовать юмор, каламбуры и сленг.

Как правило, код читают программисты, поэтому в именах допустимо использование слов из профессиональной области, например, названия алгоритмов, паттернов или математические термины. Также возможно использование слов из предметной области, к которой принадлежит решаемая задача.

Больше полезных сведений о выборе имен идентификаторов и другие советы, которые пригодятся при написании кода, можно найти в книгах:

Роберт Мартин "Чистый код" ("Содержательные имена", стр. 39);

Стив Макконнелл "Совершенный код" ("Сила имен переменных", стр. 252).

Именование переменных в PEP8

В стандарте PEP8 описаны следующие стили именования:

- b — идентификатор из одной маленькой буквы;
- B — идентификатор из одной большой буквы;
- lowercase — одно слово в нижнем регистре;
- lower_case_with_underscores — нескольких слов из маленьких букв, между словами подчеркивания;
- UPPERCASE — одно слово заглавными буквами;
- UPPERCASE_WITH_UNDERSCORES — слова из заглавных букв, между словами подчеркивания;
- CapitalizedWords — несколько слов без пробелов между ними, но каждое начинается с прописной буквы (CamelCase);
- mixedCase — несколько слов без пробелов между ними, первое слово пишется с маленькой буквы, каждое последующее — с большой;
- Capitalized_Words_With_Underscores — имена переменных состоят из нескольких слов, которые начинаются с большой буквы, разделенных между собой знаком подчеркивания "_".

О стандартах именования читать в PEP8 (раздел "Naming Conventions").

Для удобства, среды разработки (например PyCharm) автоматически проверяют, насколько код соответствует рекомендациям стандарта PEP8. Если имя идентификатора не будет соответствовать соглашениям, то IDE подчеркнет переменную, а если навести на нее мышиный курсор, появится сообщение с подсказкой.

Хороший код документирует сам себя. При правильном именовании не возникает вопросов по тому, что происходит в отдельных частях кода. И отпадает необходимость писать комментарии к каждой строке(???).

И ещё соображения по поводу именования — это PEP8 recommends using...

lower_with_under в нижнем регистре со словами, разделенными символами подчеркивания, что необходимо для улучшения удобочитаемости имен переменных и функций.

Большинство людей интерпретировало это как lower_case_with_underscores, хотя на практике и в собственных методах python кажется, что строчные буквы без подчеркивания более популярны.

Кажется, что строгое следование PEP8 было бы неудобным, поскольку оно предлагает смешивать как lower_case_with_underscores, так и lowercasewithoutunderscores, что было бы непоследовательно.

Типы и классы

python — объектно-ориентированный язык программирования. Почти всё, что реализовано и программируется в python — это созданные на основе данных различных типов по ранее определённым схемам с их свойствами и методами множества взаимодействующих объектов-представителей данных классов.

Типы данных в Python

Данные в python являются объектами. Они могут либо создаваться 'вручную' в ходе выполнения программного кода, либо быть изначально встроенными на уровне языка. Объект можно охарактеризовать как особую (особо организованную) область памяти, где хранятся некоторые значения и операции, определённые для этих значений.

Структура программы общего вида на языке python:

- Программа состоит из модулей;
- Модуль представляет собой набор инструкций;
- Инструкции содержат выражения;
- Выражения служат для создания и обработки объектов;

Таким образом,

- Программа;
- Модуль;
- Инструкция;
- Выражение;
- Объект;

python подразумевает обязательное определение типа данных для переменных, констант, массивов, списков и т.д. Практически все конструкции языка являются объектами конкретных ранее объявленных типов. Таким образом, объекты можно классифицировать по их типам.

Динамическая типизация

Среди множества языков программирования можно выделить две составляющие:

- типизированные языки;
- нетипизированные (бестиповые) языки;

Нетипизированные языки в основном являются языками низкого уровня. Программы на этих языках напрямую взаимодействует с 'железом'. Так как компьютер 'мыслит' нулями и единицами, различия между строкой (типом) и, например, целочисленным значением будут заключаться лишь в наборах этих самых 0 и 1. В связи с этим, внутри бестиповых языков, близких к машинному коду, возможны любые операции над любыми данными.

python является типизированным языком. В нём определено понятие 'типа', существуют операции распознавания и верификации этих самых 'типов'.

Таким процессом является типизация. В ходе её выполнения происходит подтверждение используемых типов и применение к ним соответствующих ограничений. И от этого зависит корректность выполнения программы. Типизация может быть статической и динамической. В

первом случае проверка выполняется во время компиляции, во втором — непосредственно во время выполнения программного кода.

python — язык с динамической типизацией. В нём одна и та же переменная при многократной инициализации может представлять объекты разных типов:

```
a = 1
print(type(a))
<class 'int'>

a = 'one'
print(type(a))
<class 'str'>

a = {1: 'one'}
print(type(a))
<class 'dict'>
```

■ Достоинства динамической типизации

К достоинствам динамической типизации можно отнести:

- Возможность создания разнородных коллекций;
- Простота абстрагирования в алгоритмах;

■ *Создание разнородных коллекций*

Благодаря тому, что в python типы данных проверяются непосредственно во время выполнения программного кода, появляется возможность создавать коллекции, состоящие иэ элементов разных типов. Причём делается это легко и просто:

```
# список, элементами которого являются строка, целое число и кортеж
variety_list = ['String', 42, (5,25)]
```

■ *Абстрагирование в алгоритмах*

Создавая код на python, можно предположить универсальную функцию сортировки и не писать отдельную её реализацию для строк и чисел, поскольку она и так корректно отработает на любом компарируемом (сравниваемом) множестве.

■ Недостатки динамической типизации

Ошибки типизации и логические ошибки на их основе, а также проблемы оптимизации.

■ *Ошибки типизации*

Они редки, однако весьма сложно отлавливаемые. Вполне реальна ситуация, когда разработчик написал функцию, предполагая, что она будет принимать числовое значение. Но в результате ошибок, возникающих в других частях программы, на вход функции поступает строка, которая отрабатывается без ошибок выполнения, однако её результат, — ОШИБКА, сам по себе. Статическая же типизация исключает такие ситуации.

■ *Оптимизация*

Статически типизированные языки обычно работают быстрее динамических, поскольку являются более "тонким" инструментом, оптимизация которого, в каждом конкретном случае, может быть настроена более тщательно и рационально.

Классификация типов данных python

По одной из классификаций типов python делятся на атомарные и ссылочные. Атомарные объекты, при их присваивании, передаются по значению, а ссылочные — по ссылке

Атомарные:

- Числа;
- Строки;

Ссылочные:

- Списки;
- Кортежи;
- Словари;
- Функции;
- Классы;

Таким образом, типы python отличаются от типов в других языках с жестко заданной типизацией. Важные типы в python:

- Числовые: целые, дробные, вещественные с плавающей точкой, комплексные.
- Логические: тип для хранения значений алгебры логики – 'истина' или 'ложь'.
- Строковые: содержат символы Юникода, в том числе, html-код.
- Списки – упорядоченные массивы данных.
- Кортежи – массив упорядоченных констант, т.е. значений, которые не могут изменяться в процессе работы.
- Множества – массивы неупорядоченных данных.
- Словари – специализированный массив, состоящий из пары – 'ключ' — 'значение'.
- Байты, массивы байтов – поименованные области памяти для хранения изображений (jpg, gif и т.д.), pdf-документов и других файлов.

Числовые типы python

Числа — важнейший из всех типов данных для всех языков программирования. В python для их представления служит числовой тип данных.

int (целое число)

Концепция целых чисел - всё просто: это числа без дробной части, которые являются расширением натурального ряда, дополненного нулём и отрицательными числами.

```
# примеры целых чисел
a = -3000
b = 0
c = 9000
```

Целые числа используются для исчисления всевозможных математических выражений. Также int применяется в качестве описаний количественных свойств какого-либо объекта.

■ *float (число с плавающей точкой)*

Действительные или вещественные числа придуманы для измерения непрерывных величин. В отличие от математического контекста, ни один из языков программирования не способен реализовать бесконечные или иррациональные числа, поэтому всегда актуально приближению с определенной точностью, из-за чего возможны такие ситуации:

```
print(0.3 + 0.3 + 0.3)
> 0.8999999999999999

print(0.3 * 3 == 0.9)
> False
```

Записи float не отличаются от записи int:

```
# примеры вещественных чисел
zero = 0.0
pi = 3.14
e = 2.71
```

Область применения float — математические вычисления.

■ *complex (комплексное число)*

Вещественный ряд расширяет множество рациональных чисел. Ряд комплексных чисел расширяет множество вещественных. Принципиальной особенностью комплексного ряда является возможность извлечения корня из отрицательных чисел.

В python комплексные числа задаются с помощью функции complex():

```
# пример комплексного числа
z = complex(1, 2)
print(z)
> (1+2j)

# вещественная часть
print(z.real)
> 1.0

# мнимая часть
print(z.imag)
> 2.0

# сопряженное комплексное число
print(z.conjugate())
> (1-2j)
```

ВАЖНО! Для комплексных чисел операция сравнения не определена:

```
z1 = complex(4, 5)
z2 = complex(100, 200)
print(z1 > z2)

>
Traceback (most recent call last):
  print(z1> z2)
TypeError: '>' not supported between instances of 'complex' and 'complex'
```

Комплексные числа широко применяются, например, для решения дифференциальных уравнений.

■ *bool (логический тип данных)*

Самый простой и понятный из всех типов данных python. У bool есть всего два значения:

- Истина (True);
- Ложь (False).

Для булевой алгебры этих значений достаточно.

```
# пример bool
pravda = True
lozh = False
```

Переменные логического типа нужны для реализации ветвлений, они применяются для установки флажков, фиксирующих состояния программы, а также используются в качестве возвращаемых значений для функций, названия которых, зачастую, начинаются на "is" (isPrime, isEqual, isDigit).

То есть тех, которые, на человеческом языке, отвечали бы на вопрос одним словом "Да" или "Нет".

■ Последовательности

Ещё одно понятие из математики, где последовательность — есть нумерованный набор элементов, в котором возможны их повторения, а порядок имеет значение.

Последовательность в python : упорядоченная коллекция объектов.

■ *str (строка)*

Строки, - единственный тип, который по частоте применения может сравниться с числовым типом данных. Определение, справедливое для python звучит так:

строка — это последовательность односимвольных строк.

```
s = 'Hello, friend. You are my world'
print(type(s))

> <class 'str'>
```

Важность строк велика в первую очередь для людей: что вся письменная речь может рассматриваться, как множество строк. А так как человеку свойственно обмениваться информацией именно в виде набора слов, то можно говорить о неограниченном количестве областей применения строкового типа данных.

■ *list (список)*

Список — это ещё один вид последовательностей... Здесь стоит остановиться и отметить, что последовательности в Python бывают изменяемыми и неизменяемыми. Список — изменяемая

последовательность, а строки и кортежи — нет. Таким образом, список можно определить, как упорядоченную и изменяемую коллекцию, состоящую из объектов произвольных типов.

```
# пример списка
list_of_lists = [['code alpha', 'code beta'], [553, 434]]
list_of_lists[0][1] = 'code omega'

print(list_of_lists)
> [['code alpha', 'code omega'], [553, 434]]
```

Список это объект для хранения наборов данных.

- ***tuple (кортеж)***

Кортежи в языке python можно рассматривать, как неизменяемые списки со всеми вытекающими последствиями:

```
# пример кортежа
tup = ('i', 'j')
# мы можем получить первый элемент
print(tup[0])
> i

# но изменить его не получится
tup[0] = 'k'

>
Traceback (most recent call last):
  tup[0] = 'k'
TypeError: 'tuple' object does not support item assignment
```

Использование кортежей оправдано, когда разработчику важна скорость работы или неизменяемость элементов последовательности.

- ***dict (словарь)***

Словари хоть и являются набором данных, однако не считаются последовательностью, потому как представляют собой неупорядоченный набор пар ключ:значение.

```
# пример простого словаря
dictionary = {'Огонёк': 'уменьш. от слова 'Огонь'}
```

Применяются они, когда для работы требуется тип данных концептуально схожий с обычным телефонным справочником, где каждая запись — есть пара сопоставленных друг с другом значений, по одному из которых (уникальному ключу) можно получить второе (собственно, значение).

- ***set (множество)***

Ещё один "набор, но не последовательность".

```
# пример множества
integer_num_set = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}
```

Если не важен порядок элементов, но важна их уникальность, то подходит множество.

```
# свойство уникальности
unique_set = {6, 6, 6, 5}

print(unique_set)
> {5, 6}
```

■ Файл

Работа с файлами, хранящимися где-то на внешнем носителе, в Python реализована в виде объектов-файлов. Они относятся к объектам базового типа, но обладают весьма характерной чертой: нельзя создать экземпляр объекта-файла при помощи литералов.

Чтобы начать работу с файлами, нужно вызвать функцию open() и передать ей в качестве аргументов имя файла из внешнего источника и строку, описывающую режим работы функции:

```
f = open('filename.txt', 'w')
```

Операции с файлами могут быть разными, а, следовательно, разными могут быть и режимы работы с ними:

- r — выбирается по умолчанию, означает открытие файла для чтения;
- w — файл открывается для записи (если не существует, то создаётся новый);
- x — файл открывается для записи (если не существует, то генерируется исключение);
- a — режим записи, при котором информация добавляется в конец файла, а не затирает уже имеющуюся;
- b — открытие файла в двоичном режиме;
- t — ещё одно значение по умолчанию, означающее открытие файла в текстовом режиме;
- + — чтение и запись;
- range object (a type of iterable).

В языке python реализована функция range(), которая создаёт непрерывную последовательность целых чисел:

```
r = range(10)
print(r)
> range(0, 10)

for i in r:
    print(i, end=' ')
> 0 1 2 3 4 5 6 7 8 9
```

Она очень удобна при создании циклов.

■ None

None — специальный тип языка python. Объект этого типа означает пустоту, всегда имеет значение False и может рассматриваться в качестве аналога NULL для языка C/C++. Объект None возвращается функциями по умолчанию.

```
null = None
```

```
print(type(null))
> <class 'NoneType'>
```

На практике этот тип данных может быть полезен, когда надо заполнить список отсутствующими значениями, чтобы он увеличился, и можно было обращаться к старшим элементам по индексу.

■ Работа с типами в python

Далее описываются некоторые простые (???) приёмы работы с данными разных типов в python.

■ *Определение типа данных*

Узнать тип данных объекта в python можно следующим способом:

```
# Нужно воспользоваться встроенной функцией type()
my_list = {'Python': 'The best!'} # объявление объекта - словаря

print(type(my_list))
> <class 'dict'>
```

■ *Изменение типа данных (приведение типов)*

Встроенные функции для приведения типов — int(), list(), set(), tuple(), str(), bin().

Важно! встроенная функция для приведения типа не модифицирует переданное значение, а возвращает новое значение другого типа.

```
# int() - преобразует числовую строку в целое число
seven = '7'
new_seven = int(seven)

print(new_seven, type(new_seven))
> 7 <class 'int'>
# list() - приведение итерируемого объекта к списку
nums = (11, 12)
new_nums = list(nums)

print(new_nums, type(new_nums))
> [11, 12] <class 'list'>
# set() - трансформация итерируемого объекта во множество
vegetables = ['carrot', 'tomato', 'cucumber']
new_vegetables = set(vegetables)

print(new_vegetables, type(new_vegetables))
> {'cucumber', 'carrot', 'tomato'} <class 'set'>
# tuple() - аналогично, но в кортеж
python = 'Python'
new_python = tuple(python)

print(new_python, type(new_python))
> ('P', 'y', 't', 'h', 'o', 'n') <class 'tuple'>
# str() - приведение к строковому типу
ex_dict = {'qiwi': 1453}
new_ex_dict = str(ex_dict)

print(new_ex_dict, type(new_ex_dict))
> {'qiwi': 1453} <class 'str'>
# bin() - преобразует десятичное число в двоичный формат
```

```
dec = 10
new_dec = bin(dec)

print(new_dec)
> 0b1010
```

Отличие type() от isinstance()

В отличие от type(), функция isinstance() возвращает не тип данных аргумента, а булево значение, говорящее о том, принадлежит объект к определенному классу или нет:

```
num = 4.44
print(isinstance(num, float))
> True
```

isinstance() также умеет проверять принадлежность объекта хотя к одному типу из кортежа, переданного в качестве второго аргумента:

```
name = 'Ash'
print(isinstance(name, (float, int, complex)))
> False

print(isinstance(name, (float, int, complex, str)))
> True
```

Важным отличием также является то, что isinstance() "знает" о наследовании. Функция воспринимает объект производного класса, как объект базового.

```
class BaseExample:
    pass

class DerivedExample(BaseExample):
    pass

test = DerivedExample()
print(isinstance(test, BaseExample))

> True
А вот вывод результата работы функции type():


print(type(test))
> <class '__main__.DerivedExample'>
```

Здесь видно, что для type() объект test является объектом-представителем класса DerivedExample.

Классы

Класс — это тип данных.

У класса есть свойства и функции (в ООП их называют методами).

- Свойства (поля, атрибуты) — это характеристики, присущие данному конкретному множеству объектов.
- Методы — действия, которые они могут совершать.

Принципы ООП

Абстрагирование, полиморфизм, наследование, инкапсуляция... Классы подобны чертежам: это не объекты, а их схемы. За счёт принципов ООП класс "банковских счетов" имеет строго определенные и одинаковые для всех атрибуты, но его объекты (сами счета) уникальны.

Далее обсуждаются вопросы, связанные с принципами ООП, объявлением классов, методов и созданием объектов, представляющих эти классы.

Абстракция

Абстракция — это выделение основных, наиболее значимых характеристик объекта и игнорирование второстепенных.

Любой "составной" объект реального мира — это абстракция. Говоря "ноутбук", не требуется дальнейших пояснений, вроде того, что это организованный набор пластика, металла, жидкокристаллического дисплея и микросхем.

Абстракция позволяет игнорировать нерелевантные детали, поэтому для сознания человека это один из способов справляться со сложностью реального мира. Если бы, подходя к холодильнику, нужно было бы "иметь дело" с отдельно металлом корпуса, пластиковыми фрагментами, лакокрасочным слоем и мотором, достать из морозилки замороженную клубнику было бы намного сложнее.

Полиморфизм

Полиморфизм подразумевает возможность нескольких реализаций одной идеи. Простой пример: есть класс "Персонаж", а у него есть метод "Атаковать". Для воина это будет означать удар мечом, для рейнджера — выстрел из лука, а для волшебника — чтение заклинания "Огненный Шар". В сущности, все эти три действия — атака, но в программном коде они будут реализованы совершенно по-разному.

Наследование

Это способность одного класса расширять понятие другого, и главный механизм повторного использования кода в ООП. На уровне абстракции "Автотранспорт" можно не учитывать особенности каждого конкретного вида транспортного средства, а рассматривать их "в целом". Если же более детализировано приглядеться, например, к грузовикам, то окажется, что у них есть такие свойства и возможности, которых нет ни у легковых, ни у пассажирских машин. Но, при этом, они всё ещё обладают всеми другими характеристиками, присущими автотранспорту.

Можно было бы объявить отдельный класс "Грузовик", который является наследником базового класса "Автотранспорт". Объекты этого класса могли бы определять все прошлые атрибуты (цвет, год выпуска), но и получить новые. Для грузовиков это могли быть грузоподъёмность, снаряженная масса и наличие жилого отсека в кабине. А методом, который есть только у грузовиков, могла быть функция сцепления и отцепления прицепа.

Инкапсуляция

Инкапсуляция — это принцип ООП, который требуется для безопасности и управления сложностью кода. Инкапсуляция блокирует доступ к деталям сложной концепции. Абстракция

подразумевает возможность рассмотреть объект с общей точки зрения, а инкапсуляция не позволяет рассматривать этот объект с какой-либо другой.

■ Методы класса

Метод — это функция, объявляемая в классе.

Например, у боевого корабля (броненосец, линкор) есть бортовое оружие. И оно может стрелять.

```
class WarShip:
    def atack(self):
        print('Ahh!')

destroyer = WarShip()
destroyer.atack()

> Ahh!
```

■ Атрибуты класса

У класса имеется собственный набор характеристик, который позволяет описать его сущность. Эти свойства задаются на этапе объявления класса и называются полями или атрибутами класса.

Поля в классе могут быть объявлены статическими или динамическими.

Статические поля (поля класса) можно использовать без создания объекта. Значения, которые записываются в эти поля (статические переменные) актуальны для всего множества объектов (которые объявлены и которые могут быть объявлены в ходе выполнения программного кода). Статические поля объявляются непосредственно в классе.

Динамические поля (поля объекта) задаются при создании конкретных объектов с помощью специального метода-конструктора. Информация, которая записывается в поля объекта, актуальна только для конкретного объекта. Экземпляр нужно создать, а полям присвоить значения. Динамические поля объявляются в теле метода-конструктора __init__.

Объявление класса с именем firstClass и статическим атрибутом (свойством) x:

```
class firstClass:
    x = 5

print(firstClass.x)
```

Объявление класса со статическим атрибутом и динамическим атрибутом, а также методом-конструктором __init__.

```
class MightiestWeapon:
    # статический атрибут
    name = "Default name"

    def __init__(self, weapon_type):
        # динамический атрибут
        self.weapon_type = weapon_type
```

Важно! статический и динамический атрибуты в классе могут иметь одно и то же имя:

```
class MightiestWeapon:
    # статический атрибут
    name = "Default name"

    def __init__(self, name):
        # динамический атрибут
        self.name = name


weapon = MightiestWeapon("sword")

print(MightiestWeapon.name)
print(weapon.name)
```

В том, что в классе объявлены атрибуты с одним и тем же именем, нет никаких проблем. Это всё равно разные идентификаторы. Полное имя статического атрибута MightiestWeapon.name (имя_класса.имя_атрибута), полное имя динамического атрибута weapon.name (имя_объекта.имя_атрибута).

На основе класса создаётся объект-представитель класса firstClass под названием p1. Функция print выводит значение поля x для объекта p1:

```
p1 = MyClass()
print (p1.x)
```

В отличие от многих объектно-ориентированных языков, в python не определены спецификаторы доступа к полям и методам класса. Отсутствие аналогов спецификаторов уровня доступа public/private/protected (все значения атрибутов и методы одинаково доступны) можно рассматривать как "упущение" со стороны принципа инкапсуляции.

■ Конструктор

Метод, который вызывается при создании объектов, называется конструктором. Он нужен для объектов, которые изначально должны иметь какие-то значение. Например, пустые экземпляры класса "Student" (Студент) бессмысленны, и желательно иметь хотя бы минимальный обозначенный набор значений вроде имени, фамилии и группы.

В качестве конструктора python обычно выступает метод __init__():

```
class Student:
    def __init__(self, name, surname, group):
        self.name = name
        self.surname = surname
        self.group = group

alex = Student("Alex", "Ivanov", "admin")
```

Таким образом, объявляемая по умолчанию функция инициализации __init__(...) . При объявлении класса встроенная функция __init__ может быть переопределена в соответствии с особенностями объявляемого класса.

Эта функция выполняется при создании объекта-представителя данного класса и обычно используется для присвоения значений свойствам (полям) объекта или выполнения других операций, которые выполняются при создании объекта. Например, при объявлении класса под названием Person и предназначенного для сохранения персональной информации, функция __init__() при создании объекта-представителя класса Person применяется для присвоения значений имени и возраста:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("Boris",  36)
print(p1.name)
print(p1.age)
```

Функция __init__() автоматически вызывается каждый раз при создании нового объекта-представителя данного класса.

В классе объявляются разнообразные методы. Методы — это функции, которые объявляются в классе и вызываются 'от имени' объектов-представителей данного класса.

У методов по умолчанию первым параметром является ссылка на объект. Он принимает в качестве значения ссылку на объект, от имени которого вызывается данный метод. Обычно этот параметр называется self.

Например, в классе Person, метод hellofunc(self) по значению ссылки self обеспечивает в форме приветствия вывод информации от имени данного объекта:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def  hellofunc(self):
        print("Hello, это "  + self.name)


p1 = Person("Boris",  36)
print(p1.name)
print(p1.age)
p1.hellofunc()
```

Таким образом, параметр self является ссылкой на объект, представляющий класс и используется для доступа к полям и методам объекта, объявленным в классе. Его не обязательно называть self, он может называться как угодно, но параметр-ссылка на объект должен быть первым параметром метода в классе, если этот метод вызывается от имени объекта-представителя класса (нестатический метод).

Принципы ООП позволяют легко изменять свойства ранее созданных объектов. Например, изменение возраста объекта p1 (значения поля age) с 36 на 40:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    def  hellofunc(self):
        print("Hello, это "  + self.name)


p1 = Person("Boris",  36)
print(p1.name)
print(p1.age)
p1.hellofunc()

p1.age = 40    # изменение возраста с 36 на 40
print(p1.age)
p1.hellofunc()
```

Ранее созданные объекты удаляются с использованием ключевого слова del.

■ Наследование

При написании кода может оказаться, что некоторые объекты-представители разных классов аналогичны аналогичны за исключением нескольких различий. Определение сходств и различий между такими объектами реализуется за счёт принципа ООП "наследованием". Несколько классов наследуют общему классу (классу-родителю).

```
# класс "Животное". Это достаточно абстрактный класс всего с одним методом
"Издать звук".
class Animal:
    def make_a_sound(self):
        print("Издаёт животный звук")
```

Известно что кошки лазают по деревьям, а собаки грызут всякие предметы. Далее объявляется пара соответствующих классов-наследников. Факт наследования в python указывается при объявлении класса-наследника. После имени класса в скобках указывается имя класса-родителя:

```
class Cat(Animal):
    def go_on_tree(self):
        print('Классная ветка! На ней может быть гнездо...')


class Dog(Animal):
    def gnaw_objects(self):
        print('Однажды я доберусь до хозяйской мобилы!')
```

Теперь объекты этих двух классов могут не только издавать животные звуки, но и выполнять собственные уникальные действия:

```
Tom = Cat()
Tom.make_a_sound()
> Издаёт животный звук

Tom.go_on_tree()
> Классная ветка! На ней может быть гнездо...!
```

Переопределение

Как кошка, так и собака просто "издают животные звуки", а хотелось бы, чтобы звуки были свойственные именно этим животным. Для этого существует механика переопределения, основанная на принципе полиморфизма. При этом в классе-наследнике объявляется метод с тем же названием, что и в базовом классе:

```
class Dog(Animal):
    def gnaw_objects(self):
        print('Однажды я доберусь до хозяйской мобилы!')
# далее для объектов класса "Собака" будет выполняться
# именно эта реализация метода
    def make_a_sound(self):
        print('Гав-гав!')

Balto = Dog()
Balto.make_a_sound()
> Гав-гав!
```

Объявление аргументов в функциях и методах классов

При вызове ранее объявленных функций или методов классов, особенно в случае наследования, параметры в функции и методы часто передаются в виде последовательностей, представляющих собой кортежи или словари. Для обозначения кортежей и словарей в объявлении аргументов функций и методов применяются специальные символы - приставки: "*" или "**" соответственно.

Это особенно полезно в случаях, когда функция / метод принимает переменное количество параметров.

В python принято, что в программе для передачи кортежа аргументов используется переменная args, а в случае со словарём — переменная kwargs. Если перед переменной args указан символ "*", то все дополнительные аргументы, переданные функции / методу, сохранятся в args в виде кортежа. Если перед kwargs указан символ "**", то все дополнительные параметры будут рассматриваться как пары "ключ - значение" в аргументе, представленном в виде словаря.

В функциях / методах, задающих свойства таких графических объектов как линия, текст, прямоугольник, параметры часто объединяют в виде последовательностей * args, либо словарей ** kwargs.

Считается, что это облегчает объявление классов, их свойств методов.

Документирование кода

Документирование кода — важная часть разработки на python. Связанной с кодом соответствующей документации может быть больше, чем самого кода. В документации разъясняется назначение и действия функции или класса, какие аргументы принимаются функциями и методами, какие значения возвращаются.

Когда документация и код находятся в разных местах, их сопровождение становиться затруднительным. Поэтому на практике документация находится непосредственно рядом (в одном файле) с кодом.

Docstring

Docstring — это строковый литерал, который располагается сразу за объявлением модуля, функции, класса или метода. О том, какие существуют соглашения в документировании Python кода описано в документации PEP257.

Документация для классов

Документация класса создается для самого класса, а также для его методов.

```
class Speaker:
    """Это docstring класса Speaker"""

    def say_something(self):
        """Это docstring метода"""

        print("something")
```

Правила документирования: после строки документации нужно оставлять пустую строку.

Документация класса

Документация для класса может содержать следующую информацию:

- краткое описание класса (+ его поведение);
- описание атрибутов класса;
- описание публичных методов;
- все, что связано с интерфейсом для подклассов.

Документация методов класса

Для методов класса документация может содержать:

- краткое описание метода (+ его поведение);
- описание аргументов метода;
- побочные эффекты (если таковые возникают при выполнении метода);
- исключения.

Далее — пример с более подробной документацией класса:

```
class TextSplitter:
    """Класс TextSplitter используется для разбивки текста на слова

    Основное применение - парсинг логов на отдельные элементы
```

```
по указанному разделителю.

Note:
    Возможны проблемы с кодировкой в Windows

Attributes
----------
file_path : str
    полный путь до текстового файла
lines : list
    список строк исходного файла

Methods
-------
load()
    Читает файл и сохраняет его в виде списка строк в lines
get_splitted(split_symbol=" ")
    Разделяет строки списка по указанному разделителю
    и возвращает результат в виде списка
"""

def __init__(self, file_path: str):
    self.file_path = file_path.strip()
    self.lines = []

def load(self) -> None:
    """Метод для загрузки файла в список строк lines

    Raises
    ------
    Exception
        Если файл пустой вызовется исключение
    """

    with open(self.file_path, encoding="utf-8") as f:
        for line in f:
            self.lines.append(line.rstrip('\n'))
        if len(self.lines) == 0:
            raise Exception(f"file {self.file_path} is empty")

def get_splitted(self, split_symbol: str = " ") -> list:
    """Разбивает текстовые строки lines, преобразуя строку в
    список слов по разделителю

    Если аргумент split_symbol не задан, в качестве разделителя
    используется пробел

    Parameters
    ----------
    split_symbol : str, optional
        разделитель
    """

    split_list = []
    for str_line in self.lines:
        split_list.append(str_line.split(split_symbol))
    return split_list
```

Документация для пакетов

Документация пакета размещается в файле __init__.py в верхней части файла (начиная с 1-й строки). В ней может быть указано:

- описание пакета;
- список модулей и пакетов, экспортируемых этим модулем;
- автор;
- контактные данные;
- лицензия.

```
"""
Пакет Mos помогает создать полноэкранный текстовый интерфейс в консоли.

Alex Ivanov [https://alex.ivanov.ru/]
alex.ivanov@gmail.com
# License: BSD
"""
__author__ = 'Alex Ivanov'

try:
    from .version import version
except ImportError:
    version = "0.0.0"

__version__ = version
```

Документация для модулей

Документация модулей аналогична документации классов. Вместо класса и методов в данном случае документируется модуль со всеми его функциями. Размещается в верхней части файла (начиная с 1-й строки).

Форматы Docstring

Строки документации могут иметь различное форматирование. В примере выше использовался стиль NumPy. Существуют и другие форматы:

- Google styleguide -> Comments and Docstrings
- Numpydoc docstring guide
- Epydoc
- reStructuredText (reST)

__doc__ и help(): вывод документации на экран
Строки документации доступны:

- из атрибута __doc__ для любого объекта;
- с помощью встроенной функции help().

Вывод документации с помощью функции help()

```
>>> import my_module
>>> help(my_module)

Help on module test:
```

```
NAME
    test - Это docstring модуля, он однострочный.

FILE
    /var/www/test.py

CLASSES
    MyClass

    class MyClass
     |  Это docstring класса.
     |
     |  Methods defined here:
     |
     |  my_method(self)
     |      Это docstring метода

FUNCTIONS
    my_function(a)
        Это многострочный docstring для функции my_function.

        В многострочном docstring первое предложение
        кратко описывает работу функции.
```

Также можно выводить документацию отдельного объекта:

```
>>> import my_module
>>> my_module.__doc__
>>> my_module.my_function.__doc__
>>> my_module.MyClass.__doc__
>>> my_module.MyClass.my_method.__doc__
```

■ pydoc

Для более удобной работы с документацией, в python существует встроенная библиотека pydoc. На основе python модулейpydoc автоматически генерирует документацию. Информацию по доступным командам модуля pydoc можно получить набрав в терминале:

```
python -m pydoc
```

Далее приводится описание функционала pydoc

■ *Вывод текста документации*

```
pydoc <name> –
```

покажет текст документации указанного модуля, пакета, функции, класса и т.д. Если <name> содержит "\", Python будет искать документацию по указанному пути.

Например, документация встроенного модуля math:

```
python -m pydoc math
Help on built-in module math:

NAME
    math
```

```
DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.
...
```

В консоль выведется название модуля, его описание и описание всех функций в модуле.

■ *Поиск по документации*

`pydoc -k <keyword> –`

найдет ключевое слово в документации всех доступных модулей.

Пусть требуется распаковать gzip файл. Поиск слова "gzip":

```
python -m pydoc -k gzip
_compression - Internal classes used by the gzip, lzma and bz2 modules
gzip - Functions that read and write gzipped files.
test.test_gzip - Test script for the gzip module.
```

В списке виден модуль gzip. Теперь можно посмотреть его документацию:

```
python -m pydoc gzip
Help on module gzip:

NAME
    gzip - Functions that read and write gzipped files.

DESCRIPTION
    The user of the file doesn't have to worry about the compression,
    but random access is not allowed.
```

По описанию, данный модуль может решить эту задачу.

■ HTTP сервер с документацией

Для удобства просмотра документации, pydoc позволяет одной командой создать HTTP-сервер:

```
python -m pydoc -p 331
Server ready at http://localhost:331/
Server commands: [b]rowser, [q]uit
server>
```

Теперь можно перейти в браузер и зайти на http://localhost:331/

Для остановки сервера ввести "q" и нажать "Enter":

```
server> q
Server stopped
```

Также HTTP-сервер доступен через

```
python -m pydoc -b -
```

эта команда создаст сервер на свободном порту, откроет браузер и перейдет на нужную страницу.

■ Запись документации в файл

```
python -m pydoc -w sqlite3 —
```

запись файла с документацией по модулю sqlite3 в html файл.

■ Автодокументирование кода

Для того чтобы облегчить написание документации и улучшить ее в целом, существуют различные python-пакеты. Один из них — pyment.

pyment работает следующим образом:

- Анализирует один или несколько скриптов.
- Получает существующие строки документации.
- Генерирует отформатированные строки документации со всеми параметрами, значениями по умолчанию и т.д.
- Далее можно применить сгенерированные строки к своим файлам.

Этот инструмент особенно полезен когда код плохо задокументирован, или когда документация вовсе отсутствует. Также pyment будет полезен в команде разработчиков для форматирования документации в едином стиле.

■ *Установка:*

```
pip install pyment
```

■ *Использование:*

```
pyment myfile.py    # для файла
pyment -w myfile.py  # для файла + запись в файл
pyment my/folder/  # для всех файлов в папке
```

Для большинства IDE также существуют плагины, помогающие документировать код:

- AutoDocstring – для VS Code.
- AutoDocstring – для SublimeText.
- Python DocBlock Package – для Atom.
- Autodoc – для PyCharm.

В PyCharm существует встроенный функционал добавления документации к коду. Для этого нужно:

- Переместить курсор под объявление функции.
- Написать тройные кавычки """ и нажать "Enter".

■ Сведения об операторах python

Операторы python применяются для формирования выражений из операндов

Большие выражения (с большим количеством операндов) можно форматировать. Для этого используется 'обратный слеш' (\).

Благодаря структуре 'лесенкой' язык позволяет проводить цепочечные сравнения.

Логические операторы просты с точки зрения синтаксиса и считаются 'ленивыми'.

Реализовано управление списками и прочими последовательностями. Есть индексация элемента в списке.

Строки выражений состоят из простых операторов, большинство из которых интуитивно понятны. При этом выражения формируются без лишних служебных скобок и других операторов типа 'начало' и 'конец'.

■ Операторы управления потоком выполнения

Выражения строятся из операндов с применением операторов. При разработке программного обеспечения возможно создание 'идеального' кода, который выполняется независимо от 'внешних' условий. На практике приходится создавать 'идеальный' код, который оказывается намного проще описываемых в реальном мире событий и работает совсем не так как это предполагалось. Например, нужно выполнить ряд выражений только в том случае, если соблюдаются определенные условия. Для обработки таких ситуаций в языках программирования (по аналогии с естественными языками) применяются операторы управления потоком выполнения. В зависимости от состояния выполняемых выражений (когда какое-то условие в программе истинно) операторы обеспечивают выполнение циклов или пропуск инструкций.

Оператор-выражение if.

Оператор-выражение if-else.

Оператор цикла while.

Оператор цикла for.

Оператор break.

Оператор continue.

■ if

Оператор if используется для проверки условия. Если условие истинно, запускается блок операторов (называемый блоком if), в противном случае обрабатывается другой блок операторов (называемый блоком else). Предложение else не является обязательным.

Синтаксис оператора if:

```
if condition:
    <indented statement 1>
    <indented statement 2>

<non-indented statement>
```

Первая строчка оператора if condition: — это условие if и логическое выражение condition, которое возвращает True или False. Далее располагается блок инструкций (блоком if). Он представляет собой одну или больше инструкций.

У каждой инструкции в блоке if одинаковый отступ от слова if.

Во многих языках (C, C++, Java, PHP,) для определения начала и конца блока используются фигурные скобки ({}). Для этого в Python используются отступы.

Каждая инструкция блока if должна содержать одинаковое количество пробелов. В противном случае интерпретатор Python возращает синтаксическую ошибку. В документации Python рекомендуется делать отступы на 4 пробела.

При выполнении инструкции if проверяется условие. Если условие истинно, тогда все инструкции в блоке if выполняются. Но если условие оказывается неверным, тогда все инструкции внутри этого блока пропускаются.

Инструкции следом за условием if, у которых нет отступов, к блоку if не относятся. Например, <non-intenden statement> (см. определение оператора выше) — это не часть блока if, поэтому она будет выполнена в любом случае.

Пример:

```
number = int(input("Введите число: "))

if number > 10:
    print("Число больше 10")
```

Следующий код в качестве демонстрации более тонкого применения оператора if:

```
# This is a sample Python script.

def print_str(number):
    print(f'number, {number}')
    if number > 10:
        print("первое уведомление")  # 3
        print("второе уведомление")  # 4
        print("третье уведомление")  # 5

    print("Этот код выполняется каждый раз при запуске программы")  # 7
    print("Конец")                                                   # 8

# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    number = int(input("Ввести число: "))
    print_str(number)
```

Первый вывод:

Введите число: 45

первое уведомление

второе уведомление

третье уведомление

Этот код выполняется каждый раз при запуске программы

Конец

Второй вывод:

Введите число: 4

Этот код выполняется каждый раз при запуске программы

Конец

Важно, что в приведённом примере к блоку if относятся только выражения на строках 3, 4 и 5.

И они будут выполнены в том случае, когда условие if будет истинно. Инструкции на строках 7 и 8 выполняются в любом случае.

При работе в консоли Python для разбития выражения на несколько строк используется оператор продолжение (\). Но в случае с операторами управления интерпретатор Python автоматически активирует мультистрочный режим, если нажать Enter после условия if.

При применении операторов управления непосредственно в консоли Python, её реакция будет другой:

```
>>>
>>> n = 100
>>> if n > 10:
...
```

Для многострочных инструкций консоль Python после нажатия Enter на строке с условием if преобразует командную строку с '>>>' на '...'. Таким образом она показывает, что начатая инструкция все еще не закончена.

Чтобы завершить инструкцию if, в блок if нужно добавить пустую инструкцию:

```
>>>
>>> n = 100
>>> if n > 10:
...     print("n v 10")
...
>>>
```

Python автоматически отступов не добавляет. Программист это делает самостоятельно., Чтобы после ввода инструкции на консоли сразу выполнить эту инструкцию, инструкцию, нужно дважды нажать Enter. После этого консоль возвращается к изначальному состоянию.

```
>>>
>>> n = 100
>>> if n > 10:
...     print("n больше чем 10")
...
n больше чем 10
>>>
```

Все эти программы заканчиваются внезапно, ничего не показывая, если условие не истинно. Но в большинстве случаев пользователю нужно показать хотя бы что-нибудь. Для этого используется оператор-выражение if-else.

■ if - else

if-else исполняет одну порцию инструкций, если условие истинно и другую — если нет. Таким образом, оператор предлагает два направления действий. Синтаксис оператора if-else следующий:

```
if  condition:
    # блок if
    statement 1
    statement 2
    and so on
else:
    # блок else
    statement 3
    statement 4
    and so on:
```

Описание работы:

При выполнении оператора if-else, условие проверяется, и если оно возвращает True, когда инструкции в блоке if исполняются. Если возвращается False, исполняются инструкции из блока else.

Пример 1: программа для расчета площади и длины окружности круга.

```
radius = int(input("Введите радиус: "))

if radius >= 0:
    print("Длина окружности = ",  2  *  3.14  *  radius)
    print("Площадь = ", 3.14 * radius ** 2)
else:
    print("Требуется ввести положительное число")
```

Первый вывод:

Введите радиус: 4

Длина окружности = 25.12

Площадь = 50.24

Второй вывод:

Введите радиус: -12

Требуется ввести положительное число

Теперь программа показывает корректный ответ пользователю, даже если условие if не является истинным.

В инструкциях if-else нужно следить за тем, чтобы условия if и else находились на одном уровне. В противном случае программа вернет синтаксическую ошибку. Например:

```
radius = int(input("Введите радиус: "))

if radius >= 0:
    print("Длина окружности = ",  2  *  3.14  *  radius)
    print("Площадь = ", 3.14 * radius ** 2)

    else:
        print("Введите положительное число")
```

При запуске этой программы, появляется сообщение об ошибке:

```
$ python3 if_and_else_not_aligned.py
  File "if_and_else_not_aligned.py", line 6
    else:
       ^
SyntaxError: invalid syntax
$
```

```
Для исправления проблемы нужно вертикально выровнять if и else
```

Другой пример:

Пример 2: код для проверки пароля, введенного пользователем.

```
password = input("Введите пароль: ")
if password == "sshh":
    print("Добро пожаловать")
else:
    print("Доступ запрещен")
```

Первый вывод:

Введите пароль: sshh

Добро пожаловать

Второй вывод:

Введите пароль: abc

Доступ запрещен

■ Оператор if внутри другого if-оператора

Операторы if-else можно использовать внутри других инструкций if или if-else (вложенные операторы). Пример 1:

```
#программа, проверяющая, имеется ли право на кредит.
```

```
def def_score(gre_score, per_grad):
    if per_grad > 70:
    # внешний блок if
        if gre_score > 150:
            # внутренний блок if
            print("вам выдан кредит")
    else:
        print("вы не имеете права на кредит")


if __name__ == '__main__':
    gre_score = int(input("Введите текущий лимит: "))
    per_grad = int(input("Введите кредитный рейтинг: "))
    def_score(gre_score, per_grad)
```

Здесь оператор if используется внутри другого if-оператора. Внутренним называют вложенный оператором if. В этом случае внутренний оператор if относится к внешнему блоку if, а у внутреннего блока if есть только одна инструкция, которая выводит 'вам выдан кредит'.

Описание работы:

Сначала оценивается внешнее условие if, то есть per_grad > 70.

Если оно возвращает True, тогда управление программой происходит внутри внешнего блока if.

Там же проверяется условие gre_score > 150.

Если оно возвращает True, тогда в консоль выводится "you are eligible for loan".

Если False, тогда программа выходит из инструкции if-else, чтобы исполнить следующие операции. Ничего при этом не выводится в консоль.

При этом если внешнее условие возвращает False, тогда выполнение инструкций внутри блока if пропускается, и контроль переходит к блоку else (9 строчка).

Первый вывод:

Введите текущий лимит: 160

Введите кредитный рейтинг: 75

вам выдан кредит

Второй вывод:

Введите текущий лимит: 160

Введите кредитный рейтинг: 60

вы не имеете права на кредит

У этой программы есть одна проблема при её повторном запуске и вводе gre_score меньше чем 150, а per_grade — больше 70:

Вывод:

Введите текущий лимит: 140

Введите кредитный рейтинг: 80

Программа ничего не выводит. Причина в том, что у вложенного оператора if нет условия else. Оно добавляется в следующем примере.

Пример 2: инструкция if-else внутри другого оператора if.

```
#программа, проверяющая, имеется ли право на кредит.

def def_score(gre_score, per_grad):
    if per_grad > 70:
    # внешний блок if
        if gre_score > 150:
            # внутренний блок if
              print("вам выдан кредит")
        else
              print("у вас низкий кредитный лимит")
    else:
        print("вы не имеете права на кредит")


if __name__ == '__main__':
    gre_score = int(input("Введите текущий лимит: "))
    per_grad = int(input("Введите кредитный рейтинг: "))
    def_score(gre_score, per_grad)
```

Вывод:

Введите текущий лимит: 140

Введите кредитный рейтинг: 80

У вас низкий кредитный лимит

Описание работы:

Эта программа работает та же, как и предыдущая. Единственное отличие — у вложенного оператора if теперь есть инструкция else. Теперь если ввести балл GRE меньше, чем 150, программа выведет: 'У вас низкий кредитный лимит'

При создании вложенных операторов if или if-else, всегда важно помнить об отступах. В противном случае выйдет синтаксическая ошибка.

Оператор if-else внутри условия else

Пример 3: программа для определения оценки клиента на основе введенных баллов.

```
def score_maker(score):
    if score >= 90:
        print("Отлично! Ваша оценка A")
    else:
        if score >= 80:
              print("Здорово! Ваша оценка - B")
        else:
              if score >= 70:
                  print("Хорошо! Ваша оценка - C")
              else:
                  if score >= 60:
                        print("Ваша оценка - D. Стоит повторить материал.")
                  else:
                        print("Вы не сдали экзамен")


if __name__ == '__main__':
    score = int(input("Введите вашу оценку: "))
    score_maker(score)
```

Первый вывод:

Введите вашу оценку: 92

Отлично! Ваша оценка A

Второй вывод:

Введите вашу оценку: 72

Хорошо! Ваша оценка - C

Третий вывод:

Введите вашу оценку: 56

Вы не сдали экзамен

Описание работы:

Когда управление программой переходит к оператору if-else, проверяется условие на строке 3 (score >= 90). Если оно возвращает True, в консоль выводится 'Отлично! Ваша оценка A'. Если значение неверное, управление переходит к условию else на 5 строке. Теперь проверяется

условие score >= 80 (6 строка). Если оно верное, тогда в консоли выводится 'Здорово! Ваша оценка — B'.

В противном случае управление программой переходит к условию else на 8 строке. И здесь снова имеется вложенный оператор if-else. Проверяется условие (score >= 70). Если оно истинно, тогда в консоль выводится “Хорошо! Ваша оценка — C”.

В противном случае управление переходит к блоку else на 11 строке. В конце концов, проверяется условие (score >= 60). Если оно возвращает True, тогда в консоль выводится “Ваша оценка — D. Стоит повторить материал.”

Если же False, тогда в консоли будет “Вы не сдали экзамен”. На этом этапе управление переходит к следующим инструкциям, написанным после внешнего if-else.

Хотя вложенные операторы if-else позволяют проверять несколько условий, их довольно сложно читать и писать. Эти же программы можно сделать более читабельными и простыми с помощью if-elif-else.

■ Оператор if-elif-else

Оператор if-elif-else — это альтернативное представление оператора if-else, которое позволяет проверять несколько условий, вместо того чтобы писать вложенные if-else. Синтаксис этого оператора следующий:

```
if condition_1:
    # блок if
    statement
    statement
    more statement
elif condition_2:
    # первый блок elif
    statement
    statement
    more statement
elif condition_3:
    # второй блок elif
    statement
    statement
    more statement

...

else
    statement
    statement
    more statement
```

Замечание: многоточие '...' в определении оператора означает, что в данном операторе можно написать сколько угодно условий eilf.

Описание работы оператора:

Когда исполняется инструкция if-elif-else, в первую очередь проверяется condition_1. Если условие истинно, тогда исполняется блок инструкций if. Следующие условия и инструкции пропускаются, и управление переходит к операторам вне if-elif-else.

Если condition_1 оказывается ложным, управление переходит к следующему условию elif, и проверяется condition_2. Если оно истинно, тогда исполняются инструкции внутри первого блока elif. Последующие инструкции внутри этого блока пропускаются.

Этот процесс повторяется, пока не находится условие elif, которое оказывается истинным. Если такого нет, тогда исполняется блок else в самом конце.

Программу для определения оценки можно переписать с помощью оператора if-elif-else.

```
def score_maker(score):

    if score >= 90:
        print("Отлично! Оценка A")
    elif score >= 80:
        print("Здорово! Оценка - B")
    elif score >= 70:
        print("Хорошо! Оценка - C")
    elif score >= 60:
        print("Оценка - D. Стоит повторить материал.")
    else:
        print("экзамен не сдан")


if __name__ == '__main__':
    score = int(input("Введите вашу оценку: "))
    score_maker(score)
```

Первый вывод:

Введите оценку: 78

Хорошо! Оценка - C

Второй вывод:

Введите оценку: 91

Отлично! Оценка A

Третий вывод:

Введите оценку: 55

экзамен не сдан

Такая программа легче читается, чем программа с вложенными if-else.

■ Итерации

Далее рассматривается циклический процесс и его составляющие элементы.

Итерация (Iteration) — это одно из повторений цикла (один шаг или один "виток" циклического процесса). Например, цикл из 3-х повторений можно представить как 3 итерации итерируемого объекта.

Итерируемый объект (Iterable) — объект, который обеспечивает повторение. Этот объект за каждую итерацию возвращает по одному результату..

Итератор (iterator) — итерируемый объект. В рамках итератора реализован метод __next__, который позволяет получать следующий элемент итерации.

Для выполнения итерации интерпретатор python выполняет следующие действия:

- Вызывает у итерируемого объекта метод iter(), тем самым получая итератор.
- Вызывает метод next(), чтобы получить каждый элемент от итератора.
- Когда метод next возвращает исключение StopIteration, цикл останавливается.

■ *Схема работы итератора в python*

итерируемый объект (iterable)

a = [1, 2, 3]

iter()

итератор (iterator)

next()	next()	next()	next()
1	2	3	StopIteration

■ Операторы цикла

Цикл — это управляющая конструкция, которая раз за разом выполняет тело цикла представленное серией команд до тех пор, пока является истинным условие для выполнения тела. Применение циклов — это возможность многократного исполнения определенного участка кода. Циклы в python представлены двумя основными конструкциями: while и for.

Цикл while считается универсальным, а цикл for нужен для обхода последовательности поэлементно.

Оба оператора одинаково применимы и являются важнейшими элементами языка python.

while..condition..block

for..in..iter_object

■ *while*

Пока условие истинно, оператор while позволяет многократно выполнять блок операторов. Оператор while является одним из вариантов оператора цикла. В инструкции while есть необязательное условие else. Но блок else на самом деле избыточен, потому что размещение его после блока while может служить той же цели.

```
count = 1 # начальное значение управляющей переменной
while count <= 10: # конечное значение управляющей переменной
    print(count, end=' ')
    count += 1
```

```
# после 9-й итерации в count окажется значение 10
# это удовлетворяет условию count <= 10,
# поэтому на 10-й итерации выводится число 10
# (значение счетчика печатается до его увеличения)
# после count станет равным 11, а, значит,
# следующая итерация не состоится, и цикл будет прерван
# в итоге получаем:
> 1 2 3 4 5 6 7 8 9 10
```

```
count = 0
    a = True
    while a:
        if count > 10:
            print('count {}'.format(count))
            count += 1
        else:
            a = False
            print('end the while loop')
```

Ещё примеры

В Python возможны составные условия. Они могут быть сколь угодно длинными, а в их записи используются логические операторы (not, and, or):

```
dayoff = False
sunrise = 6
sunset = 18

worktime = 12

# пример составного условия
while not dayoff and sunrise <= worktime <= sunset:
    if sunset == worktime:
        print("Finally it's over!")
    else:
        print('You have ', sunset - worktime, ' hours to work')
    worktime += 1

>
You have  6  hours to work
You have  5  hours to work
You have  4  hours to work
You have  3  hours to work
You have  2  hours to work
You have  1  hours to work
Finally it's over!
```

Управляющая переменная вовсе не обязана являться счётчиком. Она может быть просто логической переменной, чье значение изменяется где-то в самом цикле:

```
num = 0
control = True
while num < 10:
    num += 1

# аналогичная запись
```

```
num = 0
control = True
while control:
    if num == 10:
        control = False
    num += 1
```

Использование неинициализированной переменной в качестве управляющей цикла приводит к возникновению ошибки:

```
# unknown до этого нигде не была объявлена
while unknown:
    print('+')

>
Traceback (most recent call last):
  while unknown:
NameError: name 'unknown' is not defined
```

Ещё примеры оператора while

```
x = 20
y = 30
while x < y:
    print(x, end=' ')
    x = x + 3

word = "hello python !"
while word:
    print(word, end=" ")
    # на каждой итерации убирется символ с конца
    word = word[:-1]
```

Идея циклов while: — если требуется определенное количество раз сделать что-то, следует завести счётчик и изменять его значение (уменьшение/увеличение) в теле цикла.

```
x = 20
y = 30
while x < y:
    print(x, end=' ')
    x = x + 3

> 20 23 26 29
```

Счётчиком может быть даже строка:

```
word = "pythonstring"
while word:
    print(word, end="\n")
    # на каждой итерации убирается символ с конца
    word = word[:-1]

>
pythonstring
pythonstrin
pythonstri
pythonstr
```

```
pythonst
pythons
python
pytho
pyth
pyt
py
p
```

■ *break и continue*

Оператор break заставляет интерпретатор прервать выполнение цикла и перейти к следующей за ним инструкции:

```
counter = 0
while True:
    if counter == 10:
        break
    counter += 1
```

Цикл прервётся после того, как значение счетчика дойдёт до десяти.

Оператор continue не прекращает выполнение всего цикла, а прерывает лишь текущую итерацию, и затем переходит в начало цикла:

```
# пример вывода одних лишь чётных значений
z = 10
while z:
    z -= 1
    if z % 2 != 0:
        continue
    print(z, end=" ")

> 8 6 4 2 0
```

Эти операторы бывают удобны, однако плохой практикой считается написание кода, который чересчур ими перегружен.

■ *else*

В Python-циклах часть else выполняется лишь тогда, когда цикл отработал, не будучи прерван break-ом.

В реальной практике, else в циклах применяется нечасто. Такая конструкция отлично сработает, когда будет необходимо проверить факт выполнения всех итераций цикла.

Проверка доступности всех выбранных узлов сети:

Например, обойти все узлы локальной сети

```
def print_prime_list(list_of_numbers: list) -> None:
    """ функция выведет список чисел,
    если каждое из этих чисел является простым """
    number_count = len(list_of_numbers)  # количество чисел

    i = 0
    while i < number_count:
```

```
        x = list_of_numbers[i] // 2
        if x != 0 and list_of_numbers[i] % x == 0:
            break
        i += 1
    else:
        print(f'{list_of_numbers} - list is prime!')

print_prime_list([11, 100, 199])  #  100 - не простое число
>

print_prime_list([11, 113, 199])
> [11, 113, 199]
```

В каком-либо другом языке можно было бы завести булеву переменную, в которой хранится результат проверки, но в python существует альтернатива.

■ *while true или бесконечный цикл*

В большинстве случаев, бесконечные циклы появляются из-за логических ошибок программиста (например, когда условие цикла while при любых вариантах равно True). Поэтому следует внимательно следить за условием, при котором цикл будет завершаться.

Однако в некоторых случаях бесконечный цикл делается намерено:

Если нужно производить какие-то действия с интервалом, и выходить из цикла лишь в том случае, когда внутри тела "зашито" условие выхода.

Пример: функция, которая возвращает connection базы данных. Если связь с базой данных отсутствует, соединение будет пытаться (в цикле) установиться до тех пор, пока не установится.

Если пишется полноценный демон, который продолжительное время висит как процесс в системе и периодически производит какие-то действия, то в таком случае остановкой цикла будет прерывание работы программы.

Пример: скрипт, который раз в 10 минут "пингует" IP адреса и пишет в лог отчет о доступности этих адресов.

Важно! В бесконечных циклах рекомендуется ставить таймаут выполнения после каждой итерации:

```
import time

while True:
    print("Бесконечный цикл")
    time.sleep(1)

>
Бесконечный цикл
Бесконечный цикл
Бесконечный цикл
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    time.sleep(1)
KeyboardInterrupt

Aborted!
```

Код был прерван комбинацией клавиш ^Ctrl + C. Иначе цикл продолжался бы бесконечно.

■ *Цикл while в одну строку*

Для составных конструкций (таких, где нужен блок с отступом), можно этот отступ убрать, но только если в блоке используются простые операторы. Отделяются они всё также двоеточием.

Например, записи:

```
while x < y:
    x +=1
```

и

```
while x < y: x += 1
```

будут считаться эквивалентными, и при чтении второй из них интерпретатор не будет выдавать ошибку.

■ *Вложенные while циклы*

Вложенные while циклы встречаются реже циклов for, но они также успешно применяются. Простой пример — вывод на экран значения таблицы умножения:

```
q = 1
while q <= 9:
    w = 1
    while w <= 9:
        print(q * w, end=" ")
        w += 1
    q += 1
    print("")

>
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

Правило применения вложенных циклов while: вложения свыше третьего уровня становятся практически нечитаемыми.

■ Выход из двух циклов с помощью break

В случае вложенных циклов, оператор break завершает работу только того цикла, внутри которого он был вызван:

```
i = 100
j = 200
while i < 105:
    while j < 205:
        if j == 203:
            break
```

```
        print('J', j)
        j += 1
    print('I', i)
    i += 1
```

```
>
J 200
J 201
J 202
# здесь видно, что внутренний цикл прерывается, но внешний продолжает работу
I 100
I 101
I 102
I 103
I 104
```

■ *Цикл while в одну строку*

Для составных конструкций (таких, где нужен блок с отступом), можно этот отступ убрать, но только если в блоке используются простые операторы. Отделяются они всё также двоеточием.

Например, записи:

```
while x < y:
    x +=1

# и

while x < y: x += 1
```

будут считаться эквивалентными, и при чтении второй из них интерпретатор не будет выдавать ошибку.

■ *Выход из двух циклов с помощью break*

В случае вложенных циклов, оператор break завершает работу только того цикла, внутри которого он был вызван:

```
i = 100
j = 200
while i < 105:
    while j < 205:
        if j == 203:
            break
        print('J', j)
        j += 1
    print('I', i)
    i += 1
```

■ *for*

for..in..iter_object - оператор цикла для перебора элементов последовательности объектов (iter_object), универсальная управляющая конструкция.

```
for i in range(1, 10000000000):
    print(i)
```

В основе цикла for лежат последовательности, и в примере выше это последовательность чисел от 1 до 9999999999. for поэлементно её перебирает и выполняет код, который записан в теле цикла. В частности, для решения данной задачи туда была помещена инструкция, позволяющая выводить значение элемента последовательности на экран.

■ *Работа с числовыми последовательностями: range и enumerate*

for работает с последовательностями и позволяет повторять какую-то операцию фиксированное количество раз. Понятию "количество чего-то", всегда соответствует последовательность числовая. Для выполнения какой-либо инструкции строго определенное число раз, используется функция range().

Функцию range() можно представлять, как функцию, которая возвращает последовательность чисел, регулируемую количеством переданных в неё аргументов. Их может быть 1, 2 или 3:

range(stop);

range(start, stop);

range(start, stop, step).

Здесь start — это первый элемент последовательности (включительно), stop — последний (не включительно), а step — разность между следующим и предыдущим членами последовательности.

```
for i in range(5):
    print("Hello World!")
```

```
# 0 - начальный элемент по умолчанию
for a in range(3):
    print(a)
>
0
1
2
```

```
# два аргумента
for b in range(7, 10):
    print(b)
```

```
>
7
8
9
```

```
# три аргумента
for c in range(0, 13, 3):
    print(c)
```

```
>
```

```
0
3
6
9
12
```

Для работы с числовыми последовательностями также применяется функция enumerate(), которая определена на множестве итерируемых объектов и служит для создания кортежей на основании каждого из элементов итерируемого объекта.

Элемент кортежа представляет собой пару (индекс элемента, элемент), и это бывает удобно, когда помимо самих элементов требуется ещё и их индекс.

```
# замена каждого пятого символа в предложении, начиная с 0-го, на *
text = "Это не те признаки, которые могли бы служить для подсказки"
new_text = ""
for char in enumerate(text):
    if char[0] % 5 == 0:
        new_text += '*'
    else:
        new_text += char[1]
print(new_text)
```

■ *Работа с числовыми последовательностями: break и continue*

Ещё два оператора, которые обеспечивают управление циклами.

break — используется для завершения цикла: прерывает цикл и выходит из него;

continue — прерывает текущую итерацию и переходит к следующей.

```
# break
for num in range(0, 50):
    if num == 25:
        break
    print(num)
```

Оператор цикла, дойдя до числа 25 и вернув в условном выражении истину, прерывается и заканчивает свою работу.

Оператор continue используется для указания интерпретатору python пропустить оставшиеся операторы в текущем блоке цикла, а затем перейти к следующему циклу.

```
# continue
for num in range(40, 51):
    if num == 45:
        continue
    print(num)
>
40
41
42
43
44
```

```
46
47
48
49
50
```

В случае continue происходит похожая ситуация, только прерывается лишь одна итерация, а сам же цикл продолжается.

■ *Работа с числовыми последовательностями: else*

Если два предыдущих оператора можно часто встречать за пределами python, то else, как составная часть цикла, напрямую связана с оператором break и выполняется лишь тогда, когда выход из цикла был произведен НЕ через оператор break.

```
group_of_students = [21, 18, 19, 21, 18]
for age in group_of_students:
    if age < 18:
        break
else:
    print('Всё в порядке, проход разрешён')

> Всё в порядке, проход разрешён
```

Ещё пример

```
def print_prime_list(list_of_numbers: list) -> None:
    """ функция выведет список чисел,
    если каждое из этих чисел является простым """
    number_count = len(list_of_numbers)  # количество чисел

    i = 0
    while i < number_count:
        x = list_of_numbers[i] // 2
        if x != 0 and list_of_numbers[i] % x == 0:
            break
        i += 1
    else:
        print(f'{list_of_numbers} - list is prime!')

print_prime_list([11, 100, 199])  #  100 - не простое число

>
print_prime_list([11, 113, 199])
>
[11, 113, 199]
```

■ Пример создания итерируемого объекта

Для создания класса итерируемого объекта внутри него реализуются два метода:

__iter__() и __next__():

внутри метода __next__ () описывается процедура возврата следующего доступного элемента;

метод __iter__() возвращает сам объект, что даёт возможность использовать его, в циклах с поэлементным перебором.

Пример строкового итератора, который на каждой итерации, при получении следующего элемента (т.е. символа), приводит его к верхнему регистру.

Текст комментариев с пояснением происходящего в данном примере заключён в тройные кавычки:

```
class ToUpperCase:
    def __init__(self, string_obj, position=0):
        """сохраняется строка, полученная из конструктора,
        в поле string_obj и задаётся начальный индекс"""
        self.string_obj = string_obj
        self.position = position

    def __iter__(self):
        """ возвращается сам объект """
        return self

    def __next__(self):
        """ метод возвращает следующий элемент,
        но уже приведенный к верхнему регистру """
        if self.position >= len(self.string_obj):
        """ исключение StopIteration() сообщает
            циклу for о завершении """
            raise StopIteration()
        position = self.position
        """ инкрементируется индекс """
        self.position += 1
        """ возвращается символ в uppercase-е """
        return self.string_obj[position].upper()

low_python = "python"
high_python = ToUpperCase(low_python)  # получение итерируемого объекта
for ch in high_python:                 # применение итерируемого объекта
    print(ch, end="")                  # демонстрация результата итерации
```

■ Работа с циклами: примеры

Далее описываются некоторые приёмы программирования циклов.

■ *Цикл по списку*

Перебор list в цикле простой, поскольку список — объект итерируемый:

```
# есть список
entities_of_warp = ["Tzeench", "Slaanesh", "Khorne", "Nurgle"]
# просто берётся список, "загружается" в цикл и делается обход
for entity in entities_of_warp:
    print(entity)

>
Tzeench
Slaanesh
Khorne
Nurgle
```

Элементами списков могут быть другие итерируемые объекты. Это предполагает работу с вложенными циклами. Цикл внутри цикла достаточно просто организуется и количество уровней

вложенности не имеет пределов. Однако циклы выше второго уровня вложенности очень тяжело воспринимаются и читаются.

```
strange_phonebook = [
    ["Alex", "Andrew", "Aya", "Azazel"],
    ["Barry", "Bill", "Brave", "Byanka"],
    ["Casey", "Chad", "Claire", "Cuddy"],
    ["Dana", "Ditrich", "Dmitry", "Donovan"]
]
""" это список списков, где каждый подсписок состоит из строк
    поэтому в случае необходимости можно применить тройной for
    для посимвольного чтения всех имён и вывода их в одну строку """

for letter in strange_phonebook:
    for name in letter:
        for character in name:
            print(character, end='')

> A l e x A n d r e w A y a A z a z e l B a r ...
```

■ *Цикл по словарю*

Данный пример связан с итерированием словарей. При переборе словаря обычно нужны ключ и значение. Для этого существует метод .items(), который создает представление в виде кортежа для каждого словарного элемента. В этом случае цикл будет выглядеть следующим образом:

```
#создаётся словарь
top_10_largest_lakes = {
    "Caspian Sea": "Saline",
    "Superior": "Freshwater",
    "Victoria": "Freshwater",
    "Huron": "Freshwater",
}

# обход словаря в цикле for и подсчёт количества озер
# с солёной водой и количества озёр с пресной
salt = 0
fresh = 0
# в данном случае, пара "lake, water",
# есть распакованный кортеж,
# где lake - ключ словаря, а water - значение.
# цикл обходит не сам словарь,
# а его представление в виде пар кортежей
for lake, water in top_10_largest_lakes.items():
#                       применение метода items
    if water == 'Freshwater':
        fresh += 1
    else:
        salt += 1
print("Amount of saline lakes in top10: ", salt)
print("Amount of freshwater lakes in top10: ", fresh)

> Amount of saline lakes in top10:  1
> Amount of freshwater lakes in top10:  3
```

■ *Цикл по строке*

Строки — это простые последовательности, состоящие из символов. Поэтому обходить их в цикле достаточно просто.

```
word = 'Alabama'
for w in word:
    print(w, end=" ")

> A l a b a m a
```

■ *Цикл for с шагом*

Цикл for с шагом создается при помощи функции range, куда нужно передать размер шага, в качестве третьего по счету аргумента:

```
# вывод чисел от 100 до 1000 с шагом 150
for nums in range(100, 1000, 150):
    print(nums)

>
100
250
400
550
700
850
```

■ *Обратный цикл for*

Генератор последовательностей range() может создать убывающую последовательность чисел, которая может быть применена в качестве массива индексов при обходе последовательности с помощью for в обратном направлении (обратный цикл for).

```
# числа от 40 до 50 в убывающей последовательности
# задаётся обратный порядок с помощью шага step размером -1
for nums in range(50, 39, -1):
    print(nums)

>
50
49
48
47
46
45
44
43
42
41
40
```

■ *for в одну строку*

Генератор python (list comprehensions) позволяет задавать однострочные последовательности. Может быть их запись, сложнее для понимания, зато очевидно короче. В общем виде такой генератор задаётся следующим образом:

```
[результирующее выражение | цикл | опциональное условие]
```

Пример, в котором продублируется каждый символ строки inputString:

```
# здесь letter * 2 — результирующее выражение;
# for letter in inputString — цикл, а необязательное условие опущено
double_letter = [letter * 2 for letter in "Banana"]
# результирующее выражение  цикл
print(double_letter)    # распечатка результата

> ['BB', 'aa', 'nn', 'aa', 'nn', 'aa']
```

Второй пример (цикл с условием):

```
# список, который будет состоять из четных чисел от 0 до 30
# здесь if x % 2 == 0 — необязательное условие
even_nums = [x for x in range(30) if x % 2 == 0]

print(even_nums)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

В Python не существует конструкций, которая прерывала бы сразу несколько циклов. Но есть как минимум 3 способа, которыми можно реализовать данное поведение:

- ***Выход из циклов с помощью for … else …***

Для выхода используется конструкция for … else …

```
def same_values_exists(list_1: list, list_2: list) -> None:
# в качестве аргументов — два списка list
    """ функция выводит на экран
    первые совпавшие числа из списков """
    for i in list_1:
        for j in list_2:
            print("compare: ", i, j)
            if i == j:
                print(f"found {i}")
                break
        else:
            continue
        break

if __name__ == '__main__':
    same_values_exists([0, 10, -2, 23], [-2, 2])
# вызов функции same_values_exists с двумя списками list
# в качестве аргументов

compare:  0 -2
compare:  0 2
compare:  10 -2
compare:  10 2
compare:  -2 -2
```

```
found -2
```

Если все итерации вложенного цикла сработают, выполнится else, который скажет внешнему циклу продолжить выполнение. Если во внутреннем цикле сработает break, сразу выполнится второй break.

■ *Выход из циклов через создание дополнительного флага*

В следующем примере дополнительный флаг - это break_the_loop.

```
def same_values_exists(list_1: list, list_2: list) -> None:
    """ функция выводит на экран
    первые совпавшие числа из списков """
    break_the_loop = False

    for i in list_1:
        for j in list_2:
            print("compare: ", i, j)
            if i == j:
                print(f"found {i}")
                break_the_loop = True
                break
        if break_the_loop:
            break

if __name__ == '__main__':
    same_values_exists([0, 10, -2, 23], [-2, 2])
```

```
compare:  0 -2
compare:  0 2
compare:  10 -2
compare:  10 2
compare:  -2 -2
found -2
```

Внешний цикл был прерван вслед за внутренним!

■ *Выход из циклов через return*

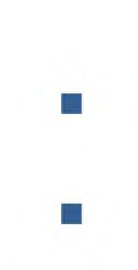
Если циклы находятся в функции (как в данном примере), достаточно просто выполнить return:

```
def same_values_exists(list_1: list, list_2: list) -> None:
    """ функция выводит на экран
    первые совпавшие числа из списков """
    for i in list_1:
        for j in list_2:
            print("compare: ", i, j)
            if i == j:
                print(f"found {i}")
                return

if __name__ == '__main__':
    same_values_exists([0, 10, -2, 23], [-2, 2])
```

```
compare:  0 -2
compare:  0 2
compare:  10 -2
compare:  10 2
```

```
compare:  -2 -2
found -2
```



■ Массивы

Массив - это структура данных, в которой хранятся значения одного типа. Массивы в Python могут содержать только значения, соответствующие одному и тому же типу данных.

Массив не является основным типом данных, как строки, целое число и т. д. из числа ранее перечисленных типов. Для применения массивов в языке Python, прежде всего нужно импортировать стандартный array модуль.

Импорт модуля array в Python

```
from array import *
```

■ Объявление массива

Массив есть специально объявляемая последовательность, позволяющая компактно хранить объекты одного из базовых (ранее объявленных) типов. Массив объявляется после импорта модуля array.
Для объявления массива с помощью конструктора array(...) в качестве аргументов требуются следующие элементы:
datatype (typecode) — информация о типе данных в выражении вызова конструктора. Также применяются typecode - это коды, которые также используются для определения типа значений массива или типа самого массива. Коды типов используются для краткого определения массивов в Python. Существует таблица, которая показывает возможные значения, которые можно использовать при объявлении массива, и его тип.
Initializers - представляется значением (множеством значений), с которыми массив объявляемый массив инициализируется.

```
from array import *

firstArray = array(typecode, [Initializers])
# firstArray имя массива, typecode позволяет определять тип массива,
# Initializers представляется значением, которыми массив
# инициализируется.

# typecode - это коды, которые используются для определения типа
# значений массива или типа массива. Таблица в разделе параметров
# показывает возможные значения, которые можно использовать
# при объявлении массива, и его тип.
secondArray = array('i', [0,1,2,3,4,5])
# В примере используется TypeCode i.
# Этот тип представляет целое число со знаком, размер которого
# составляет 2 байта.
```

В качестве initializer может быть указан список, либо строка. В этом случае они будут переданы методам .fromlist() и .fromstring() / .fromunicode() соотвественнно для добавления в массив начальных элементов.
Также существует функция typecode(), которая используется для возврата символа typecode, использованного для создания массива. В приведенной ниже таблице есть некоторые из них. Python.typecodetypecodes:

Array Type codes: представление таблицы

Type Code	Python type	C Type	Minimum size in bytes
'b'	int	Signed char	1
'B'	int	Unsigned char	1
'u'	Unicode character	wchar_t	2
'h'	Int	Signed short	2
'H'	int	Unsigned short	2
'i'	int	Signed int	2
'I'	int	Unsigned int	3
'l'	int	signed long	4
'L'	int	Unsigned long	4
'q'	int	Signed long long	8
'Q'	int	Unsigned long long	8
'f'	float	float	4
'd'	float	double	8

The array module defines a property called .typecodes that returns a string containing all supported type codes found in Table 1. While the array method defines the typecode property which returns the type code character used to create the array.

■ Библиотека numpy: объявления массивов

Массив может быть создан из обычного списка или кортежа Python с использованием функции array(). Причем тип полученного массива зависит от типа элементов последовательности.
Также может быть применена библиотека языка Python NumPy, добавляющая поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.
Основным объектом NumPy является однородный многомерный массив (в numpy называется numpy.ndarray). Это многомерный массив элементов (обычно чисел) одного типа.
Наиболее важные атрибуты объектов ndarray:

- ndarray.ndim - число измерений (чаще их называют "оси") массива.
- ndarray.shape - размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси. Для матрицы из n строк и m столбов, shape будет (n,m). Число элементов кортежа shape равно ndim.
- ndarray.size - количество элементов массива. Равно произведению всех элементов атрибута shape.
- ndarray.dtype - объект, описывающий тип элементов массива. Можно определить dtype, используя стандартные типы данных Python. NumPy здесь предоставляет целый букет возможностей, как встроенных, например: bool_, character, int8, int16, int32, int64, float8, float16, float32, float64, complex64, object_, так и возможность определить собственные типы данных, в том числе и составные.
- ndarray.itemsize - размер каждого элемента массива в байтах.
- ndarray.data - буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

В NumPy существует много способов создать массив. Один из наиболее простых - создать массив из обычных списков или кортежей Python, используя функцию numpy.array() (array - функция, создающая объект типа ndarray):

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>>
```

```
>>> a.dtype
dtype('int32')
>>> a = np.array([1.1, 2.2, 3.3])
>>> a
array([ 1.1,  2.2,  3.3])
>>>
>>> a.dtype
dtype('float64')
>>> a = np.array([1 + 2j, 2 + 3j])
>>> a.dtype
dtype('complex128')
>>>
```

Или без приглашения ('>>>') и вывода

```
import numpy as np
a = np.array([1, 2, 3])
a
type(a)
```

При объявлении массива важно, что аргументом функции array() должна быть именно последовательность, а не несколько аргументов.
Пример некорректного объявления массива:

```
>>> a = np.array(1, 2, 3)      #  Неправильно!!!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: only 2 non-keyword arguments accepted
>>>
>>> a = np.array([1, 2, 3])    #  Правильное объявление
>>> a
array([1, 2, 3])
>>>
>>> a = np.array((1, 2, 3))    #  И так тоже правильно
>>> a
array([1, 2, 3])
>>>
```

без приглашения ('>>>') и вывода

```
import numpy as np
a = np.array(1, 2, 3)      #  Неправильно!!!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: only 2 non-keyword arguments accepted

a = np.array([1, 2, 3])    #  Правильное объявление
a
array([1, 2, 3])

a = np.array((1, 2, 3))    #  И так тоже правильно
a
array([1, 2, 3])
```

Уровень вложенности исходной последовательности в выражении создания массива определяет размерность получаемого массива.

Функция array() преобразует последовательности последовательностей в двумерные массивы, а последовательности последовательностей, которые тоже состоят из последовательностей в трехмерные массивы.

```
>>> a = np.array([[2, 4], [6, 8], [10, 12]])
>>> a
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
>>>
>>> b = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]])
>>> b
array([[[ 1,  2],
        [ 3,  4]],

       [[ 5,  6],
        [ 7,  8]],

       [[ 9, 10],
        [11, 12]]])
>>>
>>> a.ndim    #  Количество осей массива
2
>>> b.ndim
3
```

без приглашения ('>>>') и вывода

```
a = np.array([[2, 4], [6, 8], [10, 12]])
b = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]])
a.ndim    #  Количество осей массива
b.ndim
```

Функция array() также позволяет определить тип данных массива.

```
>>> a = np.array([[2, 4], [6, 8], [10, 12]], dtype = complex )
>>> a
array([[  2.+0.j,   4.+0.j],
       [  6.+0.j,   8.+0.j],
       [ 10.+0.j,  12.+0.j]])
>>>
>>> a = np.array([[2, 4], [6, 8], [10, 12]], dtype = float )
>>> a
array([[  2.,   4.],
       [  6.,   8.],
       [ 10.,  12.]])
```

Очень часто возникает задача создания массива определенного размера, причем абсолютно неважно чем заполнен массив. В этом случае можно воспользоваться циклами или генераторами списков (кортежей), но NumPy для таких случаев предлагает более быстрые и менее затратные функции-заполнители:

- функция zeros заполняет массив нулями,
- функция ones заполняет массив единицами,
- функция empty заполняет массив случайными числами, которые зависят от состояния памяти.

По умолчанию, тип создаваемого массива - float64.

```
>>> np.zeros((3,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
>>> np.ones((3,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>>
>>> np.ones((3,3), dtype = complex)  #  Можно изменить тип массива
array([[ 1.+0.j,  1.+0.j,  1.+0.j],
       [ 1.+0.j,  1.+0.j,  1.+0.j],
       [ 1.+0.j,  1.+0.j,  1.+0.j]])
>>>
>>> np.empty([3, 3])
array([[ -2.56357799e-042,   1.00079160e-313,  -5.41541116e-070],
       [  1.51668796e-314,   0.00000000e+000,   1.48219694e-320],
       [  2.61270984e-262,   0.00000000e+000,   8.36469502e-316]])
```

Для создания последовательностей чисел NumPy предоставляет функцию arange, которая возвращает одномерные массивы:

```
>>> np.arange(10)    #  От 0 до указанного числа
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>> np.arange(10, 20)    #  Диапазон
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
>>>
>>> np.arange(20, 100, 10)    #  Диапазон с заданным шагом
array([20, 30, 40, 50, 60, 70, 80, 90])
>>>
>>> np.arange(0, 1, 0.1)    #  Аргументы могут иметь тип float
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

Если функция arange используется с аргументами типа float, то предсказать количество элементов в возвращаемом массиве не просто. Гораздо чаще возникает необходимость указания не шага изменения чисел в диапазоне, а количества чисел в заданном диапазоне.
Функция linspace, так же как и arange принимает три аргумента, но третий аргумент, как раз и указывает количество чисел в диапазоне.

```
>>> np.linspace(0, 1, 5)
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
>>>
>>> np.linspace(0, 1, 7)
array([ 0.        ,  0.16666667,  0.33333333,  0.5       ,  0.66666667,
        0.83333333,  1.        ])
>>>
>>> np.linspace(10, 100, 5)
array([  10. ,   32.5,   55. ,   77.5,  100. ])
```

Функция linspace удобна еще и тем, что может быть использована для вычисления значений функций на заданном множестве точек:

```
>>> x = np.linspace( 0, 2*np.pi, 10 )
```

```
>>> x
array([ 0.        ,  0.6981317 ,  1.3962634 ,  2.0943951 ,  2.7925268 ,
        3.4906585 ,  4.1887902 ,  4.88692191,  5.58505361,  6.28318531])
>>>
>>> y1 = np.sin(x)
>>> y1
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
         8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
        -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
        -2.44929360e-16])
>>>
>>> y2 = np.cos(x)
>>> y2
array([ 1.        ,  0.76604444,  0.17364818, -0.5       , -0.93969262,
       -0.93969262, -0.5       ,  0.17364818,  0.76604444,  1.        ])
```

Массив как класс. Перечень методов

Первым аргументом каждого из перечисляемых в этом разделе методов является параметр self – ссылка на соответствующий объект-представитель класса array. Другие аргументы методов будут рассмотрены позже.

- append(self, ...) добавить любое значение данного типа в массив. Отдельные элементы в массиве могут быть доступны через индексы. Индексация элементов массива Python начинается с нуля.
- insert(self, ...) вставить значение по любому индексу массива.
- extend(self, ...) расширение массива python (???). Массив Python может быть расширен с более чем одного значения с помощью метода extend().
- fromlist(self, ...) добавить элементы из списка в массив, используя метод
- remove(self, ...) удалить любой элемент массива, используя метод
- pop(self, ...) удалить последний элемент массива
- index(self, ...) получить любой элемент через его индекс
- reverse(self, ...) получить обратный массив в Python. Метод изменяет порядок индексации элементов массивов.
- buffer_info(self, ...) позволяет получить информацию о буфере массива. Этот метод предоставляет начальный адрес буфера массива в памяти и количество элементов в массиве.
- count(self, ...) проверка количества вхождений элемента в массиве.
- tounicode(self, ...) преобразовать массив в строку.
- tolist(self, ...) преобразовать массив в список Python с теми же элементами. Применяется, если вместо массива нужен список объектов.
- __len(self)__ количество элементов массива.

Индексация в массиве

При работе с индексами массивов используются квадратные скобки ([ ]). С помощью индексирования можно ссылаться на отдельные элементы, выделяя их или даже меняя значения. При создании нового массива шкала с индексами создается автоматически. Для получения доступа к одному элементу на него нужно сослаться через его индекс.

```
a = np.arange(10, 16)
>>> a
array([10, 11, 12, 13, 14, 15])
>>> a[4]
14
```

NumPy также принимает отрицательные значения. Такие индексы представляют собой аналогичную последовательность, где первым элемент будет представлен самым большим отрицательным значением.

```
>>> a[-1]
15
>>> a[-6]
10
```

Для выбора нескольких элементов в квадратных скобках можно передать массив индексов.

```
>>> a[[1, 3, 4]]
array([11, 13, 14])
```

Двухмерные массивы, матрицы, представлены в виде прямоугольного массива, состоящего из строк и колонок, определенных двумя осями, где ось 0 представлена строками, а ось 1 — колонками. Таким образом индексация происходит через пару значений; первое — это значение ряда, а второе — колонки. И если нужно получить доступ к определенному элементу матрицы, необходимо все еще использовать квадратные скобки, но уже с двумя значениями.

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A
array([[10, 11, 12],
[13, 14, 15],
[16, 17, 18]])
```

Если нужно удалить элемент третьей колонки во второй строке, необходимо ввести пару [1, 2].

```
>>> A[1, 2]
15
```

■ Итерация в массиве

Для перебора по элементам массива применяется конструкция
for <элемент> in <массив> :

```
>>> for i in A:
        print(i)

10
11
12
13
14
15
16
```

```
17
18
```

Здесь в случае с двухмерным массивом можно использовать вложенные циклы внутри for. Первый цикл будет сканировать строки массива, а второй — колонки. Если применить цикл for к матрице, он всегда будет перебирать в первую очередь по строкам.

```
>>> for row in A:
        print(row)

[10 11 12]
[13 14 15]
[16 17 18]
```

Если необходимо перебирать элемент за элементом, можно использовать следующую конструкцию, применив цикл for для A.flat:

```
>>> for item in A.flat:
        print(item)

10
11
12
13
14
15
16
17
18
```

NumPy предлагает и альтернативный способ итерации. Перебор элементов, как правило, используется в функциях для конкретных рядов, колонок или отдельных объектов. Можно запустить функцию агрегации, которая вернет значение для каждой колонки или даже для каждой строки, но существует более оптимальный способ, когда NumPy забирает процесс итерации на себя: функция apply_along_axis().

Эта функция принимает три аргумента:

- функцию,
- ось, для которой нужно применить перебор,
- сам массив.

Если значение аргумента ось равно 0, функция будет применена к элементам по колонкам, а если 1 — то по строкам. Например, можно посчитать среднее значение сперва по колонкам, а потом по строкам.

```
>>> np.apply_along_axis(np.mean, axis=0, arr=A)
array([ 13., 14., 15.])
>>> np.apply_along_axis(np.mean, axis=1, arr=A)
array([ 11., 14., 17.])
```

Ранее использовались функции из библиотеки NumPy, но можно определить (написать) и применить собственные.

■ Срезы в массиве

Срезы позволяют извлекать части массива, возможно, для создания новых массивов. Результирующие массивы, которые получаются в результате применения срезов для списков python являются копиями исходного массива на основе одного и того же буфера.

В зависимости от части исходного массива, которую необходимо извлечь с помощью операции индексации, нужно использовать срез - последовательность числовых значений, разделенную двоеточием (:) в квадратных скобках операции индексации.

Например, чтобы получить часть массива от второго до шестого элемента, необходимо ввести индекс первого элемента — 1 и индекса последнего — 5 (индексация массива начинается с 0), разделив их двоеточием (:) .

```
>>> a = np.arange(10, 16)
>>> a
array([10, 11, 12, 13, 14, 15])
>>> a[1:5]
array([11, 12, 13, 14])
```

Если нужно извлечь элемент из предыдущего отрезка и пропустить один или несколько элементов, можно использовать более сложный вариант индексации, который включает третье число, представляющее интервал последовательности. Например, со значением интервала последовательности равного 2, результат будет такой:

```
>>> a[1:5:2]
array([11, 13])
```

При использовании срезов возможны случаи, когда явные числовые значения не используются.
Если не ввести первое число, NumPy (интерпретатор python) неявно интерпретирует его как 0 (то есть, первый элемент массива).
Если пропустить второй — он будет заменен на максимальный индекс.
Если пропустить последний — он представляется как 1.
Таким образом, все элементы будут перебираться без интервалов.

```
>>> a[::2]
array([10, 12, 14])
>>> a[:5:2]
array([10, 12, 14])
>>> a[:5:]
array([10, 11, 12, 13, 14]
```

Срезы также работают в случае с двухмерными массивами, но их (срезы) нужно определять отдельно для строк и колонок. Например, если нужно получить только первую строку:

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

Если по второму индексу оставить только двоеточие без числа, будут выбраны все колонки.

```
>>> A[0,:]
array([10, 11, 12])
```

Если нужно выбрать все значения первой колонки, то нужно записать обратное.

```
>>> A[:,0]
array([10, 13, 16])
```

Если из исходного массива необходимо извлечь матрицу меньшего размера, то нужно явно указать все интервалы с соответствующими индексами.

```
>>> A[0:2, 0:2]
array([[10, 11],
       [13, 14]])
```

Если индексы в строках или колонках не последовательны, в выражении индексации исходного массива указывается массив индексов.

```
>>> A[[0,2], 0:2]
array([[10, 11],
       [16, 17]])
```

Также в языке определены списки. Список - это структура данных, в которой могут быть записаны значения разных типов.

Списки python могут содержать значения, соответствующие разным типам данных. При этом доступ к элементам списка также обеспечивается с помощью индексации. В Python это основное различие между массивами и списками.

Следующий пример демонстрирует некоторые приёмы работы с массивами:

- различные способы инициализации массива;
- добавление значений данного типа в массив;
- вставка значения определённого типа по данному индексу массива;
- сохранение массива в поле объекта-представителя данного класса;
- определение количества элементов в массиве;
- вывод элементов массива в зависимости от значения аргумента.

```
from array import *

class xClass:

    buff = None
    array0 = None
    array1 = None
    array2 = None
    array3 = None

    def __init__(self):
        self.array0 = array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
        self.array1 = array('f', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
        self.array2 = [0, 11, 22, 33, 44, 55, 66, 77, 88, 99]
        self.array3 = array('u',['q','w','e','r','t','y','u','i','o','p'])
```

```
    def xprint(self, name):
        match   name:
            case 'array0':
                self.buff = self.array0

            case 'array1':
                self.buff = self.array1

            case 'array2':
                self.buff = self.array2

            case 'array3':
                self.buff = self.array3
                self.buff.append('z')
                self.buff.insert(5, 'x')

                array3size = self.buff.__len__()
                print('size of array3 is ', array3size)

        for i in self.buff:
            print(i)

def print_hi(name):

    print(f'Hi, {name}')  # Press Ctrl+F8 to toggle the breakpoint.
    xxx = xClass()
    xxx.xprint(name)

if __name__ == '__main__':
    print_hi('array0')
    print_hi('array1')
    print_hi('array2')
    print_hi('array3')
```

■ Списки

Списки в Python - упорядоченные изменяемые коллекции объектов ПРОИЗВОЛЬНЫХ типов. В отличие от массивов, в список могут включаться данные различных типов.

■ Создание списков

Чтобы использовать списки, их нужно создать. Существует несколько способов создания списков. Например, можно обработать итерируемый объект (объект, для которого определена операция итерации — перебора объекта по составляющим его элементам, например, строка состоит и символов и для неё определены посимвольные операции) встроенной функцией list:

```
>>> list('список')
['с', 'п', 'и', 'с', 'о', 'к']
```

или

```
list('список')
```

Список можно создать и при помощи литерала:

```
>>>
>>> s = []  # Пустой список (массив)
>>> l = ['s', 'p', ['isok'], 2]  # символы, массив строк, целое
>>> s
[]
>>> l
['s', 'p', ['isok'], 2]
```

или

```
s = []  # Пустой список (массив)
l = ['s', 'p', ['isok'], 2]  # символы, массив строк, целое
s
l
```

Список может содержать любое количество любых объектов (в том числе и вложенные списки), или не содержать ничего.

Ещё один способ создания списка - применение генератора списков. Генератор списков строит новый список на основе итерируемого объекта, применяя выражение генератора к каждому элементу итерируемого объекта.

```
>>> c = [c * 3 for c in 'list']
>>> c
['lll', 'iii', 'sss', 'ttt']
```

или

```
c = [c * 3 for c in 'list']
c
```

Более сложные конструкции генераторов...

```
>>>
>>> c = [c * 3 for c in 'list' if c != 'i']
```

```
>>> c
['lll', 'sss', 'ttt']
>>> c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']
>>> c
['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```

или

```
c = [c * 3 for c in 'list' if c != 'i']
c  #  ['lll', 'sss', 'ttt']
c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']
c  #  ['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```

Первый итерируемый объект — строка 'list'. Генератор выбирает из строки каждый символ и добавляет его в список, предварительно утраивая этот символ, если только это не символ 'i'.

Во втором случае используются два итерируемых объекта — строки 'list' и 'spam'. Составляющие итерируемые объекты символы попарно объединяются с помощью операции '+', если только это не символы 'I' или 'a' .

Рекомендуется не усложнять код и не злоупотреблять генераторами, а применять простые операции генерации, в том числе и операторы цикла for.

■ Функции и методы списков

Списки созданы, как теперь их применыть. Для списков в python доступны основные встроенные функции, а также методы списков. Таблица методы списков:

Метод	Что делает
list.append(x)	Добавляет элемент в конец списка.
list.extend(L)	Расширяет список list, добавляя в конец все элементы списка L.
list.insert(i, x)	Вставляет на значение x по индексу i-го элемента.
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x. Возбуждается исключение ValueError, если такого элемента несуществует.
list.pop([i])	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент.
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end).
list.count(x)	Возвращает количество элементов со значением x.
list.sort([key=функция])	Сортирует список на основе функции.
list.reverse()	Разворачивает список.

list.copy()	Поверхностная копия списка.
list.clear()	Очищает список.

Методы списков, в отличие от строковых методов, изменяют сам список, а потому результат выполнения не нужно записывать в этот список как в переменную.

```
>>>
>>> l = [1, 2, 3, 5, 7]
>>> l.sort()
>>> l
[1, 2, 3, 5, 7]
>>> l = l.sort()
>>> print(l)
None
```

примеры работы со списками:

```
>>>
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

■ Универсальные функции NumPy (ufunc)

NumPy, предоставляет готовые инструменты и математические функции для работы с массивами. К ним относятся и универсальные функции (специализированные универсальные функции).

■ Специализированные универсальные функции

Универсальная функция - это оболочка над обычной функцией, благодаря которой возможно вполнять определенные действия над целыми массивами. Данные функции выполняют действия над массивами поэлементно, поддерживают механизм транслирования, автоматически преобразуют типы данных и обеспечивают доступ к более тонким настройкам своей работы.

Универсальная функция (сокращенно ufunc) — это функция, которая работает с ndarrays поэлементно, поддерживая широковещательную рассылку массивов, приведение типов и некоторые другие стандартные функции. То есть ufunc — это "векторизованная" оболочка для функции, которая принимает фиксированное количество конкретных входных данных и производит фиксированное количество конкретных выходных данных.

Большинство универсальных функций Numpy реализованы в скомпилированном C-коде и могут обрабатывать самые разные объекты. В таком случае перебор по колонкам и по рядам выдает один и тот же результат так как ufunc при работе с массивами и другими объектами выполняет перебор элемент за элементом.

Примеры работы универсальных функций:

```
>>> def foo(x):
    return x/2

>>> np.apply_along_axis(foo, axis=1, arr=A)
array([[5., 5.5, 6.],
       [6.5, 7., 7.5],
       [8., 8.5, 9.]])
>>> np.apply_along_axis(foo, axis=0, arr=A)
array([[5., 5.5, 6.],
       [6.5, 7., 7.5],
       [8., 8.5, 9.]])
```

В этом случае функция ufunc делит значение каждого элемента надвое вне зависимости от того, был ли применен перебор к строке или колонке.

```
# This is a sample Python ufunc script.
import numpy as np

def get_middle(arr_key):
    return arr_key/2

def arr_build(min_max, x_y):
    a = np.arange(min_max[0], min_max[1]).reshape((x_y[0], x_y[1]))
    print(a)
    return a

def ufunc_tst_0(arr_key):
    print('===0===')
    val = np.apply_along_axis(np.mean, axis=0, arr=arr_key)
    print(val)
    print('===1===')
```

```
    val = np.apply_along_axis(np.mean, axis=1, arr=arr_key)
    print(val)

def ufunc_tst_1(arr_key, get_middle):
    print('===0===')
    val = np.apply_along_axis(get_middle, axis=0, arr=arr_key)
    print(val)
    print('===1===')
    val = np.apply_along_axis(get_middle, axis=1, arr=arr_key)
    print(val)


if __name__ == '__main__':
    A = arr_build([10, 19], [3, 3])
    ufunc_tst_0(A)
    # применение функции get_middle в качестве аргумента
    # к функции np.apply_along_axis: значение аргумента axis
    # на результат не влияет.
    ufunc_tst_1(A, get_middle)
```

Универсальная функция np.sin может обрабатывать самые разные объекты:

```
>>> import numpy as np
>>>
>>> np.sin(60)
-0.3048106211022167
>>> np.sin([0, 30, 60])
array([ 0.        , -0.98803162, -0.30481062])
>>> np.sin((0, 30, 60))
array([ 0.        , -0.98803162, -0.30481062])
>>> np.sin(np.array([0, 30, 60]))
array([ 0.        , -0.98803162, -0.30481062])
>>> np.sin(np.matrix([0, 30, 60]))
matrix([[ 0.        , -0.98803162, -0.30481062]])
```

■ Перечень универсальных функций

Большинство математических функций NumPy являются универсальными, т.е. поддерживают множество параметров, которые позволяют оптимизировать их работу в зависимости от специфики реализуемого алгоритма.

■ *Тригонометрические функции*

`sin(x)`

Тригонометрический синус.

`cos(x)`

Тригонометрический косинус.

`tan(x)`

Тригонометрический тангенс.

`arcsin(x)`

Обратный тригонометрический синус.

`arccos(x)`

Обратный тригонометрический косинус.

`arctan(x)`

Обратный тригонометрический тангенс.

`hypot(x1, x2)`

Вычисляет длинну гипотенузы по указанным длинам катетов.

`arctan2(x1, x2)`

Обратный тригонометрический тангенс угла где x1 - противолежащий катет, x2 - прилежащий катет. В отличие от arctan (x) функция arctan2 (y, x) справедлива для всех углов и поэтому может быть использована для преобразования вектора в угол без риска деления на ноль, а также возвращает результат в правильном квадранте.

`degrees(x)`

Преобразует радианную меру угла в градусную.

`radians(x)`

Преобразует градусную меру угла в радианную.

`unwrap(p[, discont, axis])`

Корректировка фазовых углов при переходе через значение pi.

`deg2rad(x)`

Преобразует градусную меру угла в радианную.

`rad2deg(x)`

Преобразует радианную меру угла в градусную.

■ *Гиперболические функции*

`sinh(x)`

Гиперболический синус.

`cosh(x)`

Гиперболический косинус.

`tanh(x)`

Гиперболический тангенс.

`arcsinh(x)`

Обратный гиперболический синус.

`arccosh(x)`

Обратный гиперболический косинус.

`arctanh(x)`

Обратный гиперболический тангенс.

■ *Округление*

`around(a[, decimals, out])`

Равномерное (банковское) округление до указанной позиции к ближайшему четному числу.

`round_(a[, decimals, out])`

Эквивалентна around().

`rint(x)`

Округляет до ближайшего целого.

`fix(x[, out])`

Округляет до ближайшего к нулю целого числа.

`floor(x)`

Округление к меньшему ("пол").

`ceil(x)`

Округление к большему ("потолок").

`trunc(x)`

Отбрасывает дробную часть числа.

■ *Суммы, разности, произведения*

`prod(a[, axis, dtype, out, keepdims])`

Произведение элементов массива по заданной оси.

`sum(a[, axis, dtype, out, keepdims])`

Сумма элементов массива по заданной оси.

`nanprod(a[, axis, dtype, out, keepdims])`

Произведение элементов массива по заданной оси в котором элементы NaN учитываются как 1.

`nansum(a[, axis, dtype, out, keepdims])`

Сумма элементов массива по заданной оси в котором элементы NaN учитываются как 0.

`cumprod(a[, axis, dtype, out])`

Возвращает накопление произведения элементов по заданной оси, т.е. массив в котором каждый элемент является произведением предшествующих ему элементов по заданной оси в исходном массиве.

`cumsum(a[, axis, dtype, out])`

Возвращает накопление суммы элементов по заданной оси, т.е. массив в котором каждый элемент является суммой предшествующих ему элементов по заданной оси в исходном массиве.

`nancumprod(a[, axis, dtype, out])`

Возвращает накопление произведения элементов по заданной оси, т.е. массив в котором каждый элемент является произведением предшествующих ему элементов по заданной оси в исходном массиве. Элементы NaN в исходном массиве при произведении учитываются как 1.

`nancumsum(a[, axis, dtype, out])`

Возвращает накопление суммы элементов по заданной оси, т.е. массив в котором каждый элемент является суммой предшествующих ему элементов по заданной оси в исходном массиве. Элементы NaN в исходном массиве при суммировании учитываются как 0.

`diff(a[, n, axis])`

Возвращает n-ю разность вдоль указанной оси.

`ediff1d(ary[, to_end, to_begin])`

Разность между последовательными элементами массива.

`gradient(f, *varargs, **kwargs)`

Дискретный градиент (конечные разности вдоль осей) массива f.

`cross(a, b[, axisa, axisb, axisc, axis])`

Векторное произведение двух векторов.

`trapz(y[, x, dx, axis])`

Интегрирование массива вдоль указанной оси методом трапеций.

■ *Экспоненцирование и логарифмирование*

`exp(x, /[, out, where, casting, order, …])`

Экспонента всех элементов массива.

`expm1(x, /[, out, where, casting, order, …])`
Вычисляет exp(x)-1 всех элементов массива.

`exp2(x, /[, out, where, casting, order, …])`
Вычисляет 2**x для всех x входного массива.

`log(x, /[, out, where, casting, order, …])`
Натуральный логарифм элементов массива.

`log10(x, /[, out, where, casting, order, …])`
Десятичный логарифм элементов массива.

`log2(x, /[, out, where, casting, order, …])`
Логарифм элементов массива по основанию 2.

`log1p(x, /[, out, where, casting, order, …])`
Вычисляет log(x+1) для всех x входного массива.

`logaddexp(x1, x2, /[, out, where, casting, …])`
Натуральный логарифм суммы экспонент элементов входных массивов.

`logaddexp2(x1, x2, /[, out, where, casting, …])`
Двоичный логарифм от 2**x1 + 2**x2 для всех элементов входных массивов.

■ *Другие специальные функции*

`i0(x)`
Модифицированная функция Бесселя первого рода нулевого порядка.

`sinc(x)`
Вычисляет нормированный кардинальный синус элементов массива.

■ *Операции с плавающей точкой*

`signbit(x, / [, out, where, casting, order, …])`
Возвращает True для всех элементов массива у которых знаковый бит установлен в отрицательное значение.

`copysign(x1, x2, / [, out, where, casting, …])`
Изменяет знак элементов из массива x1 на знак элементов из массива x2.

`frexp(x[, out1, out2], / [, out, where, …])`
Разложение элементов массива в показатель мантиссы и двойки.

`ldexp(x1, x2, / [, out, where, casting, …])`
Вычисляет x1*2**x2.

`nextafter(x1, x2, / [, out, where, casting, …])`
Возвращает значение с плавающей точкой следующее за элементом из x1 в направлении элемента из x2.

`spacing(x, / [, out, where, casting, order, …])`
Поэлементно вычисляет расстояние между значением из массива x и ближайшим соседним числом.

■ *Арифметические операции*

`lcm(x1, x2, / [, out, where, casting, order, …])`
Поэлементно вычисляет наименьшее общее кратное массивов x1 и x2.

`gcd(x1, x2, / [, out, where, casting, order, …])`

Поэлементно вычисляет наибольший общий делитель массивов x1 и x2.

`add(x1, x2, / [, out, where, casting, order, …])`

Поэлементная сумма значений массивов.

`reciprocal(x, / [, out, where, casting, …])`

Вычисляет обратное значение (1/x) каждого элемента массива.

`positive(x, / [, out, where, casting, order, …])`

Эквивалентно простому копированию (numpy.copy) элементов массива, но только для массивов поддерживающих математические операции. Формально соответствует математической записи b = +a.

`negative(x, / [, out, where, casting, order, …])`

Отрицательное значение элементов массива.

`multiply(x1, x2, / [, out, where, casting, …])`

Поэлементное умножение значений массива x1 на значения массива x2.

`divide(x1, x2, / [, out, where, casting, …])`

Поэлементное деление значений массива x1 на значения массива x2.

`power(x1, x2, / [, out, where, casting, …])`

Поэлементное возведение значений массива x1 в степень равную значениям из массива x2.

`subtract(x1, x2, / [, out, where, casting, …])`

Поэлементная разность значений массива x1 и x2.

`true_divide(x1, x2, / [, out, where, …])`

Поэлементное истинное деление значений массива x1 на значения массива x2.

`floor_divide(x1, x2, / [, out, where, …])`

Поэлементное целочисленное деление значений массива x1 на значения массива x2.

`float_power(x1, x2, / [, out, where, …])`

Поэлементное возведение значений массива x1 в степень равную значениям из массива x2, адаптированное для чисел с плавающей точкой.

`fmod(x1, x2, / [, out, where, casting, …])`

Поэлементный остаток от деления значений массива x1 на значения массива x2.

`mod(x1, x2, / [, out, where, casting, order, …])`

Поэлементно вычисляет остаток от деления значений массива x1 на значения массива x2.

`modf(x[, out1, out2], / [, out, where, …])`

Дробная и целая часть элементов массива.

`remainder(x1, x2, / [, out, where, casting, …])`

Элементарный остаток от деления значений массива x1 на значения массива x2.

`divmod(x1, x2[, out1, out2], / [[, out, …])`

Результат истинного деления и остаток от деления значений массива x1 на значения массива x2.

■ *Операции с комплексными числами*

`angle(z[, deg])`

Вычисляет угол каждого комплексного числа в массиве.

`real(val)`

Действительная часть комплексного числа.

`imag(val)`

Мнимая часть комплексного числа.

`conj(x, / [, out, where, casting, order, …])`
Комплексно-сопряженный элемент.

■ *Прочие математические функции*

`convolve(a, v[, mode])`
Дискретная линейная свертка.

`clip(a, a_min, a_max[, out])`
Ограничение значений массивов указанным интервалом допустимых значений.

`sqrt(x, / [, out, where, casting, order, …])`
Квадратный корень элементов массива.

`cbrt(x, / [, out, where, casting, order, …])`
Кубический корень элементов массива.

`square(x, / [, out, where, casting, order, …])`
Квадрат элементов массива.

`absolute(x, / [, out, where, casting, order, …])`
Абсолютное значение (модуль) элементов массива.

`fabs(x, / [, out, where, casting, order, …])`
Возвращает абсолютное значение (модуль) элементов массива в виде чисел с плавающей точкой.

`sign(x, /[, out, where, casting, order, …])`
Элементарный указатель на знак числа.

`heaviside(x1, x2, / [, out, where, casting, …])`
Ступенчатая функция Хевисайда.

`maximum(x1, x2, / [, out, where, casting, …])`
Наибольшие значения после поэлементного сравнения значений массивов.

`minimum(x1, x2, /[, out, where, casting, …])`
Наименьшие значения после поэлементного сравнения значений массивов.

`fmax(x1, x2, / [, out, where, casting, …])`
Наибольшие значения после поэлементного сравнения значений массивов в виде чисел с плавающей точкой.

`fmin(x1, x2, / [, out, where, casting, …])`
Наименьшие значения после поэлементного сравнения значений массивов в виде чисел с плавающей точкой.

`nan_to_num(x[, copy])`
Заменяет nan на 0, бесконечность и минус-бесконечность заменяются на наибольшее и наименьшее доступное число с плавающей точкой соответственно.

`real_if_close(a[, tol])`
Переводит комплексные числа в вещественные если мнимая часть комплексного числа меньше машинной эпсилон.

`interp(x, xp, fp[, left, right, period])`
Одномерная линейная интерполяция.

■ Аргументы универсальных функций

Все универсальные функции могут принимать целый ряд необязательных аргументов. Большинство данных аргументов необходимы для тонкой настройки работы функций и чаще всего совсем не используются, хотя, в определенных ситуациях некоторые из аргументов могут оказаться очень полезными.

■ out

Определяет место в которое будет сохранен результат работы функции. Если параметр не указан или указано None (по умолчанию), то будет возвращен автоматически созданный массив. Если в качестве параметра указан массив NumPy, то он должен иметь форму совместимую с формой входного массива. Если универсальная функция создает несколько выходных массивов, то для каждого из них можно указать место сохранения с помощью кортежа из массивов NumPy или None.

Данный параметр очень полезен при больших объёмах вычислений, так как позволяет избежать создания временного массива, записывая результат, непосредственно в то место памяти в которое необходимо.

```
>>> import numpy as np
>>>
>>> x = np.arange(0, 100, 15)
>>> x
array([ 0, 15, 30, 45, 60, 75, 90])
>>>
>>> y = np.ones(7)
>>> y
array([1., 1., 1., 1., 1., 1., 1.])
>>>
>>> np.sin(x, out = y)
array([ 0.        ,  0.65028784, -0.98803162,  0.85090352, -0.30481062,
       -0.38778164,  0.89399666])
>>>
>>> y
array([ 0.        ,  0.65028784, -0.98803162,  0.85090352, -0.30481062,
       -0.38778164,  0.89399666])
>>>
>>>
>>> #  В данном параметре можно указывать представления массивов:
...
>>> y = np.zeros(7)
>>> y
array([0., 0., 0., 0., 0., 0., 0.])
>>>
>>> np.sin(x[::2], out = y[::2])
array([ 0.        , -0.98803162, -0.30481062,  0.89399666])
>>>
>>> y
array([ 0.        ,  0.        , -0.98803162,  0.        , -0.30481062,
        0.        ,  0.89399666])
```

Параметр out можно не использовать и просто записывать что-то вроде y[::2] = sin(x[::2]), но такая запись приводит к созданию временного массива для хранения результата вычислений sin(x[::2]) и затем к еще одной операции копирования результатов из временного массива в y[::2]. Использование аргумента out при обработке больших массивов может обеспечить значительный выигрыш в экономии используемой памяти и затраченного на вычисления времени.

■ where

Данный параметр принимает True (по умолчанию), False или массив логических значений в случае если универсальная функция возвращает несколько результирующих массивов. Значение True указывает на вычисление универсальной функции с сохранением результата в указанный в параметре out массив. В случае указания False, будут возвращены значения массива, который указан в out.

```
>>> x
array([ 0, 15, 30, 45, 60, 75, 90])
>>>
>>> np.cos(x, where = False)
array([ 0.        ,  0.65028784, -0.98803162,  0.85090352, -0.30481062,
       -0.38778164,  0.89399666])
>>>
>>> y = np.zeros(7)
>>> y
array([0., 0., 0., 0., 0., 0., 0.])
>>>
>>> np.cos(x, out = y, where = False)
array([0., 0., 0., 0., 0., 0., 0.])
>>>
>>> np.cos(x, out = y, where = True)
array([ 1.        , -0.75968791,  0.15425145,  0.52532199, -0.95241298,
        0.92175127, -0.44807362])
>>>
>>> y
array([ 1.        , -0.75968791,  0.15425145,  0.52532199, -0.95241298,
        0.92175127, -0.44807362])
```

Параметр where может оказаться полезен, в ситуациях когда обработку данных и сохранение вычислений необходимо выполнять в зависимости от некоторого условия. Например, необходимость обработки данных зависит от некоторого параметра, который должен находиться в пределах некоторого интервала:

```
>>> x = np.arange(0, 100, 15)
>>> y = np.zeros(7)
>>>
>>> a, b = 0, 1      #  Границы интервала
>>> c = 0.77         #  Параметр, определяющий необходимость вычислений
>>>
>>>  #  Если параметр находится в пределах интервала,
...  #  то результат будет вычислен и помещен в указанный массив:
...
>>> np.cos(x, out = y, where = a < c < b)
array([ 1.        , -0.75968791,  0.15425145,  0.52532199, -0.95241298,
        0.92175127, -0.44807362])
>>> y
array([ 1.        , -0.75968791,  0.15425145,  0.52532199, -0.95241298,
        0.92175127, -0.44807362])
>>>
>>> x = np.arange(0, 70, 10)      #  Новые данные в массиве 'x'
>>> x
array([ 0, 10, 20, 30, 40, 50, 60])
>>>
>>>  #  В случае, если бы параметр 'c' оказался в пределах интервала
...  #  от 0 до 1, то в массиве 'y' мы бы увидели:
```

```
...
>>> np.cos(x)
array([ 1.        , -0.83907153,  0.40808206,  0.15425145, -0.66693806,
        0.96496603, -0.95241298])
>>>
>>> c = 1.01    #  Допустим параметр вышел за пределы интервала
>>>
>>> #  Тогда в массиве 'y' мы увидим предыдущие значения:
>>> np.cos(x, out = y, where = a < c < b)
array([ 1.        , -0.75968791,  0.15425145,  0.52532199, -0.95241298,
        0.92175127, -0.44807362])
```

- **casting**

Позволяет настроить преобразование типов данных при вычислениях. Данный параметр может принимать следующие значения:

- 'no' - типы данных не преобразуются;
- 'equiv' - допускается только изменение порядка байтов;
- 'safe' - допускаются только те типы данных, которые сохраняют значения;
- 'same_kind' - допускаются только безопасные преобразования, такие как float64 в float32;
- 'usafe' - допускаются любые преобразования данных.

```
>>> x = np.arange(0, 70, 10, dtype = np.float96)
>>> y = np.zeros(7, dtype = np.float16)
>>>
>>> x
array([ 0., 10., 20., 30., 40., 50., 60.], dtype=float96)
>>> y
array([0., 0., 0., 0., 0., 0., 0.], dtype=float16)
>>>
>>> np.cos(x)
array([ 1.        , -0.83907153,  0.40808206,  0.15425145, -0.66693806,
        0.96496603, -0.95241298], dtype=float96)
>>>
>>> np.cos(x, out = y, casting = 'same_kind')    #  значение 'casting' по
умолчанию
array([ 1.    , -0.839 ,  0.4082,  0.1543, -0.667 ,  0.965 , -0.9526],
      dtype=float16)
>>>
>>> y     #  Произошла потеря точности
array([ 1.    , -0.839 ,  0.4082,  0.1543, -0.667 ,  0.965 , -0.9526],
      dtype=float16)
>>>
>>> np.cos(x, out = y, casting = 'safe')    #  Запрещает небезопасные
преобразования
Traceback (most recent call last):
  File "", line 1, in
TypeError: ufunc 'cos' output (typecode 'g') could not be coerced to provided
output
parameter (typecode 'e') according to the casting rule ''safe''
```

■ order

Этот параметр определяет в каком порядке выходные массивы должны храниться в памяти: строчном C-стиле или столбчатом стиле Fortran. Если входной массив не является массивом NumPy, то созданный массив будет находиться в памяти в строковом C порядке, если указать флаг 'F', то будет храниться в столбчатом порядке 'Fortran'. Если входной массив - это массив NumPy, то флаг 'K' либо сохраняет порядок исходного массива либо устанавливает самый близкий по структуре; флаг 'A' установит макет памяти выходного массива в 'F' если массив а является смежным со столбчатым стилем Fortran, в противном случае макет памяти будет установлен в 'C'. По умолчанию флаг установлен в значение 'K'.

```
>>> x = np.arange(0, 100, 30).reshape(2,2)
>>> x
array([[ 0, 30],
       [60, 90]])
>>> y = np.cos(x)
>>>
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>>
>>> y = np.cos(x, order = 'F')
>>>
>>> y.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

■ dtype

Позволяет переопределить тип данных выходного массива:

```
>>> x = np.arange(0, 100, 30)
>>> x
array([ 0, 30, 60, 90])
>>>
>>> y = np.cos(x)
>>> y
array([ 1.        ,  0.15425145, -0.95241298, -0.44807362])
>>>
>>> y.dtype
dtype('float64')
>>>
>>> y = np.cos(x, dtype = np.float16)
>>> y
array([ 1.    ,  0.1543, -0.9526, -0.448 ], dtype=float16)
```

■ subok

Если установлено значение True, то подклассы будут сохраняться, если False, то результатом будет базовый класс ndarray:

```
>>> x = np.matrix([0, 30, 60, 90])
>>> x
matrix([[ 0, 30, 60, 90]])
>>>
>>> y = np.cos(x, subok = True)     #  По умолчанию подклассы сохраняются
>>> y
matrix([[ 1.        ,  0.15425145, -0.95241298, -0.44807362]])
>>>
>>> y = np.cos(x, subok = False)
>>> y
array([[ 1.        ,  0.15425145, -0.95241298, -0.44807362]])
```

■ signature

Большинство универсальных функций реализованы в скомпилированном C-коде. Основные вычисления внутри универсальной функции заложены в ее базовом цикле. Точнее, таких циклов несколько и в зависимости от типа входных данных функция выбирает самый подходящий из них. Аргумент signature позволяет указать какой именно из этих циклов должен использоваться.

В качестве аргумента signature нужно указать специальную строку-подпись цикла, либо тип данных, либо кортеж типов данных. Получить список доступных строк-подписей можно с помощью атрибута types объекта ufunc (np.cos – это объект ufunc и у него атрибут types):

```
>>> import numpy as np
>>>
>>> np.cos.types
['e->e', 'f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']
>>>
>>> x = np.arange(0, 100, 30)
>>> x
array([ 0, 30, 60, 90])
>>>
>>> np.cos(x, signature = 'f->f')
array([ 1.        ,  0.15425146, -0.95241296, -0.44807363], dtype=float32)
>>>
>>> np.cos(x, signature = 'F->F')
array([ 1.        -0.j,  0.15425146+0.j, -0.95241296+0.j, -0.44807363-0.j],
      dtype=complex64)
>>>
>>> np.cos(x, signature = 'F')
array([ 1.        -0.j,  0.15425146+0.j, -0.95241296+0.j, -0.44807363-0.j],
      dtype=complex64)
>>>
>>> np.cos(x, signature = np.complex64)
Traceback (most recent call last):
  File "", line 1, in
TypeError: No loop matching the specified signature and casting
was found for ufunc cos
```

Данный параметр предоставляет небольшую, но все же оптимизацию вычислений - вместо автоматического поиска подходящего цикла, функция сразу переходит к вычислениям в указанном цикле. Скорее всего указание данного аргумента будет полезным в вычислениях в которых вызов универсальных функций происходит очень часто.

■ extobj

В качестве данного аргумента выступает список из трёх целых чисел. Первое число определяет размер буфера данных, второе - режим ошибки, третье - функция обратного вызова ошибки:

```
>>> np.cos(x, extobj = [160, 1, 1])
array([ 1.        ,  0.15425145, -0.95241298, -0.44807362])
```

Данный аргумент может быть полезен в вычислениях с частым вызовом универсальных функций, которые обрабатывают небольшие массивы данных.

■ Универсальные функции. Продолжение

Продолжение описания универсальных функций: спасибо документации...

This is documentation for an old release of NumPy (version 1.10.0). Read this page in the documentation of the latest stable release (version > 1.17).

numpy.tanh

numpy.tanh(x[, out]) = <ufunc 'tanh'>

Compute hyperbolic tangent element-wise.

Equivalent to np.sinh(x)/np.cosh(x) or -1j * np.tan(1j*x).

Parameters:

x : array_like

Input array.

out : ndarray, optional

Output array of same shape as x.

Returns:

y : ndarray

The corresponding hyperbolic tangent values.

Raises:

ValueError: invalid return array shape

if out is provided and out.shape != x.shape (See Examples)

Notes

If out is provided, the function writes the result into it, and returns a reference to out. (See Examples)

Пока неясно, что означает эта нотация:

numpy.tanh(x[, out]) = <ufunc 'tanh'>

Это такое объявление? Показывает, что это универсальная функция?

- **numpy.sin**

```
numpy.sin(x, *ufunc_args) = <ufunc 'sin'>
```

Функция sin() вычисляет тригонометрический синус элементов массива.

Параметры:

x - подобный массиву объект

Массив чисел задающих угол в радианах (360 градусов равняются 2π радиан).

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Синус элементов x.

Смотреть также: arcsin, sinh, cos, tan

Примеры

```
>>> import numpy as np
>>>
>>> #  Вычисление синуса одного угла:
... np.sin(0)
0.0
>>> np.sin(np.pi/2)
1.0
>>> np.sin(np.pi/6)
0.49999999999999994
>>> np.sin(np.pi)
1.2246467991473532e-16
>>>
>>>
>>> #  Массив значений углов, заданных в радианах:
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>> np.sin(x)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
```

```
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
>>>
>>>
>>> #  Массив значений углов, заданных в градусах:
>>> x = np.array([0, 30, 45, 60, 90])*np.pi/180
>>> x
array([0.        , 0.52359878, 0.78539816, 1.04719755, 1.57079633])
>>>
>>> np.sin(x)
array([0.        , 0.5       , 0.70710678, 0.8660254 , 1.        ])
```

- **numpy.cos**

numpy.cos(x,*ufunc_args)= <ufunc 'cos'>

Функция cos() вычисляет тригонометрический косинус элементов массива.

Параметры:

x - подобный массиву объект

Массив чисел задающих угол в радианах (360 градусов равняются 2π радиан).

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Косинус элементов x.

Смотреть также: arccos, cosh, sin, tan

Примеры

```
>>> import numpy as np
>>>
>>> #  Вычисление синуса одного угла:
... np.cos(0)
1.0
>>> np.cos(np.pi/2)
6.123233995736766e-17
>>> np.cos(np.pi/3)
0.5000000000000001
>>>
>>> #  Массив значений углов, заданных в радианах:
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>> np.cos(x)
array([ 1.        ,  0.54030231, -0.41614684, -0.9899925 , -0.65364362,
        0.28366219,  0.96017029,  0.75390225, -0.14550003, -0.91113026])
>>>
>>>
>>> #  Массив значений углов, заданных в градусах:
>>> x = np.array([0, 30, 45, 60, 90])*np.pi/180
>>> x
```

```
array([0.        , 0.52359878, 0.78539816, 1.04719755, 1.57079633])
>>>
>>> np.cos(x)
array([1.00000000e+00, 8.66025404e-01, 7.07106781e-01, 5.00000000e-01,
       6.12323400e-17])
```

■ numpy.tan

```
numpy.tan(x, *ufunc_args) = <ufunc 'tan'>
```

Функция tan() вычисляет тригонометрический тангенс элементов массива и эквивалентна np.sin(x)/np.cos(x).

Параметры:

x - подобный массиву объект

Массив чисел задающих угол в радианах (360 градусов равняются 2π радиан).

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Тангенс элементов x.

Смотреть также: arctan, tanh, sin, cos

Примеры

```
>>> import numpy as np
>>>
>>> #  Вычисление синуса одного угла:
... np.tan(0)
0.0
>>> np.tan(np.pi/2)
1.633123935319537e+16
>>> np.tan(np.pi/4)
0.9999999999999999
>>>
>>>
>>> #  Массив значений углов, заданных в радианах:
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>> np.tan(x)
array([ 0.        ,  1.55740772, -2.18503986, -0.14254654,  1.15782128,
       -3.38051501, -0.29100619,  0.87144798, -6.79971146, -0.45231566])
>>>
>>>
>>> #  Массив значений углов, заданных в градусах:
>>> x = np.array([0, 30, 45, 60, 90])*np.pi/180
>>> x
array([0.        , 0.52359878, 0.78539816, 1.04719755, 1.57079633])
>>>
>>> np.tan(x)
```

```
array([0.00000000e+00, 5.77350269e-01, 1.00000000e+00, 1.73205081e+00,
       1.63312394e+16])
```

- **numpy.arcsin**

```
numpy.arcsin(x, *ufunc_args) = <ufunc 'arcsin'>
```

Функция arcsin() вычисляет тригонометрический арксинус (обратный синус), если y = sin(x),

то x = arcsin(y).

Параметры:

x - подобный массиву объект

y - координата или массив y-координат единичной окружности.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Арксинус элементов x в интервале [-pi/2, pi/2].

Замечание

arcsin(x) - многозначная функция, т.е. для каждого x существует бесконечное количество значений углов α при которых sin(α) = x, поэтому принято соглашение о том, что функция numpy.arcsin(x) возвращает значение угла в интервале [-pi/2, pi/2].

Для комплексных входных значений arcsin так же представляет собой бесконечнозначную функцию, которая, по соглашению находится на листе D0 с разрезами [-inf, -1] и [1, inf].

Иногда арксинус обозначается как asin или sin-1

Смотреть также: sin, sinh, arccos, arctan

Примеры

```
>>> import numpy as np
>>>
>>> np.arcsin(0.77)
0.8788411516685797
>>>
>>> x = np.array([-1, -0.5, 0, 0.5, 1])
>>>
>>> np.arcsin(x)     #  Значение углов в радианах
array([-1.57079633, -0.52359878,  0.        ,  0.52359878,  1.57079633])
>>>
>>> np.arcsin(x)*180/np.pi     #  Значение углов в градусах
array([-90., -30.,   0.,  30.,  90.])
```

- **numpy.arccos**

```
numpy.arccos(x, *ufunc_args) = <ufunc 'arccos'>
```

Функция arccos() вычисляет тригонометрический арккосинус (обратный косинус), если y = cos(x), то x = arccos(y).

Параметры:

x - подобный массиву объект

y - координата или массив y-координат единичной окружности.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Арккосинус элементов x в интервале [0, pi].

Замечание

arccos(x) - многозначная функция, т.е. для каждого x существует бесконечное количество значений углов α при которых cos(α) = x, поэтому принято соглашение о том, что функция numpy.arccos(x) возвращает значение угла в интервале [0, pi].

Для комплексных входных значений arccos так же представляет собой бесконечнозначную функцию, которая, по соглашению находится на листе D0 с разрезами [-inf, -1] и [1, inf].

Иногда арккосинус обозначается как acos или cos-1

Смотреть также: cos, cosh, arcsin, arctan

Примеры

```
>>> import numpy as np
>>>
>>> np.arccos(0.77)
0.6919551751263169
>>>
>>> x = np.array([-1, -0.5, 0, 0.5, 1])
>>>
>>> np.arccos(x)      #  Значение углов в радианах
array([3.14159265, 2.0943951 , 1.57079633, 1.04719755, 0.        ])
>>>
>>> np.arccos(x)*180/np.pi     #  Значение углов в градусах
array([180., 120.,  90.,  60.,   0.])
```

■ numpy.arctan

```
numpy.arctan(x, *ufunc_args) = <ufunc 'arctan'>
```

Функция arctan() вычисляет тригонометрический арктангенс (обратный тангенс), если y = tan(x), то x = arctan(y).

Параметры:

x - подобный массиву объект

y - координата или массив y-координат единичной окружности.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Арктангенс элементов x в интервале [-pi/2, pi/2] (arctan(-inf) = -pi/2 и arctan(inf) = pi/2).

Замечание

arctan(x) - многозначная функция, т.е. для каждого x существует бесконечное количество значений углов α при которых tan(α) = x, поэтому принято соглашение о том, что функция numpy.arctan(x) возвращает значение угла в интервале [0, pi].

Для комплексных входных значений arctan так же представляет собой бесконечнозначную функцию, которая, по соглашению находится на листе D0 с разрезами [1j, infj] и [-1j, -infj].

Иногда арккосинус обозначается как atan или tan-1

Смотреть также: tan, tanh, arcsin, arccos

Примеры

```
>>> import numpy as np
>>>
>>> np.arctan(0.77)
0.6561787179913949
>>>
>>> x = np.array([-np.inf, -1, -0.5, 0, 0.5, 1, np.inf])
>>>
>>> np.arctan(x)     #  Значение углов в радианах
array([-1.57079633, -0.78539816, -0.46364761,  0.        ,  0.46364761,
        0.78539816,  1.57079633])
>>>
>>> np.arctan(x)*180/np.pi     #  Значение углов в градусах
array([-90.        , -45.        , -26.56505118,   0.        ,
        26.56505118,  45.        ,  90.        ])
```

- **numpy.hypot**

```
numpy.hypot(x1, x2, *ufunc_args) = <ufunc 'hypot'>
```

Функция hypot() вычисляет длинну гипотенузы по указанным значениям катетов. Эквивалентно sqrt(x1**2 + x2**2).

Параметры:

x1, x2 - подобные массиву объекты

Длины катетов или массивы значений длин катетов.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Длинна гипотенузы прямоугольного треугольника или массив значений длин гипотенузы.

Смотреть также: sin, cos, tan

Примеры

```
>>> import numpy as np
>>>
>>> np.hypot(3, 4)
5.0
>>>
>>> np.hypot(3, [4, 5, 6])    #  Поддерживает механизм транслирования
array([5.        , 5.83095189, 6.70820393])
>>>
>>> a = np.array([3, 5, 8, 7])
>>> b = np.array([4, 12, 15, 24])
>>>
>>> np.hypot(a, b)
array([ 5., 13., 17., 25.])
>>>
>>> a = np.arange(1, 17).reshape(4, 4)
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
>>>
>>> b = np.arange(16, 0, -1).reshape(4, 4)
>>> b
array([[16, 15, 14, 13],
       [12, 11, 10,  9],
       [ 8,  7,  6,  5],
       [ 4,  3,  2,  1]])
>>>
>>> np.hypot(a, b)
array([[16.03121954, 15.13274595, 14.31782106, 13.60147051],
       [13.        , 12.52996409, 12.20655562, 12.04159458],
       [12.04159458, 12.20655562, 12.52996409, 13.        ],
```

```
[13.60147051, 14.31782106, 15.13274595, 16.03121954]])
```

- **numpy.arctan2**

```
numpy.arctan2(x, *ufunc_args) = <ufunc 'arctan2'>
```

Функция arctan2() вычисляет arctan(x1/x2) и возвращает значение угла в правильном квадранте (четверти координатной плоскости).

Параметры:

x1 - число или подобный массиву объект

y - координата или массив y-координат.

x2 - подобный массиву объект

x - координата или массив x-координат.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Угол или массив углов в радианах из диапазона [-pi, pi].

Замечание

Квадрант вычисляется из учета, что arctan2(x1, x2) является знаком угла (в радианнах) между двумя лучами, первый, проходит через начало координат и точку (1, 0), а второй через начало координат и точку (x2, x1). В данном случае x2 соответствует y-координате, а x1 соответствует x-координате.

По соглашению IEEE функция arctan2(x1, x2) определена для x2 = +/-0, а также для одного или одновременно обоих значений x1 и x2 равных +/-inf.

Для функции arctan2(x1, x2) определены следующие специальные значения:

x1	x2	arctan2(x1, x2)
+/- 0	+0	+/- 0
+/- 0	-0	+/- pi
> 0	+/- inf	+0 / +pi
< 0	+/- inf	-0 / -pi
+/- inf	+inf	+/- (pi/4)
+/- inf	-inf	+/- (3*pi/4)

Значения -0 и +0 являются разными числами с плавающей точкой, точно так же как и значения -inf и +inf.

Данная функция не определена для комплексных аргументов. Для вычисления углов комплексных значений можно воспользоваться функцией angle.

Смотреть также: arctan, tan, angle

Примеры

```
>>> import numpy as np
>>>
>>> np.arctan2(0.77, -0.33)
1.9756881130799802
>>>
>>> #  Рассмотрим 4-е точки в разных квадрантах
...
>>> x = np.array([+1, +1, -1, -1])
>>> y = np.array([+1, -1, +1, -1])
>>>
>>> np.arctan2(y, x)    #  Значения углов в радианах
array([ 0.78539816, -0.78539816,  2.35619449, -2.35619449])
>>>
>>> np.arctan2(y, x)*180/np.pi    #  Значения углов в градусах
array([  45.,  -45.,  135., -135.])
```

■ numpy.degrees

```
numpy.degrees(x, *ufunc_args) = <ufunc 'degrees'>
```

Функция degrees() выполняет преобразование радианной меры угла в градусную.

Параметры:

x - число или подобный массиву объект

y - значение или массив значений в радианах.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Значение или массив значений в градусной мере.

Смотреть также: rad2deg, deg2rad, radians

Примеры

```
>>> import numpy as np
>>>
>>> np.degrees(np.pi/6)
```

```
29.999999999999996
>>>
>>> rad = np.arange(7)*np.pi/6    #  Значения в радианах
>>> rad
array([0.        , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
       2.61799388, 3.14159265])
>>>
>>> deg = np.degrees(rad)    #  Радианы, конвертированные в градусы
>>> deg
array([  0.,  30.,  60.,  90., 120., 150., 180.])
```

- **numpy.radians**

```
numpy.radians(x, *ufunc_args) = <ufunc 'radians'>
```

Функция radians() выполняет преобразование градусной меры угла в радианную.

Параметры:

x - число или подобный массиву объект

y - значение или массив значений в градусах.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Значение или массив значений в радианной мере.

Смотреть также: degrees, rad2deg, deg2rad

Примеры

```
>>> import numpy as np
>>>
>>> np.radians(30)      #  30 градусов
0.5235987755982988
>>>
>>> np.pi/6             #  это действительно pi/6 радиан
0.5235987755982988
>>>
>>>
>>> deg = np.arange(0, 200, 30)     #  Значения в градусах
>>> deg
array([  0,  30,  60,  90, 120, 150, 180])
>>>
>>> rad = np.radians(deg    #  Градусы, конвертированные в радианы
>>> rad
array([0.        , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
       2.61799388, 3.14159265])
```

- **numpy.deg2rad**

```
numpy.deg2rad(x, *ufunc_args) = <ufunc 'deg2rad'>
```

Функция deg2rad() выполняет преобразование градусной меры угла в радианную, эквивалентно x*pi/180.

Параметры:

x - число или подобный массиву объект

y - значение или массив значений в градусах.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Значение или массив значений в радианной мере.

Смотреть также: degrees, rad2deg, deg2rad

Примеры

```
>>> import numpy as np
>>>
>>> np.deg2rad(30)      #  30 градусов
0.5235987755982988
>>>
>>> np.pi/6             #  это действительно pi/6 радиан
0.5235987755982988
>>>
>>> deg = np.arange(0, 200, 30)     #  Значения в градусах
>>> deg
array([  0,  30,  60,  90, 120, 150, 180])
>>>
>>> rad = np.deg2rad(deg)     #  Градусы, конвертированные в радианы
>>> rad
array([0.        , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
       2.61799388, 3.14159265])
```

■ numpy.rad2deg

```
numpy.rad2deg(x, *ufunc_args) = <ufunc 'rad2deg'>
```

Функция rad2deg() выполняет преобразование радианной меры угла в градусную, эквивалентна 180*x/pi.

Параметры:

x - число или подобный массиву объект

Значение или массив значений в радианах.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - массив NumPy или его подкласс

Значение или массив значений в градусной мере.

Смотреть также: rad2deg, deg2rad, radians

Примеры

```
>>> import numpy as np
>>>
>>> np.rad2deg(np.pi/6)
29.999999999999996
>>>
>>> rad = np.arange(7)*np.pi/6    #  Значения в радианах
>>> rad
array([0.        , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
       2.61799388, 3.14159265])
>>>
>>> deg = np.rad2deg(rad)    #  Радианы, конвертированные в градусы
>>> deg
array([  0.,  30.,  60.,  90., 120., 150., 180.])
```

- **numpy.sinh**

```
numpy.sinh(x, *ufunc_args) = <ufunc 'sinh'>
```

Функция sinh() вычисляет гиперболический синус элементов массива и эквивалентна 1/2*(np.exp(x) - np.exp(-x)) или -1j*np.sin(1j*x).

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Гиперболический синус входного числа или каждого числа входного массива.

Замечание

Часто гиперболический синус обозначается sh.

Смотреть также: arcsinh, cosh, tanh

Примеры

```
>>> import numpy as np
>>>
>>> np.sinh(0)
```

```
0.0
>>>
>>> x = np.linspace(0, np.pi, num = 7)*1j
>>> x
array([0.+0.        j, 0.+0.52359878j, 0.+1.04719755j, 0.+1.57079633j,
       0.+2.0943951 j, 0.+2.61799388j, 0.+3.14159265j])
>>>
>>> np.sinh(x)
array([ 0.+0.00000000e+00j,  0.+5.00000000e-01j,  0.+8.66025404e-01j,
        0.+1.00000000e+00j, -0.+8.66025404e-01j, -0.+5.00000000e-01j,
       -0.+1.22464680e-16j])
```

- **numpy.cosh**

```
numpy.cosh(x, *ufunc_args) = <ufunc 'cosh'>
```

Функция cosh() вычисляет гиперболический косинус элементов массива, эквивалентна 1/2*(np.exp(x) + np.exp(-x)) или np.cos(1j*x).

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Гиперболический косинус входного числа или каждого числа входного массива.

Замечание

Часто гиперболический косинус обозначается ch.

Смотреть также: arccosh, sinh, tanh

Примеры

```
>>> import numpy as np
>>>
>>> np.sinh(0)
0.0
>>>
>>> x = np.linspace(0, np.pi, num = 7)*1j
>>> x
array([0.+0.        j, 0.+0.52359878j, 0.+1.04719755j, 0.+1.57079633j,
       0.+2.0943951 j, 0.+2.61799388j, 0.+3.14159265j])
>>>
>>> np.cosh(x)
array([ 1.00000000e+00+0.j,  8.66025404e-01+0.j,  5.00000000e-01+0.j,
        6.12323400e-17+0.j, -5.00000000e-01+0.j, -8.66025404e-01+0.j,
       -1.00000000e+00+0.j])
```

■ numpy.tanh

```
numpy.tanh(x, *ufunc_args) = <ufunc 'tanh'>
```

Функция tanh() вычисляет гиперболический тангенс элементов массива и эквивалентна np.sinh(x)/np.cosh(x) или -1j*np.tan(1j*x).

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Гиперболический тангенс входного числа или каждого числа входного массива.

Замечание

Часто гиперболический косинус обозначается th.

Смотреть также: arctanh, cosh, sinh

Примеры

```
>>> import numpy as np
>>>
>>> np.sinh(0)
0.0
>>>
>>> x = np.linspace(0, np.pi/2, num = 7)*1j
>>> x
array([0.+0.        j, 0.+0.26179939j, 0.+0.52359878j, 0.+0.78539816j,
       0.+1.04719755j, 0.+1.30899694j, 0.+1.57079633j])
>>>
>>> np.tanh(x)
array([0.+0.00000000e+00j, 0.+2.67949192e-01j, 0.+5.77350269e-01j,
       0.+1.00000000e+00j, 0.+1.73205081e+00j, 0.+3.73205081e+00j,
       0.+1.63312394e+16j])
```

■ numpy.arcsinh

```
numpy.arcsinh(x, *ufunc_args) = <ufunc 'tanh'>
```

Функция arcsinh() вычисляет обратный гиперболический синус (ареасинус) элементов массива.

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Обратный гиперболический синус входного числа или каждого числа входного массива.

Замечание

arcsinh(x) является многозначной функцией, т.е. для каждого входного числа x существует бесконечное количество таких чисел y, что sinh(y) = x. По соглашению, данная функция возвращает число Y у которого мнимая часть лежит в интервале [-pi/2, pi/2].

Для комплексных входных значений arcsinh так же представляет собой бесконечнозначную функцию, которая, по соглашению находится на листе D0 с разрезами [1j, infj] и [-1j, -infj].

Часто обратный гиперболический синус обозначается arsh, asinh, arsinh или sinh-1.

Смотреть также: arccosh, arctanh, sinh

Примеры

```
>>> import numpy as np
>>>
>>> np.arcsinh(0)
0.0
>>>
>>> np.arcsinh(1)
0.881373587019543
>>>
>>> np.sinh(0.881373587019543)
1.0
>>>
>>> x = np.linspace(0, np.pi/2, num = 7)*1j
>>> np.arcsinh(x)
array([0.        +0.        j, 0.        +0.26488615j,
       0.        +0.55106958j, 0.        +0.90333911j,
       0.30604211+1.57079633j, 0.76717348+1.57079633j,
       1.02322748+1.57079633j])
```

- **numpy.arccosh**

```
numpy.arccosh(x, *ufunc_args) = <ufunc 'tanh'>
```

Функция arccosh() вычисляет обратный гиперболический косинус (ареакосинус).

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Обратный гиперболический косинус входного числа или каждого числа входного массива.

Замечание

arccosh(x) является многозначной функцией, т.е. для каждого входного числа x существует бесконечное количество таких чисел y, что sinh(y) = x. По соглашению, данная функция возвращает число y у которого мнимая часть лежит в интервале [-pi, pi], а действительная часть в интервале [0, inf].

Для комплексных входных значений arccosh так же представляет собой бесконечнозначную функцию, которая, по соглашению находится на листе D0 с разрезом [-inf, 1].

Часто обратный гиперболический косинус обозначается arch, arcosh, acosh или sinh-1.

Смотреть также: arcsinh, arctanh, cosh

Примеры

```
>>> import numpy as np
>>>
>>> np.arccosh(0)
__main__:1: RuntimeWarning: invalid value encountered in arccosh
nan
>>> np.arccosh(1)
0.0
>>>
>>> np.cosh(0)
1.0
>>>
>>> x = np.linspace(0, np.pi/2, num = 7)*1j
>>> np.arccosh(x)
array([0.        +1.57079633j, 0.25889745+1.57079633j,
       0.50221899+1.57079633j, 0.72122549+1.57079633j,
       0.91435666+1.57079633j, 1.08392469+1.57079633j,
       1.23340312+1.57079633j])
```

- **numpy.arctanh**

```
numpy.arctanh(x, *ufunc_args) = <ufunc 'tanh'>
```

Функция arctanh() вычисляет обратный гиперболический тангенс (ареатангенс) элементов массива.

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Обратный гиперболический тангенс входного числа или каждого числа входного массива.

Замечание

arctanh(x) является многозначной функцией, т.е. для каждого входного числа x существует бесконечное количество таких чисел y, что tanh(y) = x. По соглашению, данная функция возвращает число y у которого мнимая часть лежит в интервале [-pi/2, pi/2].

Для комплексных входных значений arctanh так же представляет собой бесконечнозначную функцию, которая, по соглашению находится на листе D0 с разрезами [-1, -inf] и [1, inf].

Часто Обратный гиперболический синус обозначается arth, artanh, atanh или tanh-1.

Смотреть также: arcsinh, arccosh, tanh

Примеры

```
>>> import numpy as np
>>>
>>> np.arctanh(0)
0.0
>>>
>>> np.arctanh(1)
__main__:1: RuntimeWarning: divide by zero encountered in arctanh
inf
>>>
>>> np.tanh(np.inf)
1.0
>>>
>>> x = np.linspace(0, np.pi/2, num = 7)*1j
>>> np.arctanh(x)
array([0.+0.        j, 0.+0.25605277j, 0.+0.48234791j, 0.+0.66577375j,
       0.+0.80844879j, 0.+0.9184308 j, 0.+1.00388482j])
```

- **numpy.around**

```
numpy.around(a, decimals=0, out=None)
```

Функция around() выполняет равномерное (банковское) округление до указанной позиции к ближайшему четному числу. Реальная и мнимая часть комплексных чисел округляется отдельно.

Параметры:

a - число или подобный массиву объект

Число или последовательность чисел.

decimals - целое число (необязательный аргумент)

Указывает количество знаков после десятичной точки (по умолчанию decimals = 0). Если decimals < 0, то его значение указывает на количество знаков слева от десятичной точки.

out - массив Numpy (необязательный аргумент)

Указывает массив в который будет помещен результат работы функции. Данный массив должен иметь форму идентичную массиву с результатом работы функции. Подробнее о данном параметре смотрите на странице универсальные функции в разделе out.

Возвращает:

результат - массив NumPy

Массив с округленными значениями входного массива a. Тип результирующего массива совпадает с типом входного массива. Если параметр out не указан, то будет создан новый массив.

Смотреть также: rint, fix, trunc

- **numpy.round_**

numpy.round_(a, decimals=0, out=None)

Функция numpy.round_ полностью эквивалентна (вызывает в своем коде) numpy.around.

Параметры:

a - число или подобный массиву объект

Число или последовательность чисел.

decimals - целое число (необязательный аргумент)

Указывает количество знаков после десятичной точки (по умолчанию decimals = 0). Если decimals < 0, то его значение указывает на количество знаков слева от десятичной точки.

out - массив Numpy (необязательный аргумент)

Указывает массив в который будет помещен результат работы функции. Данный массив должен иметь форму идентичную массиву с результатом работы функции. Подробнее о данном параметре смотрите на странице универсальные функции в разделе out.

Возвращает:

результат - массив NumPy

Массив с округленными значениями входного массива a. Тип результирующего массива совпадает с типом входного массива. Если параметр out не указан, то будет создан новый массив.

■ numpy.rint

```
numpy.rint(x, *ufunc_args) = <ufunc 'rint'>
```

Функция rint() округляет элементы массива до ближайшего целого числа.

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Число или массив чисел, которые округлены до ближайшего целого числа.

Смотреть также: fix, floor, ceil, around

Примеры

```
>>> import numpy as np
>>>
>>> np.rint([-5.5, 5.5])
array([-6.,  6.])
>>>
>>> x = np.random.random(5)*10 - 5
>>> x
array([-1.45150824,  2.36081173, -0.01112329,  1.72539084, -3.1361012 ])
>>>
>>> np.rint(x)
array([-1.,  2., -0.,  2., -3.])
```

■ numpy.fix

```
numpy.fix(x, out=None)
```

Функция fix() округляет до ближайшего к нулю целого числа.

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

out - массив Numpy (необязательный аргумент)

Указывает массив в который будет помещен результат работы функции. Данный массив должен иметь форму идентичную массиву с результатом работы функции. Подробнее о данном параметре смотрите на странице универсальные функции в разделе out.

Возвращает:

результат - число или массив NumPy или его подкласс

Число или массив чисел, которые округлены к целому числу ближайшему к нулю.

Смотреть также: floor, ceil, around

Примеры

```
>>> import numpy as np
>>>
>>> np.fix([-5.5, 5.5])
array([-5.,  5.])
>>>
>>> x = np.random.random(5)*10 - 5
>>> x
array([ 2.2019152 ,  2.11741524, -0.7219455 , -3.11994759, -0.71123005])
>>>
>>> np.fix(x)
array([ 2.,  2., -0., -3., -0.])
```

■ numpy.floor

```
numpy.floor(x, *ufunc_args) = <ufunc 'floor'>
```

Функция floor() выполняет округление к меньшему целому числу. Данная функция часто называется пол числа x и обозначается как ⌊x⌋.

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Число или массив чисел, которые округлены к наименьшему целому числу.

Замечание

Некоторые программы вычисляют "пол" отрицательных чисел как "потолок" т.е. floor(-4.75) = -4. В NumPy округление отрицательных чисел выполняется не в сторону нуля а к меньшему числу т.е. floor(-4.75) = -5 т.к. -5 < 4.

Смотреть также: ceil, around, trunc

Примеры

```
>>> import numpy as np
>>>
>>> np.floor([-5.5, 5.5])
array([-6.,  5.])
>>>
```

```
>>> x = np.random.random(5)*10
>>> x
array([0.4947903 , 1.89080601, 9.30142446, 5.73544876, 5.45138729])
>>>
>>> np.floor(x)
array([0., 1., 9., 5., 5.])
```

■ numpy.ceil

```
numpy.ceil(x, *ufunc_args) = <ufunc 'ceil'>
```

Функция ceil() округляет к большему целому числу. Данная функция часто называется потолок числа x и обозначается как ⌈x⌉.

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Число или массив чисел, которые округлены к наибольшему целому числу.

Смотреть также: arcsinh, arccosh, tanh, tanh

Примеры

```
>>> import numpy as np
>>>
>>> np.ceil([-5.5, 5.5])
array([-5.,  6.])
>>>
>>> x = np.random.random(5)*10
>>> x
array([2.50681892, 1.08993134, 5.21603004, 1.57251841, 7.64590507])
>>>
>>> np.ceil(x)
array([3., 2., 6., 2., 8.])
```

■ numpy.trunc

```
numpy.trunc(x, *ufunc_args) = <ufunc 'trunc'>
```

Функция trunc() отбрасывает дробную часть числа.

Параметры:

x - число или подобный массиву объект

Число или последовательность чисел.

*ufunc_args - аргументы универсальной функции

Аргументы, позволяющие настроить и оптимизировать работу функции (подробнее см. универсальные функции).

Возвращает:

результат - число или массив NumPy или его подкласс

Число или массив чисел с отброшенной дробной частью.

Замечание

Данная функция доступна в NumPy начиная с версии 1.3.0

Смотреть также: rint, ceil, floor

Примеры

```
>>> import numpy as np
>>>
>>> np.trunc([-5.5, 5.5])
array([-5.,  5.])
>>>
>>> x = np.random.random(5)*10
>>> x
array([6.82408707, 6.13384365, 9.59990973, 1.45484855, 4.02585129])
>>>
>>> np.trunc(x)
array([6., 6., 9., 1., 4.])
```

Декораторы и декорирование

Функции в python являются объектами. Функции как объекты можно передавать в качестве аргумента другой функции. Функцию как объект можно возвращать из другой функции в качестве возвращаемого значения. Функцию как объект можно определить внутри другой функции.

Замыкания

Замыкание (closure) - объект, который включает в себя блок кода (например, функцию) и дополнительные переменные за его пределами. Фактически, запоминается и используется переменная, которая не является частью локальной области видимости.

Зачем нужны замыкания:

- позволяют избегать глобальных переменных,
- прячут данные (чтобы их не меняли по неаккуратности).

Для применения замыканий важно соблюдение условий:

- должна быть дочерняя (внутренняя) функция внутри другой (внешней) функции;
- дочерняя функция должна обращаться к переменной, объявленной во внешней функции;
- внешняя функция возвращает внутреннюю функцию как объект.

Код

```
# внешняя функция
def extern_example():
    x = 11
    # внутренняя функция
    def inner():
        print(f'Переменная из замыкания: {x}')

    # возвращение внутренней функции как объекта
    return inner

# вызов внешней функции, которая возвращает внутреннюю
# функцию, которая выполняется
extern_example()()

# аналогичная конструкция с применением дополнительной переменной
innerFun = extern_example()
innerFun()
```

Декоратор

Декоратор - это функция, которая позволяет изменять (дополнять) поведение декорируемой функции, не изменяя её код. Это функция, которая принимает в качестве аргумента другую функцию (декорируемую функцию) и, в зависимости от собственных аргументов, что-либо делает с этой функцией:

- возможно, обеспечивает вызов этой функции,
- получает и модифицирует результаты выполнения декорируемой функции,

- возвращает эти значения,
- в зависимости от значений собственных аргументов и результатов выполнения декорируемой функции, вызывает другие функции.

Декорируемые функции

Логика работы декоратора косвенно уже была описана в разделе про замыкания. Далее уточнение информации о замыканием с дополнением ее особенностями декораторов.

Декоратор - это обёртка над функцией (или другим объектом), которая изменяет ее поведение. Это удобно (?), так как не нужно менять исходный код (возможно, что чужой).

Схема работы декоратора:

- в качестве параметра принимается другая функция,
- возвращается замыкание,
- замыкание может использовать любое количество параметров,
- запускается код внутри замыкания,
- передаются параметры внутрь замыкания и используются в тех местах, где прописано,
- возвращается внутренняя функция с функционалом, по умолчанию дополненным декорированием.

```
# внешняя функция с аргументом
def simple_decorator(func):

    # внутренняя функция — модификатор аргумента func
    def inner():
        print('Начало работы декоратора...')
        func()
        print('Декоратор отработал!')

    # возвращение внутренней функции как объекта
    return inner

#  функция, которую задекорировали
def print_hi():
    print(f'Это функция, которую задекорировали')

print_hi() # вызов недекорированной функции

# в декоратор отправляется функция для декорирования
# изменённая (декорированная) функция принимается по ссылке
# на объект декорирования ...
print_hi = simple_decorator(print_hi)
# ... и выполняется
print_hi()
```

Таким образом, чтобы задекорировать функцию (объект), нужно передать его внутрь декоратора, а затем вызвать.

В данном примере задекорированной функции было присвоено то же имя (print_hi), потому что так принято в python, хотя можно давать любое другое имя.

Для отображения стандартного вызова внешней функции декоратора со ссылкой на декорируемую функцию в python применяется специальная конструкция:

@Имя_декоратора

Декорируемая_функция

Код из старого примера с применением новой синтаксической конструкции:

```
# внешняя функция с аргументом
def simple_decorator(func):

    # внутренняя функция — модификатор аргумента func
    def inner():
        print('Начало работы декоратора...')
        func()
        print('Декоратор отработал!')

    # возвращение внутренней функции как объекта
    return inner

@simple_decorator
#  функция, которую задекорировали
def print_hi():
    print(f'Это функция, которую задекорировали')

print_hi() # здесь выполняется уже декорированная функция
# в этом случае вызов недекорированной функции становится НЕВОЗМОЖЕН
#(синтаксические украшения ограничивают возможности программирования)
```

Здесь вызывается функция print_hi. Но на самом деле вызывается декоратор simple_decorator, которому передаётся в виде первого аргумента, целевая функция (print_hi).

■ Применение декораторов

Типичные случаи использования декораторов: оценка времени работы функции, кеширование, запуск только в определенное время, задание параметров подключения к базе данных и т.д. Существует множество известных 'стандартных' декораторов для решения конкретных задач программирования.

Так, декоратор @property, при декорировании метода area в классе Rectangle позволяет обратиться к члену area экземпляра класса Rectangle как к атрибуту.

```
class Rectangle:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @property
    def area(self):
        return self.a * self.b

rect = Rectangle(5, 6)
print(rect.area) # обращение к методу как к свойству

# 30
```

В последней строке кода к члену area экземпляра класса Rectangle можно обратиться как к атрибуту. То есть не нужно непосредственно вызывать метод area. Вместо этого при обращении к area как к атрибуту (без использования скобок, ()), соответствующий метод area вызывается неявным образом. Это возможно благодаря декоратору @property.

```
class Rectangle:
    def __init__(self, a, b):
        self.a = a
        self.b = b

#    @property
    def area(self):
        return self.a * self.b

rect = Rectangle(5, 6)
print(rect.area()) # без декоратора простое обращение к методу

# 30
```

■ Как это работает?

Размещение конструкции @property перед определением функции равносильно использованию конструкции вида area = property(area). Таким образом, property — это функция, которая принимает другую функцию в качестве аргумента и возвращает ещё одну функцию. Именно этим и занимаются декораторы (@property в том числе). В результате оказывается, что декоратор меняет поведение декорируемой функции.

■ Декоратор retry

Пусть имеется функция, которую нужно запускать повторно в том случае, если при её первом запуске происходит сбой. То есть, нужна функция (функция - декоратор, имя которого, retry, можно перевести как «повтор»), которая вызывает эту функцию один или два раза (это зависит от того, возникнет ли ошибка при первом вызове функции).

В соответствии с ранее заданным определением можно изменить код ранее описанного декоратора таким:

```
def retry(func):
    def _wrapper(*args, **kwargs):
        try:
            func(*args, **kwargs)
        except:
            time.sleep(1)
            func(*args, **kwargs)
    return _wrapper

@retry
def might_fail():
    print("might_fail")
    raise Exception

might_fail()
```

Здесь декоратор носит имя retry. Он принимает в виде аргумента (func) любую функцию.

Внутри декоратора определяется внутренняя функция (_wrapper) и обеспечивается возврат этой функции. Это совершенно корректная синтаксическая конструкция, следствием применения которой является то полезное обстоятельство, что функция _wrapper видна лишь внутри пространства имён декоратора retry.

В этом примере декорируется функция might_fail() с использованием конструкции, которая имеет вид @retry. После имени декоратора нет круглых скобок. В результате получается, что когда, как обычно, вызывается функция might_fail(), на самом деле, вызывается декоратор retry, которому передаётся, в виде первого аргумента, целевая функция (might_fail).

Получается, что, в общей сложности, тут поработали три функции:

- retry
- _wrapper
- might_fail

При выполнении функции might_fail всё равно всегда случается исключение (она так запрограммирована). Но retry и _wrapper (декоратор и внутренняя функция пытаются этому помешать) или оттянуть момент возникновения исключения на время, заданное методом time.sleep(1).

Аргументы декоратора

В некоторых случаях нужно, чтобы кроме ссылки на декорируемую функцию декораторы принимали бы дополнительные аргументы. Например, может понадобиться, чтобы декоратор retry принимал бы число, задающее количество попыток запуска декорируемой функции.

Но при этом декоратор обязательно должен принимать декорируемую функцию в качестве первого аргумента. Не надо забывать и о том, что вызывать декоратор при декорировании функции не надо. То есть — о том, что перед определением функции используется конструкция @retry, а не @retry(). Таким образом:

- Декоратор — это всего лишь функция (которая, в качестве аргумента, принимает другую функцию).
- Декораторами пользуются, помещая их имя со знаком @ перед определением функции, а не вызывая их.
- Следовательно, можно ввести в код четвёртую функцию, которая принимает параметр, с помощью которого настраивается поведение декоратора, и возвращается функция, которая и является декоратором (то есть — принимает в качестве аргумента другую функцию).

Предлагается к тестированию следующая конструкция:

```
def retry(max_retries):
    def retry_decorator(func):
        def _wrapper(*args, **kwargs):
            for _ in range(max_retries):
                try:
                    func(*args, **kwargs)
                except:
                    time.sleep(1)
```

```
        return _wrapper
    return retry_decorator


@retry(2)
def might_fail():
    print("might_fail")
    raise Exception


might_fail()
```

Разбор этого кода:

- На первом уровне тут имеется функция retry.
- Функция retry принимает произвольный аргумент (в нашем случае — max_retries) и возвращает другую функцию — retry_decorator.
- Функция retry_decorator — это и есть реальный декоратор.
- Функция _wrapper работает так же, как и прежде (только теперь она руководствуется сведениями о максимальном количестве перезапусков декорированной функции).

Использование этого кода

Функция might_fail теперь декорируется с помощью вызова функции вида @retry(2).

Вызов retry(2) приводит к тому, что вызывается функция retry, которая и возвращает реальный декоратор.

В итоге функция might_fail декорируется с помощью retry_decorator, так как именно эта функция представляет собой результат вызова функции retry(2).

Далее представляется пример декорирования множества декорируемых функций функцией-декоратором.

```
# декорируемые функции ================================================
def final_function_0():
    print('this is final function 0')
def final_function_1():
    print('this is final function 1')
def final_function_XXX():
    print('this is final function XXX')
# =====================================================================
# декоратор ===========================================================
# декорируемая функция как аргумент декоратора ========================
def function_decorator(function_to_decorate):
    print('...function_decorator is here ===>')
    def wrapper_around_the_original_function():
        print('this is the code before the function_to_decorate call')
        function_to_decorate()
        print('this is the code after the function_to_decorate call')
    print('===> function_decorator is here...')
    return wrapper_around_the_original_function
# =====================================================================
# функции function_XYZ, function_KLM декорируются во время объявления с
# применением синтаксиса декораторов ==================================


@function_decorator
```

```
def function_XYZ():
    print('this is function XYZ')
@function_decorator
def function_KLM():
    print('this is function KLM')
# ========================================================================
if __name__ == '__main__':
    # вызываются функции, предназначенные для декорирования ==============
    final_function_0()
    final_function_1()
    # ====================================================================
    # вызывается функция - декоратор. Возвращается ссылка на =============
    # внутреннюю функцию. Ссылка обеспечивает вызов декорируемой функции
    final_function_decorated = function_decorator(final_function_0)
    # ====================================================================
    # вызов внутренней функции. Запускается final_function_0 =============
    final_function_decorated()
    final_function_decorated = function_decorator(final_function_1)
    # ====================================================================
    # вызов внутренней функции. Запускается final_function_1 =============
    final_function_decorated()
    # зарядили функцию для декорации final_function_XXX ==================
    final_function_decorated = function_decorator(final_function_XXX)
    # ====================================================================
    # При вызове внутренней функции запускается декорированная функция
    # final_function_XXX ================================================
    print('*0**********************')
    final_function_decorated()
    print('*1**********************')
    final_function_decorated()
    print('*2**********************')
    final_function_decorated()
    print('*3**********************')
    final_function_decorated()
# ========================================================================
# запуск декорированных функций ========================================
    function_XYZ()
    function_KLM()
# ========================================================================
```

■ Декоратор retry

Написание «собственных» декораторов — для понимания принципов их работы.

Пусть имеется функция, которую надо запустить повторно в том случае, если при её первом запуске произойдёт сбой. То есть — нам нужна функция (декоратор, имя которого, retry), которая вызывает данную функцию один или два раза (это зависит от того, возникнет ли ошибка при первом вызове функции).

В соответствии с ранее приведённым определением можно сделать код декоратора таким:

```
def retry(func):
    def _wrapper(*args, **kwargs):
        try:
            func(*args, **kwargs)
        except:
```

```
            time.sleep(1)
            func(*args, **kwargs)
    return _wrapper

@retry
def might_fail():
    print("might_fail")
    raise Exception

might_fail()
```

Декоратор носит имя retry. Он принимает в качестве аргумента (func) любую функцию. Внутри декоратора определяется вложенная функция (_wrapper). Декоратор retry осуществляет возврат этой функции.

■ Напоминание

Имеет место объявление одной функции внутри другой функции. Это совершенно корректная синтаксическая конструкция, следствием применения которой является тот , что функция _wrapper видна лишь внутри пространства имён декоратора retry.

В этом примере декорируется функция might_fail() с использованием конструкции, которая выглядит @retry. После имени декоратора нет круглых скобок. В результате получается, что когда вызывается функция might_fail(), на самом деле, вызывается декоратор retry, которому передаётся, в виде первого аргумента, целевая функция (might_fail).

Получается, что, вызов отрабатывают в общей сложности три функции:

retry

_wrapper

might_fail

В некоторых случаях нужно, чтобы декораторы принимали бы дополнительные аргументы. Например, может понадобиться, чтобы декоратор retry принимал бы число, задающее количество попыток запуска декорируемой функции. Но декоратор обязательно должен принимать декорируемую функцию в качестве первого аргумента. Также важно, что при декорировании функции не надо вызывать декоратор. То есть, что перед определением функции используется конструкция @retry, а не @retry(). Итог:

Декоратор — это всего лишь функция (которая, в качестве аргумента, принимает другую функцию).

Декораторами пользуются, помещая их имя со знаком @ перед определением функции, а не вызывая их.

Следовательно, можно ввести в код четвёртую функцию, которая принимает параметр, с помощью которого надо настраивать поведение декоратора, и возвращает функцию, которая и является декоратором (то есть — принимает в качестве аргумента другую функцию).

Пример для иллюстрации:

```
def retry(max_retries):
    def retry_decorator(func):
        def _wrapper(*args, **kwargs):
            for _ in range(max_retries):
                try:
                    func(*args, **kwargs)
                except:
                    time.sleep(1)
        return _wrapper
    return retry_decorator


@retry(2)
def might_fail():
    print("might_fail")
    raise Exception


might_fail()
```

Разбор кода:

На первом уровне имеется функция retry.

Функция retry принимает произвольный аргумент (в нашем случае — max_retries) и возвращает другую функцию — retry_decorator.

Функция retry_decorator — это и есть реальный декоратор.

Функция _wrapper работает так же, как и прежде (только теперь она руководствуется сведениями о максимальном количестве перезапусков декорированной функции).

О коде нового декоратора больше сказать нечего. Теперь о его использовании:

Функция might_fail теперь декорируется с помощью вызова функции вида @retry(2).

Вызов retry(2) приводит к тому, что вызывается функция retry, которая и возвращает реальный декоратор.

В итоге функция might_fail декорируется с помощью retry_decorator, так как именно эта функция представляет собой результат вызова функции retry(2).

■ Множественное декорирование функции

Далее представлен пример декорирования ОДНОЙ функции НЕСКОЛЬКИМИ декораторами. При таком способе декорирования важен порядок применения декораторов к функции (порядок декорирования).

```
# несколько декораторов для одной функции. Важен порядок декорирования:
def final_function_XXX():
    print('this is final function XXX')
def decorator_0(function_to_decorate):
    def wrapper():
        print('function_decorator_0 is here')
        function_to_decorate()
        print('function_decorator_0 was here---0---')
    return wrapper
def decorator_1(function_to_decorate):
    def wrapper():
        print('function_decorator_1 is here')
        function_to_decorate()
        print('function_decorator_1 was here---1---')
    return wrapper
# декорирование с применением синтаксиса декораторов ====================
@decorator_0
@decorator_1
def final_function_KLM():
    print('this is final function KLM')
# ========================================================================
if __name__ == '__main__':
    final_function_XXX() # запуск декорируемой функции
    final_function_XXX = decorator_0(decorator_1(final_function_XXX))
    final_function_XXX() # запуск декорированной функции, заряженной
                         # последовательно двумя декораторами
    print('=========================')
    final_function_KLM() # запуск декорированной функции с применением
                         # синтаксиса декораторов ========================
    print('=========================')
```

■ Передача аргументов в декорируемую функцию

Передача аргументов декорируемой функции декоратором производится следующим образом. Вызывается внешняя функция, которая возвращает ссылку на внутреннюю функцию. Внутренняя функция обеспечивает вызов декорируемой функции и передачу ей аргументов. В следующем примере при вызове декорируемой функции ей передаётся два аргумента.

```
# Передача аргументов декорируемой функции.
# декорируемая функция. При вызове Требует два аргумента ================
def final_function_000(arg1, arg2):
    print('this is final function 000({0},{1})'.format(arg1, arg2))
# ========================================================================
# внешняя функция. возвращает ссылку на внутреннюю функцию.
def a_decorator_passing_arguments(function_to_decorate):
    # Внутренняя функция обеспечивает вызов декорируемой функции
    # и передачу ей аргументов
    def a_wrapper_accepting_arguments(arg1, arg2):
        print("аргументы в декорируемую функцию:", arg1, arg2)
        function_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments
@a_decorator_passing_arguments
def final_function_XXX(arg1, arg2):
```

```
    print('this is final function XXX({0},{1})'.format(arg1, arg2))
if __name__ == '__main__':
    # Вызывается внешняя функция, которая возвращает ссылку
    # на внутреннюю функцию. Внутренняя функция обеспечивает вызов
    # декорируемой функции и передачу ей аргументов
    final_function_decorated =
                  a_decorator_passing_arguments(final_function_000)
    final_function_decorated(512, 1024)
    final_function_XXX('=quartz perfect=', 795)
```

■ ООП: декорирование методов

Нестатические методы python — это функции с дополнительным обязательным аргументом self, который представляет собой ссылку на объект, вызывающий данный метод. И это означает, что декораторы для методов можно объявлять так же, как и для обычных функций с непустым списком аргументов.

```
def method_decorator(method_to_decorate):
    def wrapper(self, value):
        value *= 3
        return method_to_decorate(self, value)
    return wrapper
class superCar:
    def __init__(self, value):
        self.capacity = value # при создании объекта атрибуту capacity
                              # присваивается значение аргумента value

    @method_decorator
    def sayRealCapacity(self, value):
        print('the realCapacity is {0}'.format(self.capacity * value))
if __name__ == '__main__':
    the_car = superCar(125)
    the_car.sayRealCapacity(3)  # значение, возвращаемое методом
    # sayRealCapacity со значением аргумента = 3, вычисляется следующим
    # образом: 3 * 3 * 125 = 1125
    #          3 * value * self.capacity = 1125
```

■ Декорирование с распаковкой аргументов

Декоратор, который можно применить к любой функции или методу, объявляется с распаковкой аргументов.

```
# универсальный декоратор, который принимает произвольный
# список аргументов в составе кортежа *args и словаря **kwargs
# декорируемая функция как аргумент декоратора ============================
def universal_decorator_passing_arbitrary_arguments(function_to_decorate):
    # Внутренняя функция обеспечивает вызов декорируемой функции
    # и передачу ей аргументов
    def wrapper_accepting_arbitrary_arguments(*args, **kwargs):
        print("было ли что-либо передано декорируемой функции ?")
        if len(args) != 0 or len(kwargs) != 0:
            print(args)
            print(kwargs)
        else:
            print('args && kwargs are empty')
        function_to_decorate(*args, **kwargs)
    return wrapper_accepting_arbitrary_arguments
# универсальный декоратор декорирует функцию без аргументов
```

```
@universal_decorator_passing_arbitrary_arguments
def function_with_no_argument():
    print("Python is cool, no argument here.")
# универсальный декоратор декорирует функцию с аргументами a, b, c
@universal_decorator_passing_arbitrary_arguments
def function_with_arguments(a, b, c):
    print(a, b, c)
# универсальный декоратор декорирует функцию с аргументами a, b, c
# и именованным аргументом answer
@universal_decorator_passing_arbitrary_arguments
def function_with_named_arguments(a, b, c, answer):
    print("делятся ли {}, {} и {} на 17? {}".format(a, b, c, answer))
class Observer:
    def __init__(self):
        self.age = 31
    # универсальный декоратор декорирует метод со значением по умолчанию
    @universal_decorator_passing_arbitrary_arguments
    def sayTheAge(self, lie=-3):  # Теперь можно указать
                                  # значение по умолчанию
        print("Остаётся {} лет".format(self.age + lie))
# вызовы декорируемых функций и метода ================================
function_with_no_argument()
function_with_arguments(1, 2, 3)
function_with_named_arguments("25", "137", "64", answer="А кто его знает!")
obs = Observer()
obs.sayTheAge()
```

■ Декораторы с аргументами

Описание процедуры объявления декоратора с аргументами. В процессе декорирования ранее объявленной функции decorated_function функция, которая запускает процесс декорирования — decorator_maker — возвращает ссылку на функцию master_decorator. В результате обращения по этой ссылке возвращается декорированная функция по ссылке decorated_function. Возвращаемый код включает код контейнера wrapper для декорируемой функции, из которого непосредственно и вызывается декорируемая функция.

Таким образом, при вызове декорированной функции по ссылке decorated_function — уже не исходная декорируемая функция. Здесь подключается контейнер wrapper, который выполняет предварительную работу, запускает исходную декорируемую функцию, перехватывает её возвращаемое значение и дополнительно его модифицирует. Результирующее значение, возвращаемое декорированной функцией, распечатывается перед завершением приложения.

Можно передавать в декоратор аргументы. В этом случае добавляется еще один уровень абстракции, то есть, ещё одна функция-обертка. Таким образом, аргумент передаётся непосредственно декоратору. После этого функция, которая была возвращена декоратором, используется для декорации.

```
# ======================================================================

def decorator_with_args(name):
    print('> decorator_with_args:', name)
    def real_decorator(func):
        print('>> сам декоратор', func.__name__)
        def decorated(*args, **kwargs):
            print('>>> перед функцие', func.__name__)
            ret = func(*args, **kwargs)
            print('>>> после функции', func.__name__)
            return ret
```

```
        return decorated
    return real_decorator
# ========================================================================
@decorator_with_args('test')
def add(a, b):
    print('>>>> функция add')
    return a + b
# ========================================================================
if __name__ == '__main__':
    print('старт программы')
    r = add(10, 10)
    print(r)
    print('конец программы')
```

Ещё пример:

```
# Декораторы с аргументами
# Объявление декоратора с аргументами:
# ========================================================================
def decorated_function():
    print("===== Это декорируемая функция. =====")
    return ('it was decorated function')
# ========================================================================
# декоратор ==============================================================
def decorator_maker():
    print("decorator_maker: создание декоратора.")
    print("decorator_maker вызывается при создании декоратора.")
    def master_decorator(function):
        print("master_decorator! Вызывается при декорировании функции.")
        def wrapper():
            print()
            print ("wrapper: контейнер для декорируемой функции.\n"
                   "Вызывается каждый раз при вызове декорированной функции")
            ret = function()
            xstr = '|'.join(ret)
            return (xstr)
        print("Возвращается обёрнутая функция.")
        return wrapper
    print("decorator_maker: возвращается master_decorator.")
    return master_decorator
# ========================================================================
@decorator_maker()
def decorated_function_X():
    num = 125
    print("===== Это декорируемая функция X...{0}... =====".format(num))
    val = num*num
    return str(val)
# ========================================================================
if __name__ == '__main__':
    # Создание мастера-декоратора. ======================================
    # Происходит в результате вызова декоратора
    mstr_decorator = decorator_maker()
    # ====================================================================
    # ====================================================================
    # Декорирование функции: master_decorator возвращает
    # decorated_function, который включает код контейнера для
    # декорируемой функции, из которого и вызывается декорируемая функция.
    decorated_function = mstr_decorator(decorated_function)
```

```
    # ========================================================================
    # ========================================================================
    # Вызов декорированной функции. decorated_function - это уже
    # не исходная декорируемая функция! Здесь подключается контейнер
    # wrapper, который выполняет предварительную работу,
    # запускает исходную декорируемую функцию, перехватывает её
    # возвращаемое значение и дополнительно его модифицирует.
    # Результирующее значение распечатывается перед завершением приложения.
    result = decorated_function()
    # ========================================================================
    print(result)
    print('========================')
    result = decorated_function_X()
    print(result)
    print('========================')
```

Далее приведён результат работы этого приложения

decorator_maker: создание декоратора.

decorator_maker вызывается при создании декоратора.

decorator_maker: возвращается master_decorator.

master_decorator! Вызывается при декорировании функции.

Возвращается обёрнутая функция.

decorator_maker: создание декоратора.

decorator_maker вызывается при создании декоратора.

decorator_maker: возвращается master_decorator.

master_decorator! Вызывается при декорировании функции.

Возвращается обёрнутая функция.

wrapper: контейнер для декорируемой функции.

Вызывается каждый раз при вызове декорированной функции

===== Это декорируемая функция. =====

i|t| |w|a|s| |d|e|c|o|r|a|t|e|d| |f|u|n|c|t|i|o|n

========================

wrapper: контейнер для декорируемой функции.

Вызывается каждый раз при вызове декорированной функции

===== Это декорируемая функция X...125... =====

1|5|6|2|5

========================

Process finished with exit code 0

■ Аргументы декораторов

Можно создавать декораторы "на лету". Для этого используются функции, при вызове которы им как обычным функциям можно передавать произвольные аргументы.

В случае необходимости можно использовать и распаковку аргументов через *args и **kwargs .

```
def decorator_maker_with_arguments(deco_arg1, deco_arg2):
    print("Здесь создатся декораторы. И для этого применяются аргументы:",
          deco_arg1, deco_arg2)
    def x_decorator(func):
        print("это декоратор. Он может получать аргументы:",
              deco_arg1, deco_arg2)
        # Аргументы декораторов и аргументы функций - это разные вещи!
        def wrapped(fun_arg1, fun_arg2):
            print ("Это обёртка вокруг декорируемой функции.\n"
                   "Она имет доступ ко всем аргументам\n"
                   "\t- как декоратора: {0} {1}\n"
                   "\t- так и функции: {2} {3}\n"
                   "Обёртка может передать аргументы декоратора {0}, {1}\n"
                   "и функции {2}, {3} дальше\n"
                   .format(deco_arg1, deco_arg2, fun_arg1, fun_arg2))
            return func(fun_arg1, fun_arg2)
        return wrapped
    return x_decorator
@decorator_maker_with_arguments("01234", "56789")
def decorated_function_with_arguments(fun_arg1, fun_arg2):
    print ("Это декорируемая функция, которая знает только о своих
аргументах: {0}"
           " {1}".format(fun_arg1, fun_arg2))
decorated_function_with_arguments("ABCDE", "FGHIJ")
```

Здесь создатся декораторы. И для этого применяются аргументы: 01234 56789

это декоратор. Он может получать аргументы: 01234 56789

Это обёртка вокруг декорируемой функции.

Она имет доступ ко всем аргументам

- как декоратора: 01234 56789

- так и функции: ABCDE FGHIJ

Обёртка может передать нужные аргументы декоратора 01234, 56789

и функции ABCDE, FGHIJ дальше

Это декорируемая функция, которая знает только о своих аргументах: ABCDE FGHIJ

Process finished with exit code 0

■ Особенности работы с декораторами

- Декораторы замедляют вызов функции.

- Очень сложно "раздекорировать" функцию (да и зачем ???). Если функция декорирована — это лучше не отменять.
- Декораторы оборачивают функции и это может затруднить отладку.

Последняя проблема частично решена добавлением в модуле functools функции functools.wraps, копирующей всю информацию об оборачиваемой функции (её имя, из какого она модуля, её документацию и т.п.) в функцию-обёртку.

Декораторы могут быть использованы для расширения возможностей функций из сторонних библиотек (код которых невозможно изменить), или для упрощения отладки (чтобы не изменять исходный отлаживаемый код).

Также полезно использовать декораторы для расширения различных функций одним и тем же кодом, без повторного его переписывания.

Класс - декоратор

Добавление метода __call__ в объявлении класса превращает его в вызываемый объект (функция - это вызываемый объект). А так как декоратор — это функция, то есть, вызываемый объект, с помощью метода __call__ класс можно превратить в декоратор.

Если функция, которую требуется декорировать, должна получать аргументы, то для этого декоратор должен вернуть функцию с той же сигнатурой (списком аргументов), что и у декорируемой функции.

В случае с классом соответствующая сигнатура добавляется в метод __call__.

```
class xDecorator:
    def __init__(self, funct):
        print('xDecorator, method __init__')
        self.funct = funct  # На этапе инициализации объект получает
                            # ссылку на функцию funct
    # В случае с классом
    # соответствующая сигнатура добавляется в метод __call__.
    def __call__(self, *args):
        print('> перед вызовом класса...', self.funct.__name__)
        self.funct(*args)
        print('> после вызова класса')
# ====================================================================
@xDecorator
def external_funct(*args):
    print('===== external_funct =====')
    i = 0
    # множество значений, передаваемых функции, представляются в виде
    # кортежа, который состоит из одного элемента: списка значений.
    arguments = args[0]   # из аргумента - кортежа извлекается список
    for a in arguments:   # перебор элементов списка
        print('{0} ... {1}'.format(i, a))
        i += 1
# ====================================================================
# ====================================================================
if __name__ == '__main__':
    args = [1, 2.05, 'qwerty', 0]
    print('--> start')
    external_funct(args)
    print('end -->')
```

■ Аргументы в классах-декораторах

В классах-декораторах выполняются аналогичные настройки. В этом случае конструктор класса получает все аргументы декоратора. Метод (магический метод) __call__ должен возвращать функцию-обертку, которая будет выполнять декорируемую функцию.

Ранее в качестве аргумента конструктору передавалась ссылка на функцию. В этой версии объявления декоратора передаётся аргумент.

Метод __call__ получает ссылку на декорируемую функцию, функция-оболочка (wrapper) объявляется с той же сигнатурой (списком аргументов), что и декорируемая функция. Она получает аргументы, соответственно предназначенные декорируемой функции.

Аргумент, передаваемый конструктору класса, функции-оболочке (wrapper) и декорируемой функции в этой версии декорирования оказывается не известен.

```
# В этом случае конструктор класса получает все аргументы декоратора.
# Метод __call__ должен возвращать функцию-обертку, которая будет
# выполнять декорируемую функцию.
class DecoratorArgs:
    # ранее в качестве аргумента конструктору передавалась
    # ссылка на функцию. В этой версии объявления передаётся
    # аргумент. Это строка.
    def __init__(self, xstr):
        print('> Декоратор с аргументами __init__:', xstr)
        self.xstr = xstr
    def __call__(self, funct):
        # метод __call__ получает ссылку на декорируемую функцию
        # функция-оболочка (wrapper) объявляется с той же сигнатурой
        # (списком аргументов), что и декорируемая функция.
        # Она получает аргументы, соответственно предназначенные
        # декорируемой функции.
        z = self.xstr
        print(z)
        # Аргумент, передаваемый конструктору класса, функции-оболочке
        # (wrapper) и декорируемой функции в этой версии декорирования
        # оказывается не известен.
        def wrapper(a, b):
            print('>>> до обернутой функции. Начало декорирования')
            funct(a, b)
            print('>>> после обернутой функции. Конец декорирования')
        return wrapper
@DecoratorArgs("xxxXXXxxx")
def add(a, b):
    print('функция add:', a, b)
print('>> старт')
add(10, 20)
print('>> конец')
```

■

Исключения и обработка исключений

Этот раздел содержит основные сведения об исключениях - языковой конструкции, предназначенной для формального описания ошибок времени выполнения и способов их "корректного" исправления непосредственно в ходе выполнения приложения, которая широко применяется в современных языках программирования.

Tkinter: общее представление

Tkinter – это пакет для Python, который предназначен для работы с библиотекой Tk. Графическая кроссплатформенная библиотека на основе средств Tk. Свободное ПО, включено в стандартную библиотеку языка программирования Python. В состав Tkinter входит множество компонентов. Библиотека Tk содержит компоненты графического интерфейса пользователя (graphical user interface – GUI. Библиотека написана на языке Tcl.).

Графический интерфейс

Под графическим интерфейсом пользователя (GUI) подразумеваются окна, кнопки, текстовые поля для ввода, скроллеры, списки, радиокнопки, флажки и прочие элементы, которые появляются на экране при открытии того или иного приложения.

Все эти элементы интерфейса называются виджетами (widgets). Через них обеспечивается взаимодействие с программой и управление этой программой.

Приложения, которые создаются для конечного пользователя, имеют GUI.

Tk выбран для Python по умолчанию. Установочный файл интерпретатора Python обычно уже включает пакет tkinter в составе стандартной библиотеки.

Tkinter можно представить как переводчик с языка Python на язык Tcl. Программа пишется на Python, а код модуля tkinter переводит инструкции с Python на язык Tcl, который понимает библиотека Tk.

Библиотеку Tkinter надо импортировать. Стандартные способы импорта модуля Tkinter в Python:

```
import tkinter
from tkinter import *
import tkinter as tk
```

Также можно импортировать отдельные классы, но это делают редко. Обычно хватает выражения

```
from tkinter import *
```

Если нужно узнать установленную версию Tk, это можно сделать через константу TkVersion:

```
>>> from tkinter import *
>>> TkVersion
8.6
```

Ошибки

Разработка и выполнение приложения может сопровождаться различными ошибками, к которым можно отнести:

- Синтаксические – возникают при разработке кода из-за синтаксических(!) ошибок;
- Логические – являются результатом ошибок разрабатываемого алгоритма в приложении;

- Исключения – возникают при выполнении приложения в результате некорректных действий пользователя или системы.

Синтаксические ошибки

Являются результатом нарушения правил языка программирования, например, пропуска круглой, прямоугольной или фигурной скобки в предполагаемом синтаксисом языка месте разрабатываемого кода. Обнаруживаются компилятором или интерпретатором, которые сообщением (обычно стандартной формы) уведомляют разработчика кода (программиста) о проблеме в создаваемом коде.

Логические ошибки

Как правило, более сложные в выявлении, так как не "отлавливаются" ни на этапе компиляции (если таковая существует), ни во время выполнения программного кода (возможно, интерпретатором). Обычно они вызваны ошибками в логике реализуемого алгоритма, из-за чего достижение предусматриваемого результата оказывается невозможным.

Исключения

Ещё один вид ошибок, который проявляется в зависимости от наличия обстоятельств, влияющих на ход выполнения программы. Примером может быть ввод некорректного значения, либо отсутствие файла, необходимого для корректного выполнения приложения. Как правило, исключения проявляются во время выполнения приложения.

Исключения: преимущества применения

Исключения представляют собой самостоятельный тип данных, при помощи которого приложение может стандартным образом сообщать (пользователю) о различных ошибках. Обработка исключений позволяет менять сценарий выполнения приложения с учётом различных обстоятельств, так или иначе нарушающих его нормальную работу. Для управления исключениями в языке программирования python предусмотрены специальные синтаксические конструкции.

Также в языке существует библиотека готовых исключений, а также реализована возможность создавать собственные исключения.

Программа, которая способна корректно обработать возникающие исключения, завершается естественным образом, без прерываний заданного алгоритма.

Например, отсутствие файла при выполнении приложения провоцирует исключительную ситуацию, которая приводит к аварийному завершению работы программы и выводу ошибки на экран.

Пример приложения, НЕ способного "стандартным" образом реагировать на отсутствие файла при попытке его открытия или сохранения:

```
# ======================================================================

from tkinter import *

root = Tk()

print("Program started")
print("Opening file...")
f = open("data.txt")
print("Program finished")

if __name__ == '__main__':
    root.mainloop()
```

```
# ========================================================================
```

При отсутствии файла "data.txt" в ходе выполнения приложения будет выведено сообщение о возникновении соответствующего исключения:

```
Traceback (most recent call last):
  File "C:/PythonDrom/SeismicModels/SeisModel04/xGui21.py", line 57, in
<module>
Program started
Opening file...
    f = open("data.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'data.txt'
Process finished with exit code 1
```

Данное исключение предусмотрено разработчиками среды программирования python (язык, интерпретатор, библиотечные модули).

Разработчики приложения НЕ предусмотрели в приложении реакцию на отсутствие файла, что и привело к возникновению исключения.

В этом случае отображается имя файла, номер строки в коде, где было выброшено исключение FileNotFoundError (FileNotFoundError: [Errno 2] No such file or directory: 'data.txt').

■ Перехват исключений

Для того чтобы исключение не приводило к внезапному завершению приложения, в языке программирования реализуется механизм, который способен предотвращать непредвиденные (НО предусмотренные) ситуации.

С помощью специальных конструкций (try:...except:...) языка программирования в приложении описываются 'критические' фрагменты кода и варианты реагирования на возникающие исключения. Это помогает избежать сбоев при выполнении приложения.

```
# ========================================================================

from tkinter import *

root = Tk()

print("Program started")

try:
    print("Opening file...")
    f = open("data.txt")
except:
    print('file not found!')

print("Program finished")

if __name__ == '__main__':
    root.mainloop()

# ========================================================================
```

Блок try содержит "опасный код", который способнен привести к ошибке (отсутствие нужного файла).

Блок except содержит инструкции, которые приложению следует выполнить в случае возникнувшей проблемы. Теперь если требуемый файл не был найден, приложение не будет аварийно завершено.

■ except: несколько блоков

Блоков except может быть несколько, в зависимости от того, какой тип исключения нужно обработать в приложении. Как правило, сначала обрабатываются более частные случаи, затем общие.

```
# ================================================================================

from tkinter import *

root = Tk()

print("Program started")

try:
    print("Opening file...")
    f = open("data.txt")
except FileNotFoundError:
    print('file not found!')
except Exception:
    print('something gone wrong!')

print("Program finished")

if __name__ == '__main__':
    root.mainloop()

# ================================================================================
```

■ Вложенные блоки и else

Для более гибкого управления исключениями блоки try:...except:... могут быть вложенными. В следующем примере демонстрируется попытка открыть текстовый файл и записать в него некую строку. Для каждой цели используется отдельный блок try.

Также в данном примере используется конструкция else, которая выполняется в случае, если в коде не произошло исключений.

В данном случае else сработает при успешном выполнении операции write. По умолчанию файл открывается на чтение в текстовом режиме. Поэтому при открытии файла используется режим "w". При этом файл открывается на запись.

Если файла не было — создается новый, если файл был — он перезаписывается.

```
# ================================================================================

from tkinter import *

root = Tk()

print("Program started")

# внешний блок try - except ==================================================
try:
    print("Opening file...")
```

```
    f = open("data.txt", "w")

    # вложенный блок try - except - else ================================
    try:
        print("writing to file...")
        f.write("----- Hello World -----")
    except Exception:
        print('something gone wrong!')
    else:
        print("Success!")
    # ===================================================================

except FileNotFoundError:
    print('file not found!')
# ========================================================================

print("Program finished")

if __name__ == '__main__':
    root.mainloop()

# ========================================================================
```

■ finally

Бывают ситуации, когда требуется совершить некие важные действия независимо от того, было ли в приложении вызвано исключение или нет. Для этого используется блок finally (ещё один элемент системы обработки исключений), содержащий набор инструкций, которые должны быть выполнены в любом случае, независимо от ситуации, возникающей в ходе выполнения приложения. Следующий пример улучшает работу предыдущей программы, добавляя в нее возможность закрытия текстового файла:

```
# ========================================================================

from tkinter import *

root = Tk()

print("Program started")

# внешний блок try - except ================================================
try:
    print("Opening file...")
    f = open("data.txt", "w")

    # вложенный блок try - except - else ================================
    try:
        print("writing to file...")
        f.write("----- Hello World -----")
    except Exception:
        print('something gone wrong!')
    else:
        print("Success!")
    finally:
        f.close()
        print("Closing file...")
```

```
    # ========================================================================

except FileNotFoundError:
    print('file not found!')
# ============================================================================

print("Program finished")


if __name__ == '__main__':
    root.mainloop()

# ============================================================================
```

■ Конструкция with - as

Описываемый ранее подход к работе с исключениями может показаться слишком сложным, так как код, который его реализует, выглядит громоздким (?). Специально для этого случая (для "упрощения" кода) в языке python существует конструкция with - as, которая позволяет автоматизировать некоторые стандартные методы (например, закрытие файла) при работе с соответствующими объектами. Это позволяет сократить длину кода:

```
# ============================================================================

from tkinter import *

root = Tk()

print("Program started")

try:
    print("Opening file...")
    # конструкция with - as
    with open("data.txt", "w") as f:
        f.write("----- Hello World -----")
except Exception:
    print('something gone wrong!')
    print('file not found!')

print("Program finished")

if __name__ == '__main__':
    root.mainloop()

# ============================================================================
```

■ Пользовательские исключения

Как правило, в корректно написанном приложении, исключения автоматически вызываются в необходимых ситуациях. При этом в python существует возможность запускать их самостоятельно. Для этого в python применяется оператор raise. Следом за ним объявляется и создаётся новый объект типа Exception, с которым в дальнейшем можно работать при помощи уже известных конструкций try:...except:..., как в следующем примере:

```
# ============================================================================

from tkinter import *
```

```
root = Tk()

print("Program started")

try:
    raise Exception('User Exception !')
except Exception as e: # эта конструкция позволяет распечатать текст
                       # исключения (в данном случае
                       # Exception - 'User Exception !')
    print(str(e))

print("Program finished")


if __name__ == '__main__':
    root.mainloop()

# ========================================================================
```

Чтобы описать собственный (пользовательский) тип исключения, нужно создать новый класс, унаследованный от базового типа Exception. Это позволит запускать особые виды исключений в ситуациях, когда поведение пользователя не соответствует алгоритму программы. В конструкторе Exception указывается текст исключения. После того, как исключение сработало и было перехвачено, его можно получить с помощью str.
В следующем коде представлен процесс генерации исключения NegativeAge, которое не позволяет ввести отрицательный возраст:

```
# ========================================================================

from tkinter import *

# объявление пользовательского исключения ================================
class NegativeAge(Exception):
    pass
# ========================================================================

root = Tk()

print("Program started")

try:
    age = int(input("Tnter yor age: "))
    # в случае ввода некорректного значения age - генерация ==============
    # пользовательского исключения
    if age < 0:
    # ====================================================================
        raise NegativeAge("Exception: Negative age!")
    # ====================================================================
    else:
        # если значение введено правильно - приложение завершает работу
        print("Success!")
except NegativeAge as e:
    # в противном случае перехватывается исключение NegativeAge.
    # Блок его обработки предусматривает распечатку его текстового
    # представления ("Exception: Negative age!")
    print(str(e))

print("Program finished")
```

```
if __name__ == '__main__':
    root.mainloop()

# ======================================================================
```

■ Иерархия исключений

В языке программирования python существует строгая иерархия исключений. "Вершиной" является BaseException, включающий в себя все существующие разновидности исключений:

- SystemExit – возникает при выходе из программы с помощью sys.exit;
- KeyboardInterrupt – указывает на прерывание программы пользователем;
- GeneratorExit – появляется при вызове метода close для объекта generator;
- Exception – представляет множество обычных несистемных исключений.

■ Класс Exception. Несистемные исключения

В следующей таблице приведены несистемные исключения, которые содержит класс Exception.

Название	Описание
ArithmeticError	Порождается арифметическими ошибками (операции с плавающей точкой, переполнение числовой переменной, деление на ноль)
AssertionError	Возникает при ложном выражении в функции assert
AttributeError	Появляется в случаях, когда нужный атрибут объекта отсутствует
BufferError	Указывает на невозможность выполнения буферной операции
EOFError	Проявляется, когда функция не смогла прочитать конец файла
ImportError	Сообщает о неудачном импорте модуля либо атрибута
LookupError	Информирует про недействительный индекс или ключ в массиве
MemoryError	Возникает в ситуации, когда доступной памяти недостаточно
NameError	Указывает на ошибку при поиске переменной с нужным именем
NotImplementedError	Предупреждает о том, что абстрактные методы класса должны быть обязательно переопределены в классах-наследниках
OSError	Включает в себя системные ошибки (отсутствие доступа к нужному файлу или директории, проблемы с поиском процессов)
ReferenceError	Порождается попыткой доступа к атрибуту со 'слабой' ссылкой
RuntimeError	Сообщает об исключении, которое не классифицируется
StopIteration	Возникает во время выполнения функции next при отсутствии элементов

SyntaxError	Представляет собой совокупность синтаксических ошибок
SystemError	Порождается внутренними ошибками системы
TypeError	Указывает на то, что операция не может быть выполнена с объектом
UnicodeError	Сообщает о неправильной кодировке символов в программе
ValueError	Возникает при получении некорректного значения для переменной
Warning	Обозначает предупреждение

Магические методы

Этот раздел основан на руководстве от Rafe Kettler 1.17 версии.

Содержание раздела:

Введение

Магические методы применяются в объектно-ориентированном python. Это специальные методы, с помощью которых в классы можно добавить «магию». Это методы всегда обрамлены двумя нижними подчеркиваниями (например, __init__ или __lt__).

Предупреждение: они плохо документированы (!!!). То есть, они описаны, но беспорядочно и почти без всякой системы.

Поэтому, Rafe Kettler, чтобы исправить то, что воспринимется как недостаток документации python о магических методах, постарался представить подробно, понятно, с большим количеством примеров.

Конструирование и инициализация магических методов

При создании объекта применяется магический метод

```
__new__(cls, [...])        # (класс, [список параметров])
```

Этот метод вызывается первым при создании и инициализации объекта. Он принимает в качестве параметров класс и любые другие аргументы, которые затем будут переданы в инициализатор объекта — магический метод __init__.

__new__ редко используется, но иногда бывает полезен, в частности, когда класс наследуется от неизменяемого (immutable) типа, такого как кортеж (tuple) или строка.

Далее вызывается базовый магический метод, __init__. Это инициализатор. С его помощью инициализируются объекты. При выполнении оператора x = SomeClass([...]) с помощью метода __new__ создаётся объект—представитель SomeClass, который вызывает __init__ и передаёт аргументы, заданные в конструкторе, в инициализатор.

__init__(self, [...]), инициализатор класса. Ему передаётся всё, с чем был вызван первоначальный конструктор (например, при выполнении оператора x = SomeClass(10, 'foo'), нестатический метод __init__ получает в качестве аргументов значения 10 и 'foo'. __init__ всегда (почти всегда) используется при определении объектов - представителей классов.

На другом конце жизненного цикла объекта находится метод __del__(self).

Если __new__ и __init__ образуют конструктор объекта, __del__ это его деструктор. Он не определяет поведение для выражения del x (поэтому этот код не эквивалентен x.__del__()). Скорее, он определяет поведение объекта в то время, когда объект попадает в сборщик мусора (!!!).

Это может быть удобно для объектов, которые могут требовать дополнительных чисток во время удаления, таких как сокеты или файловыве объекты. __del__ всегда вызывается по завершении работы интерпретатора. Однако __del__ не может служить заменой для хороших программистских практик (всегда завершать соединение, если объект завершил с ним работу и тому подобное).

Пример: __init__ и __del__ в действии:

```
from os.path import join

class FileObject:
    '''
    Обёртка для файлового объекта, чтобы быть уверенным в том,
    что файл будет закрыт при удалении.
    '''

    def __init__(self, filepath='~', filename='s.txt'):
        # открыть файл filename в filepath в режиме чтения и записи
        self.file = open(join(filepath, filename), 'r+')

    def __del__(self):
        self.file.close()
        del self.file


def Go(path, name):
    fo = FileObject(path, name)

if __name__=='__main__':
    Go('C:\PythonDrom\Texts_2022\~~~', 'sample.txt')
```

■ Переопределение операторов на произвольных классах

Одно из преимуществ применения магических методов в python – это то, что они предоставляют простой (!!!) способ заставить объекты вести себя подобно встроенным типам (классам)).

Это означает, что можно избежать поведения базовых операторов (унылого, нелогичного и нестандартного) (???).

В некоторых языках обычное явление писать как-нибудь так:

```
if instance.equals(other_instance): # если объект эквивалентен другому
```

```
                                    # объекту, то ...
    # do something
```

Можно, конечно, так же поступать и в python, но это добавляет путаницы и ненужной многословности (???). Разные библиотеки могут по разному называть одни и те же операции, заставляя использующего их программиста совершать больше действий, чем необходимо.

Применение магических методов позволяет определить нужный метод (__eq__, в следующем примере), и так точно выразить, что имелось в виду:

```
if instance == other_instance:
    #do something
```

Это одна из сильных сторон магических методов.

Большинство из них позволяют определить, что будут делать стандартные операторы,так что можно использовать операторы на своих классах так, как будто они представляют встроенные типы.

■ Магические методы сравнения

В python много магических методов, созданных для определения интуитивного (!!!) сравнения между объектами используя операторы, а не неуклюжие (!!!) методы.

Кроме того, они предоставляют способ переопределить поведение python по умолчанию для сравнения объектов (по ссылке). Вот список этих магических методов и что они делают:

- __cmp__(self, other) Самый базовый из методов сравнения. Он, в действительности, определяет поведение для всех операторов сравнения (>, ==, !=, итд.), но не всегда так, как это нужно. (например, если эквивалентность двух объектов определяется по одному критерию, а то что один больше другого по какому-нибудь другому).

__cmp__ должен вернуть:

отрицательное число, если self < other,

ноль, если self == other,

положительное число в случае self > other.

Но, обычно, лучше определить каждое сравнение, которое нужно, чем определять их всех в __cmp__. В то же время, __cmp__ может быть хорошим способом избежать повторений и увеличить ясность, когда все необходимые сравнения оперерируют одним критерием.

- __eq__(self, other) Определяет поведение оператора равенства, ==.
- __ne__(self, other) Определяет поведение оператора неравенства, !=.
- __lt__(self, other) Определяет поведение оператора меньше, <.
- __gt__(self, other) Определяет поведение оператора больше, >.
- __le__(self, other) Определяет поведение оператора меньше или равно, <=.
- __ge__(self, other) Определяет поведение оператора больше или равно, >=.

Для примера - класс, описывающий слово. Можно сравнивать слова лексикографически (по алфавиту), что является поведением, заданным по умолчанию при сравнении строк, но можно использовать при сравнении какой-нибудь другой критерий, например, длина или количество слогов. В этом примере реализовано сравнение по длине. Вот возможная реализация:

```
class Word(str):
    '''Класс для слов, определяющий сравнение по длине слов.'''

    def __new__(cls, word):
        # Надо использовать __new__, так как тип str неизменяемый
        # и он должен быть инициализирован раньше (при создании)
        if ' ' in word:
            print "Value contains spaces. Truncating to first space."
            word = word[:word.index(' ')] # Теперь Word это все символы до
                                          # первого пробела
        return str.__new__(cls, word)

    def __gt__(self, other):
        return len(self) > len(other)
    def __lt__(self, other):
        return len(self) < len(other)
    def __ge__(self, other):
        return len(self) >= len(other)
    def __le__(self, other):
        return len(self) <= len(other)


# Теперь можно создать два объекта Word (например, Word('foo') и Word('bar'))
# и сравнить их. Важно, что __eq__ и __ne__ не определяются,
# так как это приведёт к странному поведению
#(например, Word('foo') == Word('bar') будет расцениваться как истина).
# В этом нет смысла при тестировании на эквивалентность.
# Поэтому оставляется стандартная проверка на эквивалентность от str.

def Do(word1, word2):
    print(Word(word1) == Word(word2))
    print(Word(word1) != Word(word2))

if __name__=='__main__':
    Do('gold', 'sand')
    Do('gold', 'gold')
```

Теперь можно создать два объекта Word (например, Word('foo') и Word('bar')) и сравнить их.

Важно, что __eq__ и __ne__ не определяются, так как это приведёт к странному поведению (например, Word('foo') == Word('bar') будет расцениваться как истина).

В этом нет смысла при тестировании на эквивалентность. Поэтому оставляется стандартная проверка на эквивалентность от str.

Чтобы полностью охватить все сравнения, НЕ НУЖНО определять каждый из магических методов сравнения.

Стандартная библиотека предоставляет класс-декторатор в модуле functools, который определяет все сравнивающие методы. Достаточно определить только методы __eq__ и ещё один (__gt__, __lt__ и т.п.) (???)

Эта возможность доступна начиная с 2.7 версии python и позволяет сэкономить время и усилия.

Для того, чтобы задействовать эту библиотеку, надо разместить декоратор @total_ordering над определением класса. Как-то так:

```
from functools import total_ordering
@total_ordering
      ...
      ...
```

■ Создание недостающих методов сравнения пользовательского класса

Задача создания недостающих методов сравнения предполагает применение декоратора @total_ordering.

Синтаксис:

```
from functools import total_ordering
@total_ordering
class UserObject:
    ...
    ...
```

Параметры:

Нет.

Возвращаемое значение:

Добавляет в пользовательский класс недостающие методы сравнения.

Описание:

Декоратор класса @total_ordering модуля functools оборачивает класс, который определяет один или несколько методов сравнения и добавляет остальные методы сравнения. Такое поведение декоратора упрощает усилия по определению всех возможных операций расширенного сравнения.

Класс должен предоставлять метод __eq__(). И один из методов __lt__(), __le__(), __gt__() или __ge__().

■ Примечание.

Хотя этот декоратор позволяет создавать хорошо управляемые и полностью упорядоченные типы, это происходит за счет более медленного выполнения и более сложных трассировок стека для производных методов сравнения. Если анализ производительности показывает, что это является узким местом для данного приложения, отказ от применения декоратора @total_ordering модуля functools и реализация всех шести методов сравнения обеспечит повышение скорости.

Пример использования:

```
from functools import total_ordering
```

```
@total_ordering
class Student:


    def __init__(self, full_name):
        names = full_name.split()
        self.firstname = names[0]
        self.lastname = names[1]

    # hasattr роверяет наличие атрибута объекта.
    # Синтаксис:
    # hasattr(object, name)

    # Параметры:
    # object - объект, в котором нужно проверить существование атрибута name,
    # name - имя проверяемого атрибута.

    # Возвращаемое значение:
    # bool - True, если атрибут с именем name существует, иначе False.

    # Описание:
    # Функция hasattr() проверяет существование атрибута с именем name в
объекте object.
    # Возвращает True, если атрибут с именем name существует, иначе False.
    # Реализация функция hasattr() основывается на вызове функции getattr()
    # с последующей проверкой на предмет брошенного ей исключения
AttributeError.
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))

    def __eq__(self, other):
        print(f'__eq__ is here!')
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))

    def __lt__(self, other):
        print(f'__eq__ is here!')
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))

def DoIt(names):
    stud0 = Student(names[0])
    stud1 = Student(names[1])
    print(stud0 == stud1)

if __name__ =='__main__':
    DoIt(['john kennedy', 'lindon jonson'])
```

■ Числовые магические методы

Точно так же можно определить, каким образом объекты будут сравниваться операторами сравнения.

Можно определить их поведение для числовых операторов. Их много. Для лучшей организации числовые магические методы разделены на 5 категорий:

- унарные операторы,
- обычные арифметические операторы,
- отражённые арифметические операторы (подробности позже),
- составные присваивания,
- преобразования типов.

Унарные операторы и функции

Унарные операторы и функции имеют только один операнд — отрицание, абсолютное значение, и так далее.

- __pos__(self)

Определяет поведение для унарного плюса (+some_object)

- __neg__(self)

Определяет поведение для отрицания(-some_object)

- __abs__(self)

Определяет поведение для встроенной функции abs().

- __invert__(self)

Определяет поведение для инвертирования оператором ~. Для объяснения что он делает смотри статью в Википедии о бинарных операторах.

- __round__(self, n)

Определяет поведение для встроенной функции round(). n это число знаков после запятой, до которого округлить.

- __floor__(self)

Определяет поведение для math.floor(), то есть, округления до ближайшего меньшего целого.

- __ceil__(self)

Определяет поведение для math.ceil(), то есть, округления до ближайшего большего целого.

- __trunc__(self)

Определяет поведение для math.trunc(), то есть, обрезания до целого.

Обычные арифметические операторы

Теперь - описание обычных бинарных операторов (и ещё пару функций): +, -, * и похожие. Они, по большей части, хорошо описывают сами себя.

- __add__(self, other)

Сложение.

- __sub__(self, other)

Вычитание.

- __mul__(self, other)

Умножение.

- __floordiv__(self, other)

Целочисленное деление, оператор //.

- __div__(self, other)

Деление, оператор /.

- __truediv__(self, other)

Правильное деление. Заметьте, что это работает только когда используется from __future__ import division.

- __mod__(self, other)

Остаток от деления, оператор %.

- __divmod__(self, other)

Определяет поведение для встроенной функции divmod().

- __pow__

Возведение в степень, оператор **.

- __lshift__(self, other)

Двоичный сдвиг влево, оператор <<.

- __rshift__(self, other)

Двоичный сдвиг вправо, оператор >>.

- __and__(self, other)

Двоичное И, оператор &.

- __or__(self, other)

Двоичное ИЛИ, оператор |.

- __xor__(self, other)

Двоичный xor, оператор ^.

Отражённые арифметические операторы

Опредедение понятия отражённой арифметики можно начать с простого (???) примера:

```
some_object + other
```

Это «обычное» сложение. Единственное, чем отличается эквивалентное отражённое выражение, это порядок слагаемых:

```
other + some_object
```

Таким образом, все эти магические методы делают то же самое, что и их обычные версии, за исключением выполнения операции с other в качестве первого операнда и self в качестве второго. В большинстве случаев, результат отражённой операции такой же, как её обычный эквивалент, поэтому при определении __radd__ можно ограничиться вызовом __add__ и всё...

Важно, что объект слева от оператора (other в примере) не должен иметь обычной неотражённой версии этого метода. В этом примере, some_object.__radd__ будет вызван только если в other не определён __add__.

- __radd__(self, other)

Отражённое сложение.

- __rsub__(self, other)

Отражённое вычитание.

- __rmul__(self, other)

Отражённое умножение.

- __rfloordiv__(self, other)

Отражённое целочисленное деление, оператор //.

- __rdiv__(self, other)

Отражённое деление, оператор /.

- __rtruediv__(self, other)

Отражённое правильное деление. Оно работает только когда используется модуль

from __future__ import division.

- __rmod__(self, other)

Отражённый остаток от деления, оператор %.

- __rdivmod__(self, other)

Определяет поведение для встроенной функции divmod(), когда вызывается divmod(other, self).

- __rpow__

Отражённое возведение в степерь, оператор **.

- __rlshift__(self, other)

Отражённый двоичный сдвиг влево, оператор <<.

- __rrshift__(self, other)

Отражённый двоичный сдвиг вправо, оператор >>.

- __rand__(self, other)

Отражённое двоичное И, оператор &.

- __ror__(self, other)

Отражённое двоичное ИЛИ, оператор |.

- __rxor__(self, other)

Отражённый двоичный xor, оператор ^.

Составное присваивание

В python также представлены и магические методы для составного присваивания. Составное присваивани, это комбинация «обычного» оператора и присваивания. Вот пример:

```
x = 5
x += 1 # другими словами x = x + 1
```

Каждый из этих методов должен возвращать значение, которое будет присвоено переменной слева (например, для a += b, __iadd__ должен вернуть a + b, что будет присвоено a). Вот список:

- __iadd__(self, other)

Сложение с присваиванием.

- __isub__(self, other)

Вычитание с присваиванием.

- __imul__(self, other)

Умножение с присваиванием.

- __ifloordiv__(self, other)

Целочисленное деление с присваиванием, оператор //=.

- __idiv__(self, other)

Деление с присваиванием, оператор /=.

- __itruediv__(self, other)

Правильное деление с присваиванием. Работает только если импортируется модуль from __future__ import division.

- __imod_(self, other)

Остаток от деления с присваиванием, оператор %=.

- __ipow__

Возведение в степерь с присваиванием, оператор **=.

- __ilshift__(self, other)

Двоичный сдвиг влево с присваиванием, оператор <<=.

- __irshift__(self, other)

Двоичный сдвиг вправо с присваиванием, оператор >>=.

- __iand__(self, other)

Двоичное И с присваиванием, оператор &=.

__ior__(self, other)

- Двоичное ИЛИ с присваиванием, оператор |=.

- __ixor__(self, other)

Двоичный xor с присваиванием, оператор ^=.

■ Магические методы преобразования типов

Кроме того, в python множество магических методов, предназначенных для определния поведения для встроенных функций преобразования типов, таких как float(). Вот они:

- __int__(self)

Преобразование типа в int.

- __long__(self)

Преобразование типа в long.

- __float__(self)

Преобразование типа в float.

- __complex__(self)

Преобразование типа в комплексное число.

- __oct__(self)

Преобразование типа в восьмеричное число.

- __hex__(self)

Преобразование типа в шестнадцатиричное число.

- __index__(self)

Преобразование типа к int, когда объект используется в срезах (выражения вида [start:stop:step]). Если определяется собственный числовый тип, который может использоваться как индекс списка, нужно определить __index__.

- __trunc__(self)

Вызывается при math.trunc(self). Должен вернуть своё значение, обрезанное до целочисленного типа (обычно long).

- __coerce__(self, other)

Метод для реализации арифметики с операндами разных типов. __coerce__ должен вернуть None если преобразование типов невозможно. Если преобразование возможно, он должен вернуть пару (кортеж из 2-х элементов) из self и other, преобразованные к одному типу.

■ Представление собственных классов

Часто бывает полезно представление класса в виде строки. В python существует несколько методов, которые можно определить для настройки поведения встроенных функций при представлении определяемого класса.

- __str__(self)

Определяет поведение функции str(), вызванной для объекта собственного класса.

- __repr__(self)

Определяет поведение функции repr(), вызыванной для объекта собственного класса. Главное отличие от str() в целевой аудитории. repr() больше предназначен для машинно-ориентированного вывода (более того, это часто должен быть валидный код на python), а str() предназначен для чтения людьми.

- __unicode__(self)

Определяет поведение функции unicode(), вызыванной для объекта собственного класса. unicode() похож на str(), но возвращает строку в юникоде. С этим методом надо быть осторожным: если клиент вызывает str() на объекте собственного класса, а был определён только __unicode__(), то это не будет работать. Желательно всегда определять __str__() для случая, когда кто-то не имеет такой роскоши как юникод.

- __format__(self, formatstr)

Определяет поведение, когда объект собственного класса используется при форматировании строк нового стиля. Например, "Hello, {0:abc}!".format(a) приведёт к вызову a.__format__("abc"). Это может быть полезно для определения собственных числовых или строковых типов, которым можно будет предоставить какие-нибудь специальные опции форматирования.

- __hash__(self)

Определяет поведение функции hash(), вызыванной для объекта собственного класса. Метод должен возвращать целочисленное значение, которое будет использоваться для быстрого сравнения ключей в словарях. В таком случае обычно также нужно определять и __eq__. Здесь действует следующее правило: a == b предполагаеает hash(a) == hash(b).

- __nonzero__(self)

Определяет поведение метода bool(), вызванного для объекта собственного класса. Должен вернуть True или False, в зависимости от того, когда объект будет считаться соответствующим True или False.

- __dir__(self)

Определяет поведение метода dir(), вызванного для объекта собственного класса. Этот метод должен возвращать список атрибутов. Обычно, определение __dir__ не требуется, но может быть важно для интерактивного использования собственного класса, если переопределяются методы __getattr__ или __getattribute__ (которые описаны в следующей части), или каким-либо другим образом динамически создаются атрибуты.

- __sizeof__(self)

Определяет поведение функции sys.getsizeof(), вызванного для объекта собственного класса. Метод должен вернуть размер объекта в байтах. Он главным образом полезен для классов, определённых в расширениях на C, но всё-равно полезно о нём знать.

Контроль доступа к атрибутам

Одной из особенностей языка pynhon в отличие от других языков программирования, является отсутствие настоящей инкапсуляции для классов (например, нет способа определить приватные атрибуты с публичными методами доступа). Однако это не совсем правда: просто многие вещи, связанные с инкапсуляцией, pynhon реализует через «магию», а не с помощью явных модификаторов для методов и полей. Например, метод

- __getattr__(self, name)

Можно определить его поведение для случая, когда пользователь пытается обратиться к атрибуту, который (совсем или пока ещё) не существует. Это может быть полезным для перехвата и перенаправления опечаток, предупреждения об использовании устаревших атрибутов (при желании можно всё равно вычислить и вернуть этот атрибут), или в случае необходимости хитро возвращать AttributeError. Правда, этот метод вызывается только когда пытаются получить доступ к несуществующему атрибуту, поэтому это не очень хорошее решение для инкапсуляции.

- __setattr__(self, name, value)

В отличии от __getattr__, __setattr__ представляет решение для инкапсуляции. Этот метод позволяет определить поведение для присвоения значения атрибуту, независимо от того существует атрибут или нет. То есть, можно определить любые правила для любых изменений значения атрибутов. Впрочем, надо быть осторожным с тем, как использовать __setattr__, (пример нехорошего случая приводится в конце этого списка).

- __delattr__

Это то же, что и __setattr__, но для удаления атрибутов, вместо установки значений. Здесь требуются те же меры предосторожности, что и в __setattr__ чтобы избежать бесконечной рекурсии (вызов del self.name в определении __delattr__ вызовет бесконечную рекурсию).

- __getattribute__(self, name)

__getattribute__ выглядит к месту среди методов __setattr__ и __delattr__, но всё равно не рекомендуется его использовать. __getattribute__ может использоваться только с классами нового типа (в новых версиях python все классы нового типа, а в старых версиях можно получить такой класс унаследовавшись от object). Этот метод позволяет определить поведение для каждого случая доступа к атрибутам (а не только к несуществующим, как __getattr__(self, name)). Он страдает от таких же проблем с бесконечной рекурсией, как и его коллеги (на этот раз можно вызывать __getattribute__ у базового класса, чтобы их предотвратить). Он, так же, главным образом устраняет необходимость в __getattr__, который в случае реализации __getattribute__ может быть вызван только явным образом или в случае генерации исключения AttributeError. Конечно можно использовать этот метод, но лучше этого не делать, потому что случаев, когда он действительно полезен очень мало (намного реже нужно переопределять поведение при получении, а не при установке значения) и реализовать его без возможных ошибок очень сложно.

Очень легко можно получить проблему при определении любого метода, управляющего доступом к атрибутам. Например:

```
def __setattr__(self, name, value):
    self.name = value
    # это рекурсия, так как всякий раз, когда любому атрибуту
    # присваивается значение, вызывается  __setattr__().
    # то есть, на самом деле это равнозначно self.__setattr__('name', value).
    # Так как метод вызывает сам себя, рекурсия продолжится бесконечно,
    # пока всё не упадёт

def __setattr__(self, name, value):
    self.__dict__[name] = value # присваивание в словарь переменных класса
                                # дальше определение произвольного поведения
```

Мощь магических методов в python велика, а с большой силой приходит и большая ответственность. Важно знать, как правильно использовать магические методы, ничего не ломая.

Итак, что известно об управлении доступом к атрибутам? Их не нужно использовать легкомысленно. На самом деле, они имеют склонность к чрезмерной мощи и нелогичности. Причина, по которой они всё-таки существуют, в удволетворении определённого желания: python склонен не запрещать плохие штуки полностью, а только усложнять их использование. Свобода превыше всего, поэтому поэтому можно делать всё без ограничений.

Дальше пример использования методов контроля доступа (при этом используется функция super, так как не все классы имеют атрибут __dict__):

■ Функция super() в python: доступ к унаследованным методам

(https://docs-python.ru/tutorial/)

Обеспечивает доступ к оригиналам наследованных методов.

Синтаксис:

super(type, object-or-type)

Параметры:

type - необязательно, тип, от которого начинается поиск объекта-посредника

object-or-type - необязательно, тип или объект, определяет порядок разрешения метода для поиска

Возвращаемое значение:

объект-посредник, делегирующий вызовы методов родителю или собрату класса.

Описание:

Функция super(), возвращает объект-посредник, который делегирует вызовы метода родительскому или родственному классу, указанного type типа. Это полезно для доступа к унаследованным методам, которые были переопределены в классе.

■ object-or-type - применение

object-or-type определяет порядок разрешения метода __mro__ для поиска. Поиск начинается с класса, сразу после указанного типа. Например, если __mro__ - это D -> B -> C -> A -> object, а значение type=B, то super() выполняет поиск объекта C -> A -> object.

Если object-or-type не указан, то возвращается несвязанный объект-посредник.

Если object-or-type является объектом (экземпляром), то будет получен посредник, для которого isinstance(obj, type) возвращает True.

Если object-or-type является типом (классом), то будет получен посредник, для которого issubclass(subtype, type) возвращает True.

■ Типичные случаи использования super()

■ Иерархия с единичным наследованием

В иерархиях с единичным наследованием используется для обращения к родительским классам, чтобы явно не указывать их имена. Это упрощает поддержку кода в дальнейшем.

Например:

```
class A:
    def some_method(self):
        print('some_method A')

class B(A):
    def some_method(self):
        print('some_method B')

x = B()
x.some_method()

y = A()
y.some_method()

# some_method B
```

Здесь перегружен метод родительского класса. Но что делать, если необходимо немного(???) дополнить родительский метод, не копируя его полностью? Тут и нужна функция super():

```
class A:
    def some_method(self):
        print('some_method A')

class B(A):
    def some_method(self):
        super().some_method()     # Вызов функции super
        print('some_method B')
```

```
x = B()
x.some_method()

# Результат выполнения функции super
# some_method A
# some_method B
```

■ Поддержка совместного множественного наследования

Для поддержки совместного множественного наследования в динамическом окружении super делает возможным обращение с ромбовидными иерархиями, при которых несколько базовых классов задают реализацию метода с одним и тем же именем.

Использование функция super() с обоими аргументами точно определяет объекты и делает соответствующие ссылки. Без аргументов функция super() работает только внутри определения класса, а необходимые детали для идентификации класса и доступа к текущему экземпляру для методов заполняет компилятор.

В дополнение к поиску методов, super() также работает для поиска атрибутов. Одним из вариантов использования этого является вызов дескрипторов в родительском или родственном классе.

Функция super() реализована как часть процесса привязки для явного поиска по точечным атрибутам, таких как super().__getitem__(name). Это достигается путем реализации собственного метода __getattribute__() для поиска классов в предсказуемом порядке, который поддерживает кооперативное множественное наследование __mro__. super() и не предназначена для неявных поисков с использованием инструкций или операторов, таких как super()[name].

■ Примеры получения доступа к унаследованным методам.

■ Функция в единичном наследовании

```
class Computer:
    def __init__(self, computer, ram, ssd):
        self.computer = computer
        self.ram = ram
        self.ssd = ssd

# Если создать дочерний класс 'Laptop', то будет доступ
# к свойству базового класса благодаря функции super().
class Laptop(Computer):
    def __init__(self, computer, ram, ssd, model):
        super().__init__(computer, ram, ssd)
        self.model = model


lenovo = Laptop('lenovo', 2, 512, 'l420')

print('This computer is:', lenovo.computer)
print('This computer has ram of', lenovo.ram)
print('This computer has ssd of', lenovo.ssd)
print('This computer has this model:', lenovo.model)

# Вывод
# This computer is: lenovo
# This computer has ram of 2
# This computer has ssd of 512
```

```
# This computer has this model: l420
```

В следующем примере класс Rectangle является суперклассом, а Square является подклассом, поскольку методы Square наследуются от Rectangle, то можно вызвать метод __init __() суперкласса (Rectangle.__ init __()) из класса Square используя функцию super().

Далее просто пользоваться методами родителя, не написав ни строчки кода (!!!).

В данном случае квадрат - это частный случай прямоугольника (и его дочерний класс).

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Square(Rectangle):
    def __init__(self, length):
        # Для квадрата просто нужно передать один параметр length.
        # При вызове 'super().__init__()' установим атрибуты 'length' и
'width'.
        super().__init__(length, length)

# Класс 'Square' явно не реализует метод 'area()' и
# будет использовать его из суперкласса 'Rectangle'
sqr = Square(4)
print("Area of Square is:", sqr.area())
# Area of Square is: 16
print("Perimeter of Square is:", sqr.perimeter())
# Perimeter of Square is: 16
rect = Rectangle(2, 4)
print("Area of Rectangle is:", rect.area())
# Area of Rectangle is: 8
print("Perimeter of Rectangle is:", rect.perimeter())
# Perimeter of Square is: 12
```

И наконец пример работы функции super() при использовании множественного наследования:

```
class A:
    def __init__(self):
        print('Initializing: class A')

    def sub_method(self, b):
        print('sub_method from class A:', b)


class B(A):
    def __init__(self):
        print('Initializing: class B')
        super().__init__()

    def sub_method(self, b):
        print('sub_method from class B:', b)
```

```
        super().sub_method(b + 1)

class X(B):
    def __init__(self):
        print('Initializing: class X')
        super().__init__()

    def sub_method(self, b):
        print('sub_method from class X:', b)
        super().sub_method(b + 1)

class Y(X):
    def __init__(self):
        print('Initializing: class Y')
        # super() с параметрами
        super(X, self).__init__()

    def sub_method(self, b):
        print('sub_method from class Y:', b)
        super().sub_method(b + 1)

x = X()
x.sub_method(1)
print('Обратите внимание как происходит инициализация')
print('классов при указании аргументов в функции super()')
y = Y()
y.sub_method(5)

# Вывод

# Initializing: class X
# Initializing: class B
# Initializing: class A
# sub_method from class X: 1
# sub_method from class B: 2
# sub_method from class A: 3
# важно, как происходит инициализация
# классов при указании аргументов в функции super()
# Initializing: class Y
# Initializing: class B
# Initializing: class A
# sub_method from class Y: 5
# sub_method from class X: 6
# sub_method from class B: 7
# sub_method from class A: 8
```

Класс, содержащий атрибут value и реализующий счётчик доступа к нему. Счётчик увеличивается каждый раз, когда меняется value.

```
class AccessCounter(object):
'''
Класс, содержащий атрибут value и реализующий счётчик доступа к нему.
Счётчик увеличивается каждый раз, когда меняется value.
'''
```

```
def __init__(self, val):
    super(AccessCounter, self).__setattr__('counter', 0)
    super(AccessCounter, self).__setattr__('value', val)

def __setattr__(self, name, value):
    if name == 'value':
        super(AccessCounter, self).__setattr__('counter',
                                               self.counter + 1)
    # Здесь нет никаких условий.
    # Нужно предотвратить изменение других атрибутов,
    # надо выбросить исключение AttributeError(name)
    super(AccessCounter, self).__setattr__(name, value)

def __delattr__(self, name):
    if name == 'value':
        super(AccessCounter, self).__setattr__('counter',
                                               self.counter + 1)
    super(AccessCounter, self).__delattr__(name)]
```

■ Произвольные последовательности

В python существует множество способов заставить классы вести себя как встроенные последовательности (словари, кортежи, списки, строки и так далее). Это магические методы из-за высокой степени контроля, которую они дают и той магии, от которой с объектами классов начинает работать целое множество глобальных функций. Но, до того как будут описаны всякие важные вещи, надо знать о протоколах.

■ Протоколы

Теперь, когда речь зашла о создании собственных последовательностей в python, нужно ознакомиться к протоколам. Протоколы похожи на интерфейсы в других языках тем, что они предоставляют набор методов, которые необходимо реализовать. Однако протоколы в python абсолютно ни к чему не обязывают и не требуют обязательной реализации какого-либо объявления. Наверное, они больше похожи на руководящие указания.

Почему речь о протоколах? Потому, что реализация произвольных контейнерных типов в python влечёт за собой использование некоторых из них.

Во-первых, протокол для определения неизменяемых контейнеров: чтобы создать неизменяемый контейнер, нужно только определить __len__ и __getitem__ (продробнее о них дальше). Протокол изменяемого контейнера требует того же, что и неизменяемого контейнера, плюс __setitem__ и __delitem__.

Во-вторых, наконец, если необходимо, чтобы объекты можно было перебирать итерацией, нужно определить __iter__, который возвращает итератор. Этот итератор должен соответствовать протоколу итератора, который требует методов __iter__(возвращает самого себя) и next.

■ Магия контейнеров

Магические методы, используемые контейнерами:

- __len__(self)

Возвращает количество элементов в контейнере. Часть протоколов для изменяемого и неизменяемого контейнеров.

- __getitem__(self, key)

Определяет поведение при доступе к элементу, используя синтаксис self[key]. Тоже относится и к протоколу изменяемых и к протоколу неизменяемых контейнеров. Должен выбрасывать соответствующие исключения: TypeError если неправильный тип ключа и KeyError если ключу не соответствует никакого значения.

- __setitem__(self, key, value)

Определяет поведение при присваивании значения элементу, используя синтаксис

self[nkey] = value.

Часть протокола изменяемого контейнера. Опять же, нужно выбрасывать KeyError и TypeError в соответсвующих случаях.

- __delitem__(self, key)

Определяет поведение при удалении элемента (то есть del self[key]). Это часть только протокола для изменяемого контейнера. Вы должны выбрасывать соответствующее исключение, если ключ некорректен.

- __iter__(self)

Должен вернуть итератор для контейнера. Итераторы возвращаются в множестве ситуаций, главным образом для встроенной функции iter() и в случае перебора элементов контейнера выражением for x in container:. Итераторы сами по себе объекты и они тоже должны определять метод __iter__, который возвращает self.

- __reversed__(self)

Вызывается чтобы определить поведения для встроенной функции reversed(). Должен вернуть обратную версию последовательности. Реализуйте метод только если класс упорядоченный, как список или кортеж.

- __contains__(self, item)

__contains__ предназначен для проверки принадлежности элемента с помощью in и not in. Это не часть протокола последовательности. Потому что когда __contains__ не определён, python просто перебирает всю последовательность элемент за элементом и возвращает True если находит нужный.

- __missing__(self, key)

__missing__ используется при наследовании от dict. Определяет поведение для для каждого случая, когда пытаются получить элемент по несуществующему ключу (так, например, если

имеется есть словарь d и пишется d["george"] когда "george" не является ключом в словаре, вызывается d.__missing__("george")).

■ Пример

Для примера можно рассмотреть список, который реализует некоторые функциональные конструкции, которые можно было бы встретить в других языках.

```
class FunctionalList:
    '''
    Класс-обёртка над списком с добавлением некоторой функциональной магии:
    head, tail, init, last, drop, take.
    '''

    def __init__(self, values=None):
        if values is None:
            self.values = []
        else:
            self.values = values

    def __len__(self):
        return len(self.values)

    def __getitem__(self, key):
        # если значение или тип ключа некорректны, list выбросит исключение
        return self.values[key]

    def __setitem__(self, key, value):
        self.values[key] = value

    def __delitem__(self, key):
        del self.values[key]

    def __iter__(self):
        return iter(self.values)

    def __reversed__(self):
        return FunctionalList(reversed(self.values))

    def append(self, value):
        self.values.append(value)

    def head(self):
        # получить первый элемент
        return self.values[0]

    def tail(self):
        # получить все элементы после первого
        return self.values[1:]

    def init(self):
        # получить все элементы кроме последнего
        return self.values[:-1]
```

```
    def last(self):
        # получить последний элемент
        return self.values[-1]

    def drop(self, n):
        # все элементы кроме первых n
        return self.values[n:]

    def take(self, n):
        # первые n элементов
        return self.values[:n]


def DoIt(parameters):
    fl = FunctionalList(parameters)
    res = fl.take(3)
    return res

if __name__ == '__main__':
    parameters = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    res = DoIt(parameters)
    print(res)
```

Теперь есть полезный (относительно) пример реализации своей собственной последовательности. Существуют, конечно, и куда более практичные реализации произвольных последовательностей, но большое их число уже реализовано в стандартной библиотеке, такие как Counter, OrderedDict, NamedTuple.

■ Отражение

Вы можно контролировать и отражение, использующее встроенные функции isinstance() и issubclass(), определив некоторые магические методы:

- __instancecheck__(self, instance)

Проверяет, является ли экземлпяр членом пользовательского класса (isinstance(instance, class), например.

- __subclasscheck__(self, subclass)

Проверяет, является наследуется ли класс от пользовательского класса (issubclass(subclass, class)).

Может показаться, что вариантов полезного использования этих магических методов немного и, возможно, это на самом деле так. Можно не тратить слишком много времени на магические методы отражения, не особо они и важные, но они отражают кое-что важное об объектно-ориентированном программировании в python и о python вообще: почти всегда существует простой способ что-либо сделать, даже если надобность в этом «что-либо» возникает очень редко. Эти магические методы могут не выглядеть полезными, но если они когда-нибудь понадобятся, будет хорошо, что они есть (для этого и пишется эта глава!).

■ Вызываемые объекты

Как уже известно, в python функции являются объектами первого класса. Это означает, что они могут быть переданы в функции или методы так же, как любые другие объекты. Это невероятно мощная особенность.

Специальный магический метод позволяет объектам пользовательского класса вести себя так, как будто они функции, то есть их можно «вызывать» их, передавать их в функции, которые принимают функции в качестве аргументов и так далее. Это другая удобная особенность, которая делает программирование на python таким приятным.

■ __call__(self, [args...])

Позволяет любому экземпляру класса быть вызванным как-будто он функция. Главным образом это означает, что x() означает то же, что и x.__call__(). При этом, __call__ принимает произвольное число аргументов; то есть, можно определить __call__ так же как любую другую функцию, принимающую столько аргументов, сколько это нужно.

__call__, в частности, может быть полезен в классах, чьи экземпляры часто изменяют своё состояние. «Вызвать» объект может быть интуитивно понятным и элегантным способом изменить состояние объекта. Примером может быть класс, представляющий положение некоторого объекта на плоскости:

```
class Entity:
'''
Класс, описывающий объект на плоскости.
"Вызываемый", чтобы обновить позицию объекта.
'''

    def __init__(self, size, x, y):
        self.x, self.y = x, y
        self.size = size

    def __call__(self, x, y):
        # Изменить положение объекта.
        self.x, self.y = x, y
```

■ Менеджеры контекста

В python 2.5 было представлено новое ключевое слово вместе с новым способом повторно использовать код. Это ключевое слово with. Концепция менеджеров контекста не являлась новой для python (она была реализована раньше как часть библиотеки), но в PEP 343 достигла статуса языковой конструкции. Выражения с with:

```
with open('foo.txt') as bar:
    # выполнение каких-нибудь действий с bar
```

Менеджеры контекста позволяют выполнить какие-то действия для настройки или очистки, когда создание объекта обёрнуто в оператор with. Поведение менеджера контекста определяется двумя магическими методами:

- __enter__(self)

Определяет, что должен сделать менеджер контекста в начале блока, созданного оператором with. Заметьте, что возвращаемое __enter__ значение и есть то значение, с которым производится работа внутри with.

- __exit__(self, exception_type, exception_value, traceback)

Определяет действия менеджера контекста после того, как блок будет выполнен (или прерван во время работы). Может использоваться для контроллирования исключений, очистки, любых действий которые должны быть выполнены незамедлительно после блока внутри with.

Если блок выполнен успешно, exception_type, exception_value, и traceback будут установлены в None. В другом случае выбор, перехватывать ли исключение или предоставить это пользователю. Если принято решение перехватить исключение, надо убедиться, что __exit__ возвращает True после того как всё сделано. Если не предусматривается, чтобы исключение было перехвачено менеджером контекста, просто надо позволить ему случиться.

__enter__ и __exit__ могут быть полезны для специфичных классов с хорошо описанным и распространённым поведением для их настройки и очистки ресурсов. Можно использовать эти методы и для создания общих менеджеров контекста для разных объектов. Вот пример:

```
class Closer:
'''
Менеджер контекста для автоматического закрытия объекта
вызовом метода close в with-выражении.
'''

    def __init__(self, obj):
        self.obj = obj

    def __enter__(self):
        return self.obj # привязка к активному объекту with-блока

    def __exit__(self, exception_type, exception_val, trace):
        try:
           self.obj.close()
        except AttributeError: # у объекта нет метода close
           print 'Not closable.'
           return True # исключение перехвачено
```

Пример использования Closer с FTP-соединением (сокет, имеющий метод close):

```
>>> from magicmethods import Closer
>>> from ftplib import FTP
>>> with Closer(FTP('ftp.somesite.com')) as conn:
...     conn.dir()
...
# output omitted for brevity
>>> conn.dir()
# long AttributeError message, can't use a connection that's closed
>>> with Closer(int(5)) as i:
...     i += 1
```

```
...
Not closable.
>>> i
6
```

Здесь видно, как обёртка управляется и с правильными и с неподходящими объектами. В этом сила менеджеров контекста и магических методов. Стандартная библиотека python включает модуль contextlib, который включает в себя contextlib.closing() — менеджер контекста, который делает приблизительно то же (без какой-либо обработки случая, когда объект не имеет метода close()).

■ Абстрактные базовые классы

Смотреть http://docs.python.org/2/library/abc.html.

■ Построение дескрипторов

Дескрипторы это такие классы, с помощью которых можно добавить свою логику к событиям доступа (получение, изменение, удаление) к атрибутам других объектов. Дескрипторы не подразумевается использовать сами по себе; скорее, предполагается, что ими будут владеть какие-нибудь связанные с ними классы. Дескрипторы могут быть полезны для построения объектно-ориентированных баз данных или классов, чьи атрибуты зависят друг от друга. В частности, дескрипторы полезны при представлении атрибутов в нескольких системах исчисления или каких-либо вычисляемых атрибутов (как расстояние от начальной точки до представленной атрибутом точки на сетке).

Чтобы класс стал дескриптором, он должен реализовать по крайней мере один метод из __get__, __set__ или __delete__. Эти магические методы подробно:

- __get__(self, instance, instance_class)

Определяет поведение при возвращении значения из дескриптора. instance это объект, для чьего атрибута-дескриптора вызывается метод. owner это тип (класс) объекта.

- __set__(self, instance, value)

Определяет поведение при изменении значения из дескриптора. instance это объект, для чьего атрибута-дескриптора вызывается метод. value это значение для установки в дескриптор.

- __delete__(self, instance)

Определяет поведение для удаления значения из дескриптора. instance это объект, владеющий дескриптором.

■ Пример использования дескрипторов

Теперь пример полезного (???) использования дескрипторов: преобразование единиц измерения.

```
class Meter(object):
'''
    Дескриптор для метра. У него есть метод __init__(self, value),
    значит, он один может принимать параметры при создании объектов
'''

    def __init__(self, value=0.0):
        self.value = float(value)

    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)

class Foot(object):
    '''Дескриптор для фута.'''

    def __get__(self, instance, owner):
        print(f'Foot __get__')
        return instance.meter * 3.2808

    def __set__(self, instance, value):
        print(f'Foot __set__')
        instance.meter = float(value) / 3.2808

class Distance(object):
    '''
    Класс, описывающий расстояние.
    содержит два статических дескриптора для метров и футов.
    '''

    # значения передаются ТОЛЬКО в конструктор класса Meter
    meter = Meter(10.0)
    foot = Foot()


def DoIt():
    d = Distance()
    print(f'{d.meter} ... {d.foot} ')


if __name__ == '__main__':
   DoIt()
```

■ Копирование

В python оператор присваивания не копирует объекты, а только добавляет ещё одну ссылку. Но для коллекций, содержащих изменяемые элементы, иногда необходимо полноценное копирование, чтобы можно было менять элементы одной последовательности, не затрагивая другую. Здесь применяется copy. И инструкции для python как правильно копировать.

- __copy__(self)

Определяет поведение copy.copy() для объекта пользовательского класса . copy.copy() возвращает поверхностную копию объекта — это означает, что хоть сам объект и создан заново, все его данные ссылаются на данные оригинального объекта. И при изменении данных нового объекта, изменения будут происходить и в оригинальном.

- __deepcopy__(self, memodict={})

Определяет поведение copy.deepcopy() для объекта пользовательского класса. copy.deepcopy() возвращает глубокую копию объекта — копируются и объект и его данные. memodict это кэш предыдущих скопированных объектов, он предназначен для оптимизации копирования и предотвращения бесконечной рекурсии, когда копируются рекурсивные структуры данных. Когда требуется полностью скопировать какой-нибудь конкретный атрибут, на нём вызывается copy.deepcopy() с первым параметром memodict.

Когда использовать эти магические методы? Как обычно — в любом случае, когда требуется больше, чем стандартное поведение. Например, если нужно скопировать объект, который содержит кэш как словарь (возможно, очень большой словарь), то может быть и не нужно копировать весь кэш, а обойтись всего одним в общей памяти объектов.

Использование модуля pickle на объектах пользовательского класса

Pickle это модуль для сериализации структур данных python и он может быть полезен, когда нужно сохранить состояние какого-либо объекта и восстановить его позже (обычно, в целях кэширования). Кроме того, это ещё и отличный источник переживаний и путаницы.

Сериализация настолько важна, что кроме своего модуля (pickle) имеет и свой собственный протокол и свои магические методы. Но для начала о том, как сериализовать с помощью pickle уже существующие типы данных.

Сериализация

Теперь о сериализации. Пусть есть словарь, который нужно сохранить и восстановить позже. Нужно записать его содержимое в файл, убедившись, что он пишется с правильным синтаксисом, потом восстановить его, выполнив exec(), или прочитав файл. Но это в лучшем случае рискованно: если хранить важные данные в тексте, он может быть повреждён или изменён множеством способов, с целью обрушить программу или, вообще, запустить какой-нибудь опасный код. Лучше использовать pickle:

```
import pickle

data = {'foo': [1, 2, 3],
        'bar': ('Hello', 'world!'),
        'baz': True}
jar = open('data.pkl', 'wb')
pickle.dump(data, jar) # записать сериализованные данные в jar
jar.close()
```

И вот, спустя некоторое время, снова нужен словарь:

```
import pickle

pkl_file = open('data.pkl', 'rb') # открывается
data = pickle.load(pkl_file) # сохраняется в переменной
print data
```

```
pkl_file.close()
```

Что произошло? Точно то, что и ожидалось. data как-будто всегда тут и была.

Теперь, об осторожности: pickle не идеален. Его файлы легко испортить случайно или преднамеренно. Pickle, может быть, безопаснее чем текстовые файлы, но он всё ещё может использоваться для запуска вредоносного кода. Кроме того, он несовместим между разными версиями python, поэтому если распространять объекты с помощью pickle, не обязательно, что все смогут их использовать. Тем не менее, модуль может быть мощным инструментом для кэширования и других распространённых задач с сериализацией.

■ Сериализация собственных объектов

Модуль pickle не только для встроенных типов. Он может использоваться с каждым классом, реализующим его протокол. Этот протокол содержит четыре необязательных метода, позволяющих настроить то, как pickle будет с ними обращаться (есть некоторые различия для расширений на C, но это за рамками нашего руководства):

- __getinitargs__(self)

Если необходимо, чтобы после десериализации класса был вызыван __init__, то можно определить __getinitargs__, который должен вернуть кортеж аргументов, который будет отправлен в __init__. Этот метод работает только с классами старого стиля.

- __getnewargs__(self)

Для классов нового стиля можно определить, какие параметры будут переданы в __new__ во время десериализации. Этот метод так же должен вернуть кортеж аргументов, которые будут отправлены в __new__.

- __getstate__(self)

Вместо стандартного атрибута __dict__, где хранятся атрибуты класса, можно вернуть произвольные данные для сериализации. Эти данные будут переданы в __setstate__ во время десериализации.

- __setstate__(self, state)

Если во время десериализации определён __setstate__, то данные объекта будут переданы сюда, вместо того чтобы просто записать всё в __dict__. Это парный метод для __getstate__: когда оба определены, вы можете представлять состояние вашего объекта так, как вы только захотите.

- __reduce__(self)

Если определён собственный тип (с помощью Python's C API), надо сообщить python как его сериализовать, если нужно, чтобы он его сериализовал. __reduce__() вызывается когда сериализуется объект, в котором этот метод был определён. Он должен вернуть или строку, содержащую имя глобальной переменной, содержимое которой сериализуется как обычно, или кортеж. Кортеж может содержать от 2 до 5 элементов: вызываемый объект, который будет вызван, чтобы создать десериализованный объект, кортеж аргументов для этого вызываемого

объекта, данные, которые будут переданы в __setstate__ (опционально), итератор списка элементов для сериализации (опционально) и итератор словаря элементов для сериализации (опционально).

- __reduce_ex__(self, protocol)

Иногда полезно знать версию протокола, реализуя __reduce__. И этого можно добиться, реализовав вместо него __reduce_ex__. Если __reduce_ex__ реализован, то предпочтение при вызове отдаётся ему (при этом всё равно нужно реализовать __reduce__ для обратной совместимости).

Пример

Для примера описывается грифельная доска (Slate), которая запоминает что и когда было на ней записано. Впрочем, конкретно эта доска становится чистой каждый раз, когда она сериализуется: текущее значение не сохраняется.

```
import time

class Slate:
    '''Класс, хранящий строку и лог изменений. И забывающий своё значение
после
    сериализации.'''

    def __init__(self, value):
        self.value = value
        self.last_change = time.asctime()
        self.history = {}

    def change(self, new_value):
        # Изменить значение. Зафиксировать последнее значение в истории.
        self.history[self.last_change] = self.value
        self.value = new_value
        self.last_change = time.asctime()

    def print_changes(self):
        print 'Changelog for Slate object:'
        for k, v in self.history.items():
            print '%s\t %s' % (k, v)

    def __getstate__(self):
        # Намеренно не возвращаем self.value or self.last_change.
        # Мы хотим "чистую доску" после десериализации.
        return self.history

    def __setstate__(self, state):
        self.history = state
        self.value, self.last_change = None, None
```

Дополнение 1: Как вызывать магические методы

Некоторые из магических методов напрямую связаны со встроенными функциями; в этом случае совершенно очевидно как их вызывать. Однако, так бывает не всегда. Это дополнение посвящено тому, чтобы раскрыть неочевидный синтаксис, приводящий к вызову магических методов.

Магический метод	Когда вызывается (пример)	Объяснение
__new__(cls [,...])	instance = MyClass(arg1, arg2)	__new__ вызывается при создании объекта
__init__(self [,...])	instance = MyClass(arg1, arg2)	__init__ вызывается при создании объекта
__cmp__(self, other)	self == other, self > other, etc.	Вызывается для любого сравнения
__pos__(self)	+self	Унарный знак плюса
__neg__(self)	-self	Унарный знак минуса
__invert__(self)	~self	Побитовая инверсия
__index__(self)	x[self]	Преобразование, когда объект используется как индекс
__nonzero__(self)	bool(self), if self:	Булевое значение объекта
__getattr__(self, name)	self.name # name не определено	Пытаются получить несуществующий атрибут
__setattr__(self, name, val)	self.name = val	Присвоение любому атрибуту
__delattr__(self, name)	del self.name	Удаление атрибута
__getattribute__(self, name)	self.name	Получить любой атрибут
__getitem__(self, key)	self[key]	Получение элемента через индекс
__setitem__(self, key, val)	self[key] = val	Присвоение элементу через индекс
__delitem__(self, key)	del self[key]	Удаление элемента через индекс
__iter__(self)	for x in self	Итерация
__contains__(self, value)	value in self, value not in self	Проверка принадлежности с помощью in
__call__(self [,...])	self(args)	«Вызов» объекта
__enter__(self)	with self as x:	with оператор менеджеров контекста
__exit__(self, exc, val, trace)	with self as x:	with оператор менеджеров контекста
__getstate__(self)	pickle.dump(pkl_file, self)	Сериализация

__setstate__(self) data = pickle.load(pkl_file) Сериализация

Эта таблица может избавить от вопросов по поводу синтаксиса вызова магических методов.

Следующие разделы про магические методы основываются на статьях Сергея Балакирева

■ Магический метод __call__

После объявления любого класса:

```
class Counter:
    def __init__(self):
        self.__counter = 0
```

Можно создавать его объекты с помощью оператора:

```
c = Counter()
```

После имени класса - круглые скобки. В общем случае – это оператор вызова. Так, например, можно вызываются функции. Но так можно вызывать и классы. Когда происходит вызов класса, то автоматически запускается магический метод __call__ и в данном случае он создает новый объект — представитель (экземпляр) этого класса.

c = Counter()

__call__(self, *args, **kwargs):

obj = self.__new__(self, *args, **kwargs)

self.__init__(self, *args, **kwargs)

return obj

Это упрощенная схема реализации метода __call__. Всё в действительности несколько сложнее, но принцип тот же: сначала вызывается магический метод __new__ для создания самого объекта в памяти, а затем, метод __init__ - для инициализации этого объекта. То есть, класс можно вызывать подобно функции благодаря встроенной для него реализации магического метода __call__.

А вот объекты (экземпляры классов) так вызывать уже нельзя. То есть, при попытке выполнения оператора

c()

, то возникнет ошибка: «TypeError: 'Counter' object is not callable».

Эта ситуация может быть исправлена. Для этого в классе Counter надо явно прописать магический метод __call__, например, так:

```
class Counter:
```

```
    def __init__(self):
        self.__counter = 0

    def __call__(self, *args, **kwargs):
        print('__call__ is here')
        self.__counter += 1
        return  self.__counter
```

Здесь выводится сообщение, что был вызван метод __call__, затем увеличивается счетчик counter для текущего объекта на 1 и возвращается.

Если снова запустить этот код, никаких ошибок не будет, а в консоли отобразится строка «__call__», что означает вызов и выполнение магического метода __call__. То есть, благодаря добавлению этого магического метода в класс Counter, можно вызывать его экземпляры (обращаться к ним) подобно функциям через оператор круглые скобки. Классы, объекты которых можно вызывать подобно функциям, называются функторами.

В данном случае метод __call__ возвращает значение счетчика, поэтому с объектом можно работать, следующим образом:

```
c = Counter()
c()
c()
res = c()
print(f'{res}')
```

Здесь три раза был вызван метод __call__ и счетчик __counter трижды увеличился на единицу. Поэтому в консоли видно значение 3. Если создать еще один объект-счетчик:

```
c0 = Counter()
c1 = Counter()
c0()
c0()
res0 = c0()

c1()
c1()
res1 = c1()
print(f'{res0}, {res1}')
```

То они будут работать совершенно независимо и подсчитывать число собственных вызовов.

В определении метода __call__ записаны параметры *args, **kwargs. Это значит, что при вызове объектов можно передавать им произвольное количество аргументов. Например, в данном случае можно указать значение изменения счетчика при текущем вызове. Для этого метод __call__ можно переписать , следующим образом:

```
    def __call__(self, step = 1, *args, **kwargs):
        print('__call__ is here')
        self.__counter += step
        return  self.__counter
```

Здесь появился в явном виде первый параметр step с начальным значением 1. То есть, можно вызывать объекты, например, так:

```
c = Counter()
c(2)
c(10)
res = c(-5)
```

```
print(f'{res}')
```

Вот общий принцип работы магического метода __call__. Но здесь остается, как всегда, один важный вопрос: зачем это нужно и где может пригодиться? Несколько примеров его использования.

■ Класс с методом _call_ вместо замыканий функций

Использование класса с методом __call__ вместо замыканий функций. Можно объявить класс StripChars, который бы удалял вначале и в конце строки заданные символы

```
"""
Использование класса с методом __call__ вместо замыканий функций.
Можно объявить класс StripChars, который бы удалял вначале и в конце
строки заданные символы
"""

class StripChars:
    def __init__(self, chars):
        self.__chars = chars

    def __call__(self, *args, **kwargs):
        if not isinstance(args[0], str):
            raise ValueError("Аргумент должен быть строкой")

        return args[0].strip(self.__chars)

# Для этого в инициализаторе сохраняется строка __chars - удаляемые символы,
# а затем, при вызове метода __call__ символы удаляются через строковый метод
# strip для символов __chars. Таким образом, теперь можно создать объект
# (экземпляр) класса и указать те символы, которые следует убирать
s1 = StripChars("?:!.; ")

# А затем вызвать объект s1 подобно функции
res = s1(" Hello World! ")
print(res)
```

В результате объект s1 будет отвечать за удаление указанных символов в начале и конце строки. Но ничего не мешает определять другие объекты этого класса с другим набором символов:

```
s1 = StripChars("?:!.; ")
s2 = StripChars(" ")
res = s1(" Hello World! ")
res2 = s2(" Hello World! ")
print(res, res2, sep='\n')
```

■ Реализация декораторов с помощью классов

Второй пример – это реализация декораторов с помощью классов. Ранее создавались декораторы для вычисления значения производной функции в определенной точке x. Далее - подобная реализация, но с использованием класса.

Сначала — объявление класса:

```
class Derivate:
    def __init__(self, func):
        self.__fn = func
```

```
    def __call__(self, x, dx=0.0001, *args, **kwargs):
        return (self.__fn(x + dx) - self.__fn(x)) / dx
```

Здесь в инициализаторе сохраняется ссылка на декорируемую функцию, а в методе __call__ принимается один обязательный параметр x – точку, где вычисляется производная и dx – шаг изменения при вычислении производной. Далее, определяется функция, например, просто синус:

```
def df_sin(x):
    return math.sin(x)
```

и ее вызов пока без декорирования:

```
print(df_sin(math.pi/4))
```

После запуска программы увидим значение примерно 0.7071. Теперь добавляется декоратор. Это можно сделать двумя способами. Первый, прописать все в явном виде:

```
Df_sin = Derivate(df_sin)
```

Теперь Df_sin – это объект класса Derivate, а не исходная функция. Поэтому, когда она будет вызываться, то запустится метод __call__ и вычислится значение производной в точке math.pi/4.

```
print(Df_sin(math.pi/4))
```

Второй способ – это воспользоваться оператором @ перед объявлением функции:

```
@Derivate
def dDf_sin(x):
    return math.sin(x)

print(dDf_sin(math.pi/4))
```

Получается абсолютно тот же самый результат. Вот принцип создания декораторов функций на основе классов. Все достаточно просто (!!!) – запоминается ссылка на функцию, а затем, расширяется ее функционал в магическом методе __call__.

В реальных задачах, проектах, нужно принимать решения о том, какие механизмы, приемы следует применять для решения конкретных задач. Привести алгоритмы решений на все случаи жизни просто невозможно – это уже навык алгоритмизации, которым должен владеть каждый программист. И вырабатывается он, в основном, на решении практических задач. Так что надо больше практиковаться параллельно с изучением возможностей языка python.

■ Магические методы _str_, _repr_, _len_, _abs_

Каждый магический метод автоматически срабатывает в определенный момент времени, например:

- __str__() – магический метод для отображения информации об объекте класса для пользователей (например, для функций print, str);
- __repr__() – магический метод для отображения информации об объекте класса в режиме отладки (для разработчиков).

Для описания работы этих методов, объявдяется класс для описания собак:

```
class Dog:
    def __init__(self, name):
```

```
        self.name = name

dog = Dog('Мухтар')
print(dog.name)     # имя собачки
print(dog)          # служебная информация об объекте dog
```

Если нужно эту информацию переопределить и отобразить в другом виде (формате), то, как раз для этого и используются магические методы __str__ и __repr__. Для начала - переопределение метода __repr__ , который должен отразится на выводе служебной информации о классе.

```
def __repr__(self):
        return f"{self.__class__}: {self.name}"
```

Этот метод должен возвращать строку, поэтому здесь записан оператор return и формируемая строка. Что именно возвращать — это решение программиста. В данном случае – это название класса и имя собаки.

При переопределении класса Dog и создании объекта-представителя этого класса видна другая информация при его выводе:

class Dog:

```
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f'{self.__class__} : {self.name}'


dog = Dog('Мухтар')
print(dog)
```

Это как раз то, что было определено в магическом методе __repr__. То же самое можно увидеть и при использовании функции print и str. Здесь должен отрабатывать магический метод __str__, но так как он еще не переопределен, то автоматически выполняется метод __repr__.

Добавить второй магический метод __str__ , что бы было видно, как это повлияет на отображение данных:

```
class Dog:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        print('__repr__')
        return f'{self.__class__} : {self.name}'

    def __str__(self):
        print('__str__')
        return f'{self.__class__} = {self.name}'


dog = Dog('Мухтар')
print(dog)
```

В результате, если был один метод, то он и работал. Если два — то по каким правилам они работают, осталось непонятно...

■ Методы _len_ и _abs_

Следующие два магических метода:

- __len__() – позволяет применять функцию len() к объектам класса;
- __abs__() - позволяет применять функцию abs() к объектам класса.

Их использование достаточно простое и очевидное. Для примера можно объявить класс Point, который может хранить произвольный вектор координат точек:

```
class Point:
    def __init__(self, *args):
        self.__coords = args

p = Point(1, 2)

# print(len(p))
# print(len(p.__coords))
```

А далее (по программе) было бы желательно определять размерность координат с помощью функции len(), следующим образом:

```
p = Point(1, 2)
print(len(p))
```

Сейчас это невыполнимо, так как функция len не применима к экземплярам классов по умолчанию. Чтобы изменить это поведение, можно переопределить магический метод __len__() и в данном случае это можно сделать так:

```
    def __len__(self):
        return len(self.__coords)


class Point:
    def __init__(self, *args):
        self.__coords = args

    def __len__(self):
        return len(self.__coords)


p = Point(1, 2)
print(len(p))
```

Здесь возвращается размер списка __coords и если после этого запустить программу, то как раз это значение и будет выведено в консоль. То есть, магический метод __len__ указал, что нужно возвращать, в момент применения функции len() к экземпляру класса.

Следующий магический метод __abs__ работает аналогичным образом, только активируется в момент вызова функции abs для экземпляра класса, например, так:

```
print(abs(p))
```

Опять же, если сейчас выполнить программу, то увидим ошибку, т.к. функция abs не может быть напрямую применена к экземпляру. Но, если переопределить магический метод:

```
    def __abs__(self):
        return list( map(abs, self.__coords) )
```

который возвращает список из абсолютных значений координат точки, то программа отработает в штатном режиме и с ожидаемым результатом.

```
class Point:
    def __init__(self, *args):
        self.__coords = args

    def __len__(self):
        return len(self.__coords)

    def __abs__(self):
        return list(map(abs, self.__coords) )


p = Point(1, 2)
print(len(p))
print(abs(p))
```

■ Магические методы __add__, __sub__, __mul__, __truediv__

Это методы для работы с арифметическими операторами:

- __add__() – для операции сложения;
- __sub__() – для операции вычитания;
- __mul__() – для операции умножения;
- __truediv__() – для операции деления.

Объяснить работу этих методов проще всего на конкретном примере. Предположим, что требуется определить класс для работы со временем. Его экземпляры (объекты) будут хранить часы, минуты и секунды текущего дня. Начальной точкой отсчета будет 00:00 часов ночи. Время будет храниться в виде секунд с максимальным значением 86400 – число секунд в одном дне. Поэтому перед присвоением в инициализаторе будем применяться остаток от деления на это значение:

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
```

```
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY
```

Здесь важно, как записан параметр seconds. После него стоит двоеточие и указан ожидаемый тип данных. Эта нотация языка Python подсказывает программисту, какой тип данных следует передавать в качестве seconds. Конечно, можно передавать и другие типы данных, строгого ограничения здесь нет, это лишь пометка для программиста и не более того. Поэтому далее внутри инициализатора делается проверка, что параметр seconds должен являться целым числом. Если это не так, генерируется исключение TypeError.

Далее в этом же классе определяется метод get_time для получения текущего времени в виде форматной строки:

```
    def get_time(self):
        s = self.seconds % 60            # секунды
        m = (self.seconds // 60) % 60    # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}:{self.__get_formatted(m)}:
{self.__get_formatted(s)}"



    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")
```

Здесь дополнительно определен метод класса для форматирования времени (добавляется незначащий первый ноль, если число меньше 10).

Далее, можно применить этот класс, например, так:

```
c1 = Clock(1000)
print(c1.get_time())
```

Если понадобится изменить время в объекте c1, то сейчас это можно сделать через локальное свойство seconds:

```
c1.seconds = c1.seconds + 100
```

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def get_time(self):
        s = self.seconds % 60  # секунды
        m = (self.seconds // 60) % 60  # минуты
        h = (self.seconds // 3600) % 24  # часы
```

```
        return f"{self.__get_formatted(h)}: {self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")

c1 = Clock(1000)
print(c1.get_time())
c1.seconds = c1.seconds + 100
print(c1.get_time())
```

Было добавлено 100 секунд. Но если бы это изменение можно было бы прописать вот так:

```
c1 = c1 + 100
```

то было бы очень хорошо.

Конечно, при запуске программы возникнет ошибка, так как оператор сложения не работает с экземплярами класса Clock. Однако, это можно поправить, если добавить в этот класс магический метод __add__. Его можно записать в следующем виде:

```
    def __add__(self, other):
        if not isinstance(other, int):
            raise ArithmeticError("Правый операнд должен быть типом int")
        return Clock(self.seconds + other)
```

И теперь, при запуске программы, все работает так, как и задумывалось...

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def __add__(self, other):
        if not isinstance(other, int):
            raise ArithmeticError("Правый операнд должен быть типом int")
        return Clock(self.seconds + other)

    def get_time(self):
        s = self.seconds % 60  # секунды
        m = (self.seconds // 60) % 60  # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}: {self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")

c1 = Clock(1000)
print(c1.get_time())
```

```
c1.seconds = c1.seconds + 100
print(c1.get_time())
c1 = c1 + 100
print(c1.get_time())
```

Вначале есть объект класса Clock со значением секунд 1000. Затем, арифметическая операция

c1 = c1 + 100 фактически означает выполнение команды:

```
c1 = c1.__add__(100)
```

В результате активируется метод __add__ и параметр other принимает целочисленное значение 100. Проверка проходит и формируется новый объект класса Clock со значением секунд 1000+100 = 1100. Этот объект возвращается методом __add__ и переменная c1 начинает ссылаться на этот новый экземпляр класса. На прежний уже не будет никаких внешних ссылок, поэтому он будет автоматически удален сборщиком мусора.

Сложность процессов, когда всего лишь нужно прибавить 100 секунд к уже имеющемуся значению может удивить. Но эта сложность оправданна. Чтобы это понять, нужно расширить функционал оператора сложения и допустить, что можно складывать два разных объекта класса Clock, следующим образом:

```
c1 = Clock(1000)
c2 = Clock(2000)
c3 = c1 + c2
print(c3.get_time())
```

Конечно, если сейчас запустить программу, то возбудится исключение ArithmeticError, так как параметр other не соответствует целому числу. Это можно поправить и немного изменить реализацию метода __add__:

```
    def __add__(self, other):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть типом int или
объектом Clock")

        sc = other if isinstance(other, int) else other.seconds
        return Clock(self.seconds + sc)
```

Теперь в программе можно складывать и отдельные целые числа и объекты классов Clock. И это удобно. Кроме того, можно прописывать и более сложные конструкции при сложении, например, такие:

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def __add__(self, other):
        if not isinstance(other, (int, Clock)):
```

```
            raise ArithmeticError("Правый операнд должен быть типом int или
объектом Clock")
        sc = other if isinstance(other, int) else other.seconds
        return Clock(self.seconds + sc)

    def get_time(self):
        s = self.seconds % 60  # секунды
        m = (self.seconds // 60) % 60  # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}: {self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")

c1 = Clock(1000)
print(c1.get_time())
c1.seconds = c1.seconds + 100
print(c1.get_time())
c1 = c1 + 100
print(c1.get_time())
c2 = Clock(1000)
c3 = Clock(2000)
c4 = Clock(3000)
c5 = c2 + c3 + c4
print(c5.get_time())
```

И она сработала благодаря тому, что метод __add__ возвращает каждый раз новый экземпляр класса Clock. Детальнее все выглядит так:

Сначала идет сложение объектов c1 + c2, в результате формируется новый объект класса Clock со значением секунд 1000 + 2000 = 3000. Пусть на этот класс ведет внутренняя переменная t1. Затем, для этого нового объекта вызывается снова метод __add__ и идет сложение с объектом t1 + c3. Получаем еще один объект с числом секунд 6000. На этот объект, как раз и будет ссылаться переменная c4, а объект с t1 будет автоматически уничтожен сборщиком мусора.

Если бы не создавались экземпляры классов Clock в методе __add__ и не возвращались, то конструкцию с двумя сложениями было бы невозможно реализовать.

Еще одним важным нюансом работы оператора сложения для объектов классов, является порядок их записи. Мы всегда прописывали его в виде:

```
c1 = c1 + 100
```

то есть, сначала шел объект, а затем, число. Если записать наоборот, то возникнет ошибка:

```
c1 = 100 + c1
```

и это очевидно, так как здесь, фактически идет вызов метода:

```
100.__add__(c1)
```

но он не существует для объекта int и экземпляров класса Clock. Как выйти из этой ситуации? Очень просто. Язык python предоставляет специальный набор магических методов с добавлением буквы r:

```
__radd__()
```

Он автоматически вызывается, если не может быть вызван метод __add__(). Его определение можно добавить в класс Clock:

```
    def __radd__(self, other):
        return self + other
```

Здесь записана команда сложения текущего объекта класса Clock с параметром other, который может быть или числом или тоже объектом класса Clock. В свою очередь будет вызван метод __add__, но с правильным порядком типов данных, поэтому сложение пройдет без ошибок.

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def __add__(self, other):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть типом int или
объектом Clock")
        sc = other if isinstance(other, int) else other.seconds
        return Clock(self.seconds + sc)

    def __radd__(self, other):
        return self + other


    def get_time(self):
        s = self.seconds % 60  # секунды
        m = (self.seconds // 60) % 60  # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}: {self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")

c1 = Clock(1000)
print(c1.get_time())
c1.seconds = c1.seconds + 100
print(c1.get_time())
c1 = c1 + 100
print(c1.get_time())
c1 = 100 + c1
print(c1.get_time())
c2 = Clock(1000)
c3 = Clock(2000)
c4 = Clock(3000)
c5 = c2 + c3 + c4
print(c5.get_time())
```

Наконец, у всех магических методов, связанных с арифметическими операторами, есть еще одна модификация с первой буквой i:

```
__iadd__()
```

Она вызывается для команды:

```
c1 += 100
```

Если запустить сейчас программу, то никаких ошибок не будет и отработает метод __add__(). Но в методе __add__ создается новый объект класса Clock, тогда как при операции += этого делать не обязательно. Поэтому в класс Clock надо добавить еще один магический метод __iadd__:

```
    def __iadd__(self, other):
        print("__iadd__")
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть типом int или
объектом Clock")

        sc = other if isinstance(other, int) else other.seconds
        self.seconds += sc

        return self
```

Здесь не создаётся нового объекта, а меняется число секунд в текущем. Это логичнее, так как вызывать цепочкой операцию += не предполагается и, кроме того, она изменяет (по смыслу) состояние текущего объекта. Поэтому и добавляется этот магический метод.

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def __add__(self, other):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть типом int или
объектом Clock")
        sc = other if isinstance(other, int) else other.seconds
        return Clock(self.seconds + sc)

    def __radd__(self, other):
        return self + other

    def __iadd__(self, other):
        print("__iadd__")
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть типом int или
объектом Clock")

        sc = other if isinstance(other, int) else other.seconds
        self.seconds += sc

        return self

    def get_time(self):
        s = self.seconds % 60  # секунды
        m = (self.seconds // 60) % 60  # минуты
        h = (self.seconds // 3600) % 24  # часы
```

```
        return f"{self.__get_formatted(h)}: {self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")

c1 = Clock(1000)
print(c1.get_time())
c1.seconds = c1.seconds + 100
print(c1.get_time())
c1 = c1 + 100
print(c1.get_time())
c1 = 100 + c1
print(c1.get_time())
c1 += 100
print(c1.get_time())
c2 = Clock(1000)
c3 = Clock(2000)
c4 = Clock(3000)
c5 = c2 + c3 + c4
print(c5.get_time())
c1 += c2
print(c1.get_time())
```

Вот и была подробно рассмотрена работа одного арифметического магического метода __add__() с его вариациями __radd__() и __iadd__(). По аналогии используются и все остальные подобные магические методы:

Оператор	Метод оператора	Оператор	Метод оператора
x + y	__add__(self, other)	x += y	__iadd__(self, other)
x — y	__sub__(self, other)	x -= y	__isub__(self, other)
x * y	__mul__(self, other)	x *= y	__imul__(self, other)
x / y	__truediv__(self, other)	x /= y	__itruediv__(self, other)
x // y	__floordiv__(self, other)	x //= y	__ifloordiv__(self, other)
x % y	__mod__(self, other)	x %= y	__imod__(self, other)

■ Методы сравнений __eq__, __ne__, __lt__, __gt__ и другие

Магические методы для реализации операторов сравнения:

- __eq__() – для равенства ==
- __ne__() – для неравенства !=
- __lt__() – для оператора меньше <
- __le__() – для оператора меньше или равно <=
- __gt__() – для оператора больше >
- __ge__() – для оператора больше или равно >=

Работа этих методов будет рассматриваться на примере ранее определённого класса Clock:

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def get_time(self):
        s = self.seconds % 60            # секунды
        m = (self.seconds // 60) % 60    # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}:{self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")
```

Изначально для класса реализован только один метод сравнения на равенство, например:

```
c1 = Clock(1000)
c2 = Clock(1000)
print(c1 == c2)
```

Но здесь объекты сравниваются по их id (адресу в памяти), а хотелось, чтобы сравнивались секунды в каждом из объектов c1 и c2. Для этого следующим образом переопределяется магический метод __eq__():

```
    def __eq__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return self.seconds == sc
```

Теперь, после запуска программы видим значение True, т.к. объекты содержат одинаковое время.

```
class Clock:
    __DAY = 86400  # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def get_time(self):
        s = self.seconds % 60  # секунды
        m = (self.seconds // 60) % 60  # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}:{self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    def __eq__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")
```

```
        sc = other if isinstance(other, int) else other.seconds
        return self.seconds == sc

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")

c1 = Clock(1000)
c2 = Clock(1000)
print(c1 == c2)
```

Кроме того, можно выполнять проверку и на неравенство:

```
print(c1 != c2)
```

Если интерпретатор языка Python НЕ находит определение метода ==, то он пытается выполнить противоположное сравнение с последующей инверсией результата. То есть, в данном случае находится оператор == и выполняется инверсия:

```
not (a == b)
```

В этом можно убедиться, поставив точку останова в метод __eq__ и запустить программу. Он срабатывает и результат в последствии меняется на противоположный.

Объекты класса Clock, сравниваются, на равенство и неравенство, а также с целыми числами. Однако, сравнение на больше или меньше пока не работает. Строчка программы:

```
print(c1 < c2)
```

приведет к ошибке. Поэтому надо добавить эту операцию сравнения:

```
    def __lt__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return self.seconds < sc
```

Как здесь получается дублирование кода. Поэтому нужно, вынести общее для методов сравнения в отдельный метода класса:

```
    @classmethod
    def __verify_data(cls, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")

        return other if isinstance(other, int) else other.seconds
```

А сами методы примут вид:

```
    def __eq__(self, other):
        sc = self.__verify_data(other)
        return self.seconds == sc
```

```
    def __lt__(self, other):
        sc = self.__verify_data(other)
        return self.seconds < sc
```

Итак, мы определили сравнение на равенство и меньше. Теперь, можно сравнивать объекты класса Clock на эти операции и дополнительно на неравенство и больше. Сейчас команда:

```
class Clock:
    __DAY = 86400  # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def get_time(self):
        s = self.seconds % 60  # секунды
        m = (self.seconds // 60) % 60  # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}:{self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    @classmethod
    def __verify_data(cls, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")

        return other if isinstance(other, int) else other.seconds

    def __eq__(self, other):
        sc = self.__verify_data(other)
        return self.seconds == sc

    def __lt__(self, other):
        sc = self.__verify_data(other)
        return self.seconds < sc

    # def __eq__(self, other):
    #     if not isinstance(other, (int, Clock)):
    #         raise TypeError("Операнд справа должен иметь тип int или
Clock")
    #
    #     sc = other if isinstance(other, int) else other.seconds
    #     return self.seconds == sc
    #
    # def __lt__(self, other):
    #     if not isinstance(other, (int, Clock)):
    #         raise TypeError("Операнд справа должен иметь тип int или
Clock")
    #
    #     sc = other if isinstance(other, int) else other.seconds
    #     return self.seconds < sc

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")
```

```
c1 = Clock(1000)
c2 = Clock(1000)
print(c1 == c2)

c1 = Clock(1000)
c2 = Clock(2000)
print(c1 < c2)
```

Выдаст True, так как первое время меньше второго. И также можно делать проверку на больше:

```
print(c1 > c2)
```

Здесь сработает тот же метод меньше, но для объекта c2:

```
c2 < c1
```

То есть, в отличие от оператора ==, где применяется инверсия, здесь меняется порядок операндов. Но при этом надо определить в классе метод больше:

```
    def __gt__(self, other):
        sc = self.__verify_data(other)
        return self.seconds > sc
```

Он будет найден и выполнен. Подмена происходит только в случае отсутствия соответствующего магического метода.

И то же самое для методов сравнения на меньше или равно и больше или равно:

```
    def __le__(self, other):
        sc = self.__verify_data(other)
        return self.seconds <= sc
```

Если его вызвать непосредственно для объектов класса:

```
print(c1 <= c2)
```

то он сработает и результат отобразится в консоли. Но, если прописать обратное сравнение:

```
print(c1 >= c2)
```

то просто изменится порядок операндов и будет взято все то же сравнение меньше или равно.

То есть, для определения операций сравнения достаточно в классе определить только три метода: ==, <, <=, если остальные являются их симметричной противоположностью. В этом случае язык python сам подберет нужный метод и выполнит его при сравнении объектов.

```
class Clock:
    __DAY = 86400  # число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
```

```
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def get_time(self):
        s = self.seconds % 60  # секунды
        m = (self.seconds // 60) % 60  # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}:{self.__get_formatted(m)}:
{self.__get_formatted(s)}"

    @classmethod
    def __verify_data(cls, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")

        return other if isinstance(other, int) else other.seconds

    def __eq__(self, other):
        sc = self.__verify_data(other)
        return self.seconds == sc

    def __lt__(self, other):
        sc = self.__verify_data(other)
        return self.seconds < sc

    def __gt__(self, other):
        sc = self.__verify_data(other)
        return self.seconds > sc

    def __le__(self, other):
        sc = self.__verify_data(other)
        return self.seconds <= sc

    def __ge__(self, other):
        sc = self.__verify_data(other)
        return self.seconds >= sc

    # def __eq__(self, other):
    #     if not isinstance(other, (int, Clock)):
    #         raise TypeError("Операнд справа должен иметь тип int или
Clock")
    #
    #     sc = other if isinstance(other, int) else other.seconds
    #     return self.seconds == sc
    #
    # def __lt__(self, other):
    #     if not isinstance(other, (int, Clock)):
    #         raise TypeError("Операнд справа должен иметь тип int или
Clock")
    #
    #     sc = other if isinstance(other, int) else other.seconds
    #     return self.seconds < sc

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")

c1 = Clock(1000)
```

```
c2 = Clock(1000)
print(c1 == c2)
```

■ Магические методы _eq_ и _hash_

Вычисления хеша для объектов классов. Вначале что это такое и зачем нужно? В python имеется специальная функция:

```
hash(123)
hash("Python")
hash((1, 2, 3))
```

которая формирует по определенному алгоритму целочисленные значения для неизменяемых объектов. Причем, для равных объектов на выходе всегда должны получаться равные хэши:

```
hash("Python")
hash((1, 2, 3))
```

А вот обратное утверждение делать нельзя: равные хэши не гарантируют равенство объектов. Это, как в известном выражении: камбала – это рыба, но не каждая рыба камбала. С хэшами все то же самое.

Однако, если хеши не равны, то и объекты точно не равны. Получаются следующие свойства для хеша:

- Если объекты a == b (равны), то равен и их хэш.
- Если равны хеши: hash(a) == hash(b), то объекты могут быть равны, но могут быть и не равны.
- Если хеши не равны: hash(a) != hash(b), то объекты точно не равны.

Причем, хэши можно вычислять только для неизменяемых объектов. Например, для списков:

```
hash([1, 2, 3])
```

получается ошибка «unhashable type» - не хэшируемый объект.

Таким образом, для любого неизменяемого объекта можно вычислять хэш с помощью функции hash(), но зачем все это надо? В действительности некоторые объекты в Python, например, словари используют хэши в качестве своих ключей. Так, когда у словаря указывается ключ, то он должен относиться к неизменяемому типу данных:

```
d = {}
d[5] = 5
d["python"] = "python"
d[(1, 2, 3)] = [1, 2, 3]
```

В действительности, это необходимо, чтобы можно было вычислить хеш объектов и ключи хранить в виде:

```
(хэш ключа, ключ)
```

Для чего это понадобилось? Дело в том, что первоначально нужная запись в словаре ищется по хэшу, так как существует быстрый алгоритм поиска нужного значения хэша. А затем, для равных хешей (если такие были обнаружены), отбирается запись с указанным в ключе объекте. Такой подход значительно ускоряет поиск значения в словаре.

Теперь объявляется класс Point для представления координат на плоскости:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Для экземпляров этого класса:

```
p1 = Point(1, 2)
p2 = Point(1, 2)
```

можно вычислять хеш:

```
print(hash(p1), hash(p2), sep='\n')
```

Важно, что несмотря на то, что координаты точек p1 и p2 равны, их хэши разные. То есть, с точки зрения функции hash() – это два разных объекта. Но как она понимает, равные объекты или разные? Если оператор сравнения:

```
print(p1 == p2)
```

дает True, то объекты равны, иначе – не равны. Соответственно, для разных объектов будут получаться и разные хэши. Но раз это так, если переопределить поведение этого оператора сравнения с помощью магического метода __eq__()?

```
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

Теперь у нас объекты с одинаковыми координатами будут считаться равными. Но при запуске программы возникает ошибка «unhashable type», то есть, наши объекты стали не хэшируемыми. Да, как только происходит переопределение оператора ==, то начальный алгоритм вычисления хэша для таких объектов перестает работать. Поэтому, нам здесь нужно прописать свой способ вычисления хэша объектов через магический метод __hash__(), например, так:

```
    def __hash__(self):
        return hash((self.x, self.y))
```

Здесь вызывается функция hash для кортежа из координат точки. Этот кортеж относится к неизменяемому типу, поэтому для него можно применить стандартную функцию hash(). То есть, подменяется вычисление хэша объекта класса Point на вычисление хэша от координат точки. Теперь, после запуска программы видно, что объекты равны и их хэши также равны.

Что это в итоге означает? Если взять пустой словарь:

```
d = {}
```

А затем сформировать записи через объекты p1 и p2:

```
d[p1] = 1
d[p2] = 2
print(d)
```

то они будут восприниматься как один и тот же ключ, так как объекты равны и их хэши тоже равны. А вот если магические методы в классе Point поставить в комментарии и снова запустить программу, то увидим, что это уже разные объекты, которые формируют разные ключи словаря. Вот для чего может понадобиться тонко настраивать работу функции hash() для объектов классов.

■ Магический метод __bool__ определения правдивости объектов

О способах настройки и определения правдивости объектов классов. Что такое правдивость? Это когда к экземпляру явно или неявно применяется функция bool(). С ней мы с вами уже знакомы и применяли к обычным типам данных:

```
bool(123)
bool(-1)
bool(0)
bool("python")
bool("")
bool([])
```

В стандартном поведении она возвращает True для непустых объектов и False – для пустых. Что функцию bool() будет выдавать для экземпляров классов. Для этого используется ранее объявленный класс Point:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Его объект:

```
p = Point(3, 4)
```

и применение к нему функции bool():

```
print(bool(p))
```

Даёт значение True. В действительности, эта функция всегда возвращает True для любых объектов пользовательского класса. Получается, что смысла в ней особого нет, применительно к экземплярам классов? Не совсем. Можно переопределить ее поведение либо через магический метод __len__(), либо через метод __bool__():

- __len__() – вызывается функцией bool(), если не определен магический метод __bool__();
- __bool__() – вызывается в приоритетном порядке функцией bool().

Вначале прописывается магический метод __len__() в классе Point, следующим образом:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        print("__len__")
        return self.x * self.x + self.y * self.y
```

В этом методе вычисляется и возвращается квадрат длины радиус-вектора с координатами (x; y). Запустим программу и видим значение True, а также сообщение «__len__». То есть, действительно был вызван метод __len__() и, так как он вернул не нулевое значение, то функция bool() интерпретировала его как True.

Чтобы длина вектора была нулевой, в объекте класса прописываются нулевые координаты:

```
p = Point(0, 0)
```

И теперь получается ожидаемое значение False.

Конечно, если нужно явно описать алгоритм работы функции bool() применительно к экземплярам класса, то следует использовать магический метод __bool__(). Как то так:

```
    def __bool__(self):
        print("__bool__")
        return self.x == self.y
```

Теперь, объект будет считаться правдивым (истинным), если его координаты равны. При запуске программы для нулей отображается значение True. Если же прописать не равные координаты:

```
p = Point(10, 20)
```

то получается значение False. Конечно, такая реализация магического метода __bool__() – это лишь учебный пример для описания принципа его работы. В реальности, в этом методе можно прописывать любую логику. Единственное условие, чтобы данный метод возвращал булево значение True или False. Указывать в операторе return другие типы данных запрещено.

Все это хорошо, но где это используется? Чаще всего в условных конструкциях. Например, если прописать вот такое условие:

```
if p:
    print("объект p дает True")
else:
    print("объект p дает False")
```

Здесь происходит неявный вызов функции bool() при проверке условия. Поэтому в программах, где требуется описать собственные проверки истинности или ложности объектов, то пользуются или магическим методом __len__(), но чаще всего, магическим методом __bool__().

■ Магические методы _getitem_, _setitem_ и _delitem_

Здесь будет описан следующий наборе магических методов:

- __getitem__(self, item) – получение значения по ключу item;
- __setitem__(self, key, value) – запись значения value по ключу key;
- __delitem__(self, key) – удаление элемента по ключу key.

Для чего они нужны и как их можно использовать. Предполагается, что объявлен класс для представления студентов:

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = list(marks)
```

Его экземпляр можно сформировать, следующим образом:

```
stud_1 = Student('Boris', [2,3,4,5,5,1])
```

В объекте s1 имеется локальное свойство marks со списком отметок студента. К нему можно обратиться и выбрать любую оценку:

```
print(stud_1.marks[2])
```

Но что если нужно сделать то же самое, но используя только ссылку на объект s1:

```
print(stud_1[2])
```

Если сейчас запустить программу, то появится сообщение об ошибке, что класс (объект) не поддерживает такой синтаксис. Это можно поправить с применением магического метода __getitem__. Его объявление в классе Student:

```
    def __getitem__(self, item):
        return self.marks[item]
```

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = list(marks)

    def __getitem__(self, item):
        return self.marks[item]

stud_1 = Student('Boris', [2,3,4,5,5,1])
print(stud_1.marks[2])
print(stud_1[3])
```

Теперь ошибок нет и на экране видны соответствующие значения. Однако, если указать неверный индекс:

```
print(s1[20])
```

то возникнет исключение IndexError, которое сгенерировал список marks. При необходимости, можно контролировать эту ошибку, если в методе __getitem__ прописать проверку:

```
    def __getitem__(self, item):
        if 0 <= item < len(self.marks):
            return self.marks[item]
        else:
            raise IndexError("Неверный индекс")
```

При запуске программы видно сообщение «Неверный индекс». Также можно сделать проверку на тип индекса:

```
print(s1['abc'])
```

для списков он должен быть целым числом. Поэтому дополнительно можно записать такую проверку:

```
    def __getitem__(self, item):
        if not isinstance(item, int):
            raise TypeError("Индекс должен быть целым числом")

       if 0 <= item < len(self.marks):
            return self.marks[item]
        else:
            raise IndexError("Неверный индекс")
```

То есть, здесь возможны самые разные вариации обработки и проверки исходных данных, прежде чем обратиться к списку marks и вернуть значение.

Пусть теперь требуется реализовать возможность менять оценки студентов, используя следующий синтаксис:

```
s1[2] = 4
print(s1[2])
```

Сейчас, после запуска программы будет ошибка TypeError, что объект не поддерживает операцию присвоения, так как в классе не реализован метод __setitem__. Его реализация:

```
    def __setitem__(self, key, value):
        if not isinstance(key, int) or key < 0:
            raise TypeError("Индекс должен быть целым неотрицательным
числом")

        self.marks[key] = value
```

Однако, если указать несуществующий индекс:

```
s1[6] = 4
```

то операция присвоения новой оценки приведет к ошибке. И если предполагается использовать такую возможность, то реализовать ее можно, следующим образом:

```
    def __setitem__(self, key, value):
        if not isinstance(key, int) or key < 0:
            raise TypeError("Индекс должен быть целым неотрицательным
числом")

        if key >= len(self.marks):
            off = key + 1 - en(self.marks)
            self.marks.extend([None]*off)

        self.marks[key] = value
```

Если индекс превышает размер списка, то он расширяется значениями None до нужной длины (с помощью метода extend), а затем, в последний элемент в него записывается переданное значение value. Теперь, при выполнении команд:

```
s1[10] = 4
print(s1.marks)
```

Получается список:

```
[5, 5, 3, 2, 5, None, None, None, None, None, 4]
```

То есть, он был расширен до 10 элементов и последним элементом записано 4. И так можно прописывать любую нужную нам логику при записи новых значений в список marks.

Наконец, последний третий магический метод __delitem__ вызывается при удалении элемента из списка. Если сейчас записать команду:

```
del s1[2]
```

то в консоли появится сообщение: «AttributeError: __delitem__». Здесь явно указывается, что при удалении вызывается метод __delitem__. Добавить его в класс:

```
    def __delitem__(self, key):
        if not isinstance(key, int):
            raise TypeError("Индекс должен быть целым числом")

        del self.marks[key]
```

Теперь оценки успешно удаляются, если указан верный индекс. Таковы общие возможности данных магических методов.

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = list(marks)

    # def __getitem__(self, item):
    #     return self.marks[item]

    # def __getitem__(self, item):
    #     if 0 <= item < len(self.marks):
    #         return self.marks[item]
    #     else:
    #         raise IndexError("Неверный индекс")

    def __getitem__(self, item):
```

```
        if not isinstance(item, int):
            raise TypeError("Индекс должен быть целым числом")

        if 0 <= item < len(self.marks):
            return self.marks[item]
        else:
            raise IndexError("Неверный индекс")

    def __setitem__(self, key, value):
        if not isinstance(key, int) or key < 0:
            raise TypeError("Индекс должен быть целым неотрицательным
числом")

        if key >= len(self.marks):
            off = key + 1 - len(self.marks)
            self.marks.extend([None] * off)

        self.marks[key] = value

    def __delitem__(self, key):
        if not isinstance(key, int):
            raise TypeError("Индекс должен быть целым числом")

        del self.marks[key]


stud_1 = Student('Boris', [2,3,4,5,5,1])
print(stud_1.marks[2])
print(stud_1[3])
#print(stud_1[25])
#print(stud_1['qwerty'])

print(stud_1[1])
stud_1[1] = 5
print(stud_1[1])

stud_1[10] = 5
print(stud_1[9])
print(stud_1[10])
print(stud_1.marks)

del stud_1[7]
print(stud_1.marks)
```

Аргумент item метода __getitem__ может иметь любое значение:

```
class XYZ:
# инициализация атрибутов объекта аргументом
# из одного списка: явное преобразование к списку
# и присвоение значений списка атрибутам объекта
    def __init__(self, xyz):
        xyzData = list(xyz)
        self.x = xyzData[0]
        self.y = xyzData[1]
        self.z = xyzData[2]

# аргумент метода __getitem__ - стока
```

```
    def __getitem__(self, item):
        if item == 'xyz':
            return [self.x, self.y, self.z]

# инициализация объекта xyz_ray списком
xyz_ray = XYZ([1, 2, 3])
# значение аргумента item метода __getitem__ -
# список строк, состоящих из одной строки 'xyz'
ret = xyz_ray['xyz']
print(f'{ret[2]}')
```

■ Магические методы _iter_ и _next_

Методы:

- __iter__(self) – получение итератора для перебора объекта;
- __next__(self) – переход к следующему значению и его считывание.

Для чего они нужны и как их можно использовать. Известно, как работает функция range(). Она выдает значения арифметической прогрессии, например:

```
list(range(5))
```

дает последовательность целых чисел от 0 до 4. Перебрать значения объекта range также можно через итератор:

```
a = iter(range(5))
next(a)
next(a)
…
```

В конце генерируется исключение StopIteration. Так вот, можно создать подобный объект, используя магические методы __iter__ и __next__. И это будет сделано для объекта frange, который будет выдавать последовательность вещественных чисел арифметической прогрессии. Для этого объявляется класс:

```
class FRange:
    def __init__(self, start=0.0, stop=0.0, step=1.0):
        self.start = start
        self.stop = stop
        self.step = step
        self.value = self.start – self.step
```

Здесь в инициализатор передаётся начальное значение прогрессии, конечное и шаг изменения. Также формируется свойство value, которое будет представлять собой текущее значение для считывания.

Для перебора элементов в этот класс добавляется метод, который будет соответствовать магическому методу __next__:

```
    def __next__(self):
        if self.value + self.step < self.stop:
            self.value += self.step
            return self.value
        else:
            raise StopIteration
```

В этом методе увеличивается значение value на шаг step и возвращатся до тех пор, пока не будет достигнуто значения stop (не включая его). При достижении конца будет сгенерировано исключение StopIteration, ровно так, как это делает объект range.

Объект этого класса:

```
fr = FRange(0, 2, 0.5)
```

четыре раза вызывается метод __next__()

```
print(fr.__next__())
print(fr.__next__())
print(fr.__next__())
print(fr.__next__())
```

Видны четыре значения арифметической прогрессии. Но если вызвать __next__() еще раз:

```
print(fr.__next__())
```

то будет сгенерировано исключение StopIteration. В целом получился неплохой учебный пример. Благодаря возможности определения магического метода __next__ в классе FRange, можно применять функцию next() для перебора значений его объектов:

```
fr = FRange(0, 2, 0.5)
print(next(fr))
print(next(fr))
print(next(fr))
print(next(fr))
```

Здесь функция next() вызывает метод __next__ и возвращенное им значение, возвращается функцией next(). При этом, в качестве аргумента ей передаётся экземпляр самого класса. То есть, объект класса выступает в роли итератора. В этом случае так и задумывалось. Однако, перебрать объект fr с помощью цикла for не получится:

```
for x in fr:
    print(x)
```

Появится ошибка, что объект не итерируемый. И это не смотря на то, что было прописано поведение функции next(). Оказывается, этого не достаточно. Необходимо еще, чтобы объект возвращал итератор при вызове функции iter:

```
it = iter(fr)
```

Для этого в классе нужно прописать еще один магический метод __iter__. В этом примере он будет выглядеть, так:

```
    def __iter__(self):
        self.value = self.start - self.step
        return self
```

Здесь устанавливается начальное значение value и возвращается ссылка на объекта класса, так как этот объект в данном примере и есть итератор – через него вызывается магический метод __next__.

Теперь, после запуска программы не возникает никаких ошибок и цикл for перебирает значения объекта fr. То же самое можно сделать и через next():

```
fr = FRange(0, 2, 0.5)
```

```
it = iter(fr)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

Цикл for именно так и перебирает итерируемые объекты, сначала неявно вызывает функцию iter(), а затем, на каждой итерации – функцию next(), пока не возникнет исключение StopIteration. Кроме того, благодаря магическому методу __iter__ теперь можно обходить значения объекта fr много раз с самого начала, например:

```
it = iter(fr)
print(next(it))
print(next(it))
print(next(it))
print(next(it))

it = iter(fr)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

Таким образом, сформировали класс FRange, который воспринимается как итерируемый объект с возможностью перебора функцией next() или циклом for.

И ещё один пример. Это класс FRange2D для формирования таблиц значений:

```
class FRange2D:
    def __init__(self, start=0.0, stop=0.0, step=1.0, rows=5):
        self.fr = FRange(start, stop, step)
        self.rows = rows
```

Здесь в инициализаторе создается одномерный объект FRange, который будет формировать строки таблицы. Параметр rows – число строк. Далее, пропишем два магических метода __iter__ и __next__, следующим образом:

```
    def __iter__(self):
        self.value_row = 0
        return self

    def __next__(self):
        if self.value_row < self.rows:
            self.value_row += 1
            return iter(self.fr)
        else:
            raise StopIteration
```

Важно, что метод __next__ возвращает не конкретное значение, а итератор на объект класса FRange. Создаётся объект класса FRange2D:

```
fr = FRange2D(0, 2, 0.5, 4)
```

и для перебора его значений нам понадобятся два цикла for:

```
for row in fr:
    for x in row:
        print(x, end=" ")
    print()
```

Первый цикл перебирает первый итератор – объект класса FRange2D и на каждой итерации возвращает итератор объекта класса FRange. Именно поэтому в методе __next__ класса FRange2D возвращается иетратор, иначе нельзя было бы перебирать объект row во вложенном цикле for.

После запуска программы на экране появляется следующая таблицу чисел:

```
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
```

Здесь в инициализаторе создается одномерный объект FRange, который будет формировать строки таблицы. Параметр rows – число строк. Далее, пропишем два магических метода __iter__ и __next__, следующим образом:

```
    def __iter__(self):
        self.value_row = 0
        return self

    def __next__(self):
        if self.value_row < self.rows:
            self.value_row += 1
            return iter(self.fr)
        else:
            raise StopIteration
```

Важно, что метод __next__ возвращает не конкретное значение, а итератор на объект класса FRange. Далее создаётся объект класса FRange2D:

```
fr = FRange2D(0, 2, 0.5, 4)
```

и для перебора его значений нам понадобятся два цикла for:

```
for row in fr:
    for x in row:
        print(x, end=" ")
    print()
```

Первый цикл перебирает первый итератор – объект класса FRange2D и на каждой итерации возвращает итератор объекта класса FRange. Именно поэтому мы в методе __next__ класса FRange2D возвращаем иетратор, иначе бы не смогли перебирать объект row во вложенном цикле for.

После запуска программы увидим на экране следующую таблицу чисел:

```
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
```

Таков общий принцип создания итерируемых объектов.

■ Дополнение 2: Изменения в python 3

Несколько главных случаев, когда python 3 отличается от 2.x в терминах его объектной модели:

Так как в python 3 различий между строкой и юникодом больше нет, __unicode__ исчез, а появился __bytes__ (который ведёт себя так же как __str__ и __unicode__ в 2.7) для новых встроенных функций построения байтовых массивов.

Так как деление в python 3 теперь по-умолчанию «правильное деление», __div__ больше нет.

__coerce__ больше нет, из-за избыточности и странного поведения.

__cmp__ больше нет, из-за избыточности.

__nonzero__ было переименовано в __bool__.

next у итераторов был переименован в __next__.

■ События и программирование событий

Что такое структурное и объектно-ориентированное программирование предполагается известным.

Событийно-ориентированное программирование ориентировано на события. То есть та или иная часть программного кода начинает выполняться лишь тогда, когда случается то или иное событие.

Событийно-ориентированное программирование основывается на структурном и объектно-ориентированном.

Программы с графическим интерфейсом пользователя (GUI-порграммы) являются событийно-ориентированными.

■ Событийно-ориентированное программирование

В python реализованы разные события:

- сработал временной фактор,
- кто-то кликнул мышкой или нажал Enter,
- кто-то начал вводить текст,
- кто-то переключил радиокнопки,
- кто-то прокрутил страницу вниз
- и т. д.

Когда что-то подобное происходит, это можно рассматривать как событие. Если был создан соответствующий обработчик события, происходит срабатывание определенной части программного кода, что приводит к какому-либо результату.

■ Применение событий

Даже если не создаются собственные классы и объекты, классы в программе всё равно используются. Например, виджеты – это объекты, порожденные встроенными классами.

Пусть в программе определено текстовое поле для ввода текста. Этот текст при нажатии на кнопку (событие) разбивается на список слов, которые сортируются по алфавиту и выводятся в метке. Выполняющий все это код располагается в функции str_to_sort_list:

```
def str_to_sort_list(event):
    s = ent.get()
    s = s.split()
    s.sort()
    lab['text'] = ' '.join(s)
```

В функции str_to_sort_list (обработка события) с помощью метода get из поля забирается текст, представляющий собой строку. Она преобразуется в список слов с помощью метода split. Далее этот список сортируется. В конце изменяется свойство text метки. Ему присваивается строка, полученная из списка с помощью строкового метода join.

Функции обработки события могут быть связаны с событием при помощи метода bind, что обеспечивает вызов функции при наступлении события. У таких функций должен быть один обязательный параметр. Обычно его называют event, то есть "событие". С помощью метода bind функция обработки события связывается с событием. Методу bind передаётся событие (событие 'щелчок левой кнопкой мыши' - обозначается строкой '<Button-1>' и функция-обработчик

```
but.bind('<Button-1>', str_to_sort_list)
```

По щелчку левой кнопкой мыши по виджету but будет вызвана функция обработки события str_to_sort_list, параметру event будет присвоено соответствующее значение.

При разработке приложения (все вопросы - к дизайнерам!) предполагается, что виджеты не разбросаны по окну как попало, а рационально сгруппированы, интерфейс продуман и обычно подчинен определенным правилам .

Пока же виджеты будут располагаться в окне приложения с помощью простого менеджера геометрии tkinter (метода pack) друг под другом.

```
ent.pack()
but.pack()
lab.pack()
```

Далее в примере метод mainloop объекта Tk запускает главный цикл обработки событий, что и приводит к отображению главного окна приложения со всеми "упакованными" на нем виджетами:

```
root.mainloop()
```

Полный текст приложения:

```
from tkinter import *

# Функция str_to_sort_list. ================================================
# При нажатии на кнопку обеспечивает разбиение текста,
# вводимого в текстовое поле, на слова, из которых строится список слов.
# Слова из списка сортируются по алфавиту и выводятся в метке.
def str_to_sort_list(event):
    s = ent.get()
    s = s.split()
    s.sort()
    lab['text'] = ' '.join(s)
# ==========================================================================

root = Tk()    # главное окно приложения

# создание виджетов ============================
ent = Entry(width=20)
but = Button(text="Преобразовать")
lab = Label(width=20, bg='black', fg='white')
# ==========================================================================
# Размещение виджетов. Пока виджеты будут располагаться друг под другом
# в главном окне приложения с помощью самого простого менеджера геометрии
# tkinter - метода pack: ===================================================
ent.pack()
but.pack()
lab.pack()
# ==========================================================================
# Вызов функции str_to_sort_list связывается с событием.
# В данном случае это делается с помощью метода bind, которому передаётся
# событие '<Button-1>' и функция-обработчик str_to_sort_list:
but.bind('<Button-1>', str_to_sort_list)
# ==========================================================================
```

```
root.mainloop()     # запуск цикла обработки событий
```

■ Перечень событий

Полный набор типов событий велик, и многие из них используются очень редко.

Далее представлены наиболее часто применяемые события:

Тип	Имя	Описание
36	ActivateState	Виджет меняется с неактивного на активный. Это относится к изменениям в опции виджета, такой как изменение кнопки с неактивной на активную.
4	Button	Нажата одна из кнопок мыши. Дополниительно указывается, какая.
5	ButtonRelease	Отпущена кнопка мыши. Это, вероятно в большинстве случаев, лучший выбор, чем событие нажимания, потому что, если пользователь случайно нажимает кнопку, он может переместить мышь с нажатой кнопкой с виджета, чтобы избежать запуска события Button.
22	Configure	Пользователь изменил размер виджета, например, перетащив угол или сторону окна.
37	Deactivate	Виджет меняет состояние с активного на неактивное. Это относится к изменениям в параметре виджета, такого как переключатель с активного на неактивное (серое) состояние.
17	Destroy	Виджет уничтожается.
7	Enter	Указатель мыши перемещён в видимую часть виджета. (Это отличается от клавиши ввода, которая является событием для ключа, именем которого на самом деле является KeyPress 'return'.)
12	Expose	Это событие происходит всякий раз, когда хотя бы какая-то часть приложения или виджета становится видимой после того, как она была скрыта другим окном.
9	FocusIn	Виджет получил фокус ввода. Это может произойти либо в ответ на действие пользователя (например, использование клавиши табуляции для перемещения фокуса между виджетами), либо программно (например, программа вызывает от имени виджета метод .focus_set()).
10	FocusOut	Фокус ввода был перемещен из виджета. Как и в случае с FocusIn, это событие могут вызвать пользователь или программа.
2	KeyPress	Пользователь нажал клавишу на клавиатуре. В событии в атрибуте detail указывается, что за ключ. Это ключевое слово может быть сокращено Key.
3	KeyRelease	Пользователь отпускает ключ.
8	Leave	Пользователь переместил указатель мыши из виджета.

19	Map	Виджет отображается, то есть становится видимым в приложении. Это происходит, например, при вызове метода виджета .grid().
6	Motion	Пользователь перемещает указатель мыши внутри виджета.
38	MouseWheel	Пользователь прокрутил колесо мыши вверх или вниз. В настоящее время это событие актуально для Windows и MacOS, но не для Linux.
18	Unmap	Виджет не отображается и больше отображаться не будет. Это происходит, например, при вызове метода виджета.grid_remove().
15	Visibility	Происходит, когда хотя бы какая-то часть окна приложения становится видимой на экране.

События, которые происходят при одновременном нажатии клавиш (сочетания клавиш), кодируются через тире. В случае применения, так называемого модификатора, в составном имени события он указывается первым, детали в описании - на третьем месте. Например,

- <Shift-Up> - одновременное нажатие клавиш Shift и стрелки вверх,
- <Control-B1-Motion> – движение мышью с зажатой левой кнопкой и клавишей Ctrl.

■ Создание GUI-программы

Последовательность действий при создании GUI-программы:

1. ! Создать главное окно!
2. Создать виджеты и выполнить конфигурацию их свойств (опций).
3. Определить события - то, на что будет реагировать программа.
4. Описать обработчики событий - то, как будет реагировать программа.
5. Расположить виджеты в главном окне.
6. ! Запустить цикл обработки событий!

Последовательность действий 2 - 5 не обязательно именно такая, НО пункты 1 и 6 ВСЕГДА остаются на своих местах.

a. Возможные сценарии создания пользовательского приложения

В современных операционных системах пользовательское приложение заключается в окно, которое можно назвать главным, так как в нем располагаются все остальные виджеты. Это объект окна верхнего уровня. Этот объект создается от класса Tk модуля tkinter.

Переменная, которая связывается с объектом, часто называют root (корень):

```
root = Tk()
```

Пример: в окне приложения располагаются текстовое поле, метка и кнопка. Эти объекты создаются от классов Entry, Label и Button модуля tkinter. С помощью передачи аргументов конструкторам этих классов можно сразу сконфигурировать некоторые их свойства :

```
ent = Entry(root, width=20)                    # текстовое поле
but = Button(root, text="Преобразовать")       # кнопка
```

```
lab = Label(root, width=20, bg='black', fg='white') # метка
```

При этом устанавливать свойства объектов не обязательно при их создании.

Существуют ещё способы, с помощью которых это можно сделать после создания виджета.

Первым аргументом в конструктор определяемого виджета передается виджет-хозяин - виджет, на котором будет располагаться создаваемый виджет.

!!!!!

В случае, когда элементы GUI помещаются непосредственно на главное окно, родителя можно не указывать.

!!!!!

То есть в этом примере можно убрать root:

```
ent = Entry(width=20)
but = Button(text="Преобразовать")
lab = Label(width=20, bg='black', fg='white')
```

Однако виджеты не обязательно располагаются на root'е. Они могут находиться на других виджетах.

И тогда указывать хозяина ("мастера") необходимо.

b. Объектно-ориентированный принцип

Применение классов в коде не обязательно, но ЖЕЛАТЕЛЬНО.

Пусть группа из поля, кнопки и метки представляет собой один объект, который порождается от класса Block. Тогда в основной ветке программы будет создано главное окно, объект типа Block и произведён запуск цикла обработки событий.

Объект типа Block должен быть привязан к главному окну. Для этого в конструктор класса передаётся ссылка на главное окно приложения.

В этом случае в приложении виджеты являются значениями полей объекта типа Block, функция-обработчик события нажатия на кнопку устанавливается с помощью свойства кнопки command, а не с помощью метода bind. В этом случае в вызываемой функции не требуется параметр event.

В метод передаётся только сам объект.

```
from tkinter import *
class Block:
    # конструктор ============================================================
    def __init__(self, masterRoot):
        self.ent = Entry(masterRoot, width=25)
        self.but = Button(masterRoot, text='Преобразовать')
        self.lab = Label(masterRoot, width=25, bg='black', fg='white')

        self.ent.pack()
        self.but.pack()
        self.lab.pack()

        # Метод-обработчик события нажатия на кнопку устанавливается ========
        # с помощью свойства кнопки command
```

```
        self.but['command'] = self.str_to_sort
        # ================================================================
    # ====================================================================

    # Метод str_to_sort_list. ========================================
    # При нажатии на кнопку обеспечивает разбиение текста,
    # вводимого в текстовое поле, на слова, из которых строится список слов.
    # Слова из списка сортируются по алфавиту и выводятся в метке.
    def str_to_sort(self):
        s = self.ent.get()
        s = s.split()
        s.sort()
        self.lab['text'] = ' '.join(s)
    # ====================================================================
root = Tk()
first_block = Block(root)

root.mainloop()
```

6.b.i. Замечание

Некоторое упрощение в разработке приложения, возможно, просматривается уже на данном этапе, когда существует один объект-представитель класса Block, который и сам имеет очень простую структуру (ТРИ виджета и ОДИН метод). Если же потребуется объект, состоящий из нескольких похожих объектов-блоков, то применение ООП оказывается ещё более оправданным.

Если требуется несколько блоков, состоящих из метки, кнопки, поля. И при этом у кнопки каждой группы будет свой метод-обработчик события клика, то в конструктор можно передавать значения для свойства command. Значение будет представлять собой привязанную к кнопке функцию-обработчик (метод обработки).

В результате выполнения этого кода будут созданы и выведены в окне root ДВА однотипных объекта-блока, кнопки которых будут вызывать разные действия, задаваемые аргументом fun при создании приложения.

6.b.ii. Конструктор с дополнительным аргументом funName и строкой для надписи на кнопке

Метод-обработчик события нажатия на кнопку устанавливается с помощью свойства кнопки command и выражения getattr(self, funName), в котором вместо funName подставляется строка с именем метода 'str_to_sort' или 'str_reverse', которые задают алгоритм преобразования. Функция getattr в объекте self по имени функции находит объект-ссылку на метод, который применяется в качестве метода-обработчика события нажатия на кнопку.

6.b.iii. Getattr как она есть...

Функция getattr() позволяет получить значение атрибута объекта по его имени.

Её синтаксис:

getattr(object, name, default)

Её параметры:

object - объект, значение атрибута которого требуется получить

name - имя атрибута: объект, должен быть строкой

default - значение по умолчанию, которое будет возвращено, если имя атрибута name отсутствует.

Возвращаемое значение:

значение именованного атрибута объекта

Описание:

Функция getattr() возвращает значение атрибута указанного объекта object по его имени name. Имя атрибута name должно быть строкой. Если строка является именем name одного из атрибутов объекта object, результатом является значение этого атрибута. Если атрибут с именем name не существует, возвращается значение по умолчанию default, если оно передано в функцию, в противном случае бросается исключение AttributeError.

Вызов getattr(object, 'x') полностью эквивалентно вызову object.x.

Есть только два широко известных случая, когда функция getattr() может быть полезна:

- когда невозможно вызвать object.x, т. к. заранее не известно, какой атрибут будет нужен. Например атрибут представляет собой строковую переменную. Очень часто встречается в метапрограммировании.
- когда точно неизвестно есть ли значение у нужного атрибута объекта. Например вызов object.z вызовет исключение AttributeError, если у атрибута z нет значения. Но если можно указать какое то значение по умолчанию для z, то вызов getattr(object, 'z', 5) вернет 5.

Метод str_to_sort_list:

При нажатии на кнопку обеспечивает разбиение текста, вводимого в текстовое поле, на слова, из которых строится список слов. Слова из списка сортируются по алфавиту и выводятся в метке.

Метод str_reverse:

При нажатии на кнопку обеспечивает переворачивание текста, вводимого в текстовое поле. Перевёрнутый список выводится в метке.

Третий аргумент выражения вызова конструктора Block является строкой, которая задаёт имя соответствующего метода обработки события 'нажимания на кнопку' и определяет алгоритм преобразования вводимой пользователем строки.

```
block_0 = Block(root, 'sort', 'str_to_sort')
block_1 = Block(root, 'reverse', 'str_reverse')

# В результате выполнения этого кода будут созданы и выведены в окне root ДВА
# однотипных объекта-блока, кнопки которых будут вызывать разные действия,
# задаваемые аргументом fun при создании приложения.

from tkinter import *
class Block:

    # конструктор с дополнительным аргументом funName и строкой =============
    # для надписи на кнопке ==================================================
    def __init__(self, masterRoot, buttontxt, funName):
        self.ent = Entry(masterRoot, width=25)
        self.but = Button(masterRoot, text=buttontxt)
        self.lab = Label(masterRoot, width=25, bg='black', fg='white')

        self.ent.pack()
        self.but.pack()
        self.lab.pack()

    # Метод-обработчик события нажатия на кнопку устанавливается ==========
```

```
# с помощью свойства кнопки command и выражения getattr(self, funName),
# в котором вместо funName подставляется строка с именем метода
# 'str_to_sort' или 'str_reverse', которые задают алгоритм преобразования
# getattr: в объекте self по имени функции находит объект-ссылку
# на метод, который применяется в качестве метода-обработчика
# события нажатия на кнопку.

    self.but['command'] = getattr(self, funName)

# ====================================================================
# Функция getattr() позволяет получить значение атрибута объекта
# по его имени.
# Синтаксис:
# getattr(object, name, default)
# Параметры:
# object - объект, значение атрибута которого требуется получить
# name - имя атрибута: объект, должен быть строкой
# default - значение по умолчанию, которое будет возвращено,
# если имя атрибута name отсутствует.
# Возвращаемое значение:
# значение именованного атрибута объекта
# Описание:
# Функция getattr() возвращает значение атрибута указанного
# объекта object по его имени name.
# Имя атрибута name должно быть строкой. Если строка является именем name
# одного из атрибутов объекта object, результатом является значение
# этого атрибута.
# Если атрибут с именем name не существует, возвращается значение
# по умолчанию default,
# если оно передано в функцию, в противном случае бросается
# исключение AttributeError.
# вызов getattr(object, 'x') полностью эквивалентно вызову object.x.
# Есть только два широко известных случая, когда функция getattr()
# может быть полезна:
# - когда невозможно вызвать object.x, т. к. заранее не известно,
# какой атрибут будет нужен.
# Например атрибут представляет собой строковую переменную.
# Очень часто встречается в метапрограммировании.
# - когда точно неизвестно есть ли значение у нужного атрибута объекта.
#   Например вызов object.z вызовет исключение AttributeError,
# если у атрибута z нет значения.
#   Но если можно указать какое то значение по умолчанию для z,
#  то вызов getattr(object, 'z', 5) вернет 5.
# ====================================================================
# Метод str_to_sort_list. ============================================
# При нажатии на кнопку обеспечивает разбиение текста,
# вводимого в текстовое поле, на слова, из которых строится список слов.
# Слова из списка сортируются по алфавиту и выводятся в метке.
def str_to_sort(self):
    s = self.ent.get()
    s = s.split()
    s.sort()
    self.lab['text'] = ' '.join(s)
# ====================================================================
# Метод str_reverse. =================================================
# При нажатии на кнопку обеспечивает переворачивание текста,
# вводимого в текстовое поле. Перевёрнутый список выводится в метке. ====
def str_reverse(self):
```

```
        s = self.ent.get()
        s = s.split()
        s.reverse()
        self.lab['text'] = ' '.join(s)
    # ========================================================================

root = Tk()

# третий аргумент выражения вызова конструктора является строкой,
# которая задаёт имя соответствующего метода обработки события
# 'нажимания на кнопку' и определяет алгоритм преобразования
# вводимой пользователем строки

block_0 = Block(root, 'sort', 'str_to_sort')
block_1 = Block(root, 'reverse', 'str_reverse')

root.mainloop()     # запуск цикла обработки событий
```

c. Виджеты Button, Label, Entry

Далее представлены три простых виджета GUI – Виджеты Button, Label, Entry (кнопка, метка, однострочное текстовое поле).

В tkinter объекты этих элементов интерфейса порождаются соответственно от классов Button, Label и Entry.

Свойства и методы виджетов бывают как относительно общими, характерными для многих типов, так и частными, встречающимися только у какого-то одного класса. В любом случае список настраиваемых свойств ожидаемо велик. Здесь будут представлены только основные свойства и методы классов пакета tkinter.

В Tkinter существует три способа конфигурирования свойств виджетов:

- в момент создания объекта,
- с помощью метода config, он же configure,
- путем обращения к свойству как к элементу словаря.

▪ Button – кнопка

Самыми важными свойствами виджета класса Button являются:

- text, с помощью которого устанавливается надпись на кнопке,
- command для установки действия, то есть того, что будет происходить при нажатии на кнопку.

Размер кнопки по умолчанию соответствует ширине и высоте текста, однако с помощью свойств width и height эти параметры можно изменить. Единицами измерения в данном случае являются знакоместа.

Такие свойства как

bg, fg, activebackground и activeforeground

определяют соответственно цвет фона и текста, цвет фона и текста во время нажатия и установки курсора мыши над кнопкой.

```
# ================================================================
from tkinter import *
```

```
def change():
    b1['text'] = "Изменено"
    b1['bg'] = '#000000'
    b1['activebackground'] = '#555555'
    b1['fg'] = '#ffffff'
    b1['activeforeground'] = '#ffffff'


root = Tk()

b1 = Button(text="Изменить", width=15, height=3)
b1.config(command=change)
b1.pack()


root.mainloop()

# ========================================================================
```

Здесь свойство command устанавливается с помощью метода config.

```
b0.config(command=change)
```

Однако можно было сделать и так:

```
b0['command'] = change
```

Это потому, что у виджета - объекта Button есть опция 'command'

- **Label - метка**

Виджет Label для отображения текста в окне. Применяется в основном для информационных целей (пропаганды) (вывод сообщений, подпись других элементов интерфейса).

Свойства метки похожи на свойства кнопки. Но у меток нет опции command. Поэтому связать метки с событием можно ТОЛЬКО с помощью метода bind.

И написать lbl['command'] = ... НЕЛЬЗЯ(!)

На примере объекта типа Label демонстрация свойства font – шрифт.

```
# ========================================================================
from tkinter import *

root = Tk()


l1 = Label(text="Работа над ошибками", font="Arial 32")
l2 = Label(text="Распознавание вариантов", font=("Comic Sans MS",24, "bold"))

l1.config(bd=20, bg='#ffaaaa')
l2.config(bd=20, bg='#aaffff')

l1.pack()
l2.pack()

root.mainloop()     # запуск цикла обработки событий
# ========================================================================
```

Значение шрифта можно передать как строку или как кортеж.

Если имя шрифта состоит из двух и более слов - то второй вариант более предпочтителен.

После названия шрифта можно указать размер и стиль.

Также как font свойство bd есть не только у метки. С помощью этого свойства регулируется размер границ (единица измерения – пиксель). Пока связь метки с событием показать не получилось.

Ещё вариант...

Если к меткам и кнопкам в коде не обращаются, то ссылки на метки и кнопки при их создании НЕ присваиваются переменным. Они создаются от своего класса и сразу размещаются с помощью метода pack.

```
# ================================================================
from tkinter import *

# функция-обработчик события (вешается на кнопку)
# и обеспечивает изменение свойства 'text' для
# конкретной метки, представленной глобальной
# переменной lbl, настроенной на объект-представитель
# класса Label
def take():
    lbl['text'] = "Выдано"
# ================================================

root = Tk()

# виджеты располагаются друг под другом в главном окне приложения
# с помощью менеджера геометрии - метода pack
Label(text="Пункт выдачи").pack()          # безымянный виджет Label.
Button(text="Взять", command=take).pack()  # безымянный виджет Button.
# Функция-обработчик события нажатия на кнопку устанавливается с помощью
# опции (свойства) кнопки command
lbl = Label(width=10, height=1)  # виджет Label.
                                 # Представлен ссылкой на объект lbl.
                                 # Ссылка актуальна, так как непосредственно
                                 # используется в функции-обработчике take
lbl.pack()

root.mainloop()
# ================================================================================
# В данном примере только у одной метки есть связь с переменной,
# так как одно из ее свойств может быть изменено в процессе
# выполнения программы.
# ================================================================================
```

▪ Entry – однострочное текстовое поле

Текстовые поля предназначены для ввода информации пользователем. Также и для вывода, если при этом предполагается, что текст из них будет скопирован. Текстовые поля как элементы графического интерфейса бывают однострочными и многострочными. В tkinter многострочным соответствует класс Text, который будет рассмотрен позже.

Свойства объектов Entry во многом схожи с двумя предыдущими виджетами.

А вот методы – НЕТ.

- Из текстового поля можно взять текст. За это действие отвечает метод get.
- В текстовое поле можно вставить текст методом insert.
- Также можно удалить текст методом delete.

Метод insert принимает позицию, в которую надо вставлять текст и сам текст.

```
from tkinter import *
from datetime import datetime as dt     # Модуль datetime предоставляет классы
                                        # для обработки времени и даты.

# Функция-обработчик события нажатия на кнопку ==============================
def insert_time():
    t = dt.now().time()
    e1.insert(0, t.strftime('%H:%M:%S  '))
# ============================================================================

root = Tk()
e1 = Entry(width=50)                                # однострочное текстовое поле
but = Button(text="Время", command=insert_time) # insert_time - функция-
                                                    # обработчик события
                                                    # нажатия на кнопку

# виджеты располагаются друг под другом ====================================
# в главном окне приложения с помощью менеджера геометрии - метода pack: ====
e1.pack()
but.pack()

root.mainloop()
# ============================================================================
# Этот код приведет к тому, что после каждого нажатия на кнопку будет
# вставляться новое время перед уже существующей в поле строкой.
#=============================================================================
```

Методы позиционирования элементов

Прежде чем продолжить разбираться с виджетами GUI, желательно прояснить вопрос их размещения в окне.

При работе с Tkinter для позиционирования элементов применяются разные методы:

- pack();
- place();
- grid().

Метод grid

позволяет поместить элемент в конкретную ячейку условной сетки либо грида. При этом используются параметры:

- column — это номер столбца, отсчитывается с нуля;
- row — это номер строки, отсчитывается с нуля;
- columnspan — указывает число столбцов, занимаемых элементом;
- rowspan — указывает число строк;
- ipadx и ipady — подразумеваются отступы по горизонтали и вертикали от границ компонента до текста компонента;
- padx и pady — аналогичные отступы, но от границ ячейки грида до границ компонента;

- sticky — определяет выравнивание элемента в ячейке в случае, когда ячейка больше компонента.

```
# приложение с двумя окнами (root0, root1), к root0 по умолчанию
# цепляется грид из 9 кнопок:

from tkinter import *

root0 = Tk()
root0.title('root0 500x450')
root0.geometry('500x450')

root1 = Tk()
root1.title('root1 300x250')
root1.geometry('300x250')

# Далее — грид:
for x in range(3):
 for y in range(3):
 btn = Button(text="{0}-{1}".format(x, y))
 btn.grid(row = x, column = y,
          ipadx = 10, ipady = 5,
          padx = 10, pady = 10)

# Метод mainloop используется для вызова окна виджета.
# И достаточно вызвать хотя бы одно!
root0.mainloop()

# root1.mainloop()
```

Метод mainloop используется для вызова окна виджета.

Метод pack

Это влияет на удобство применения приложения. Размещение виджетов в окне - это вопросы дизайна. Здесь программист и швец, и жнец, и на дудке игрец, и ещё интерфейсы разрабатывает.

В Tkinter, который далее будет рассмотрен в отдельной главе, существует три менеджера геометрии (так их принято называть) – упаковщик, сетка и размещение по координатам.

Сейчас - об упаковщике. Он самый простой и часто используемый. Остальные два - позже.

Упаковщик (packer) вызывается методом pack, который имеется у всех виджетов-объектов. Он уже много раз применялся.

Если к элементу интерфейса не применить какой-либо из менеджеров геометрии, то он вообще не отобразится в окне.

При этом в одном окне (или любом другом родительском виджете) нельзя комбинировать разные менеджеры.

Если для размещения виджетов был применён метод pack, то тут же применять методы grid (сетка) и place (место) НЕ получится!

Если в упаковщики не передавать аргументы, то виджеты будут располагаться вертикально, друг над другом. Тот объект, который первым вызовет pack, будет вверху. Который вторым – под первым, и так далее.

У метода pack есть параметр (атрибут) side (сторона), который принимает одно из четырех значений-констант, заданных в tkinter –

TOP, BOTTOM, LEFT, RIGHT

(верх, низ, лево, право).

По умолчанию, когда в pack не указывается side, его значение устанавливается в TOP.

Из-за этого виджеты располагаются вертикально.

Для демонстрации возможностей pack - определение РАСКРАШЕНЫХ меток:

```
.........
l1 = Label(width=7, height=4, bg='yellow', text='1')
l2 = Label(width=7, height=4, bg='orange', text='2')
l3 = Label(width=7, height=4, bg='lightgreen', text='3')
l4 = Label(width=7, height=4, bg='lightblue', text='4')
.........
```

и разные комбинации значений сайда:

```
.........
l1.pack()
l2.pack()
l3.pack()
l4.pack()
.........

.........
l1.pack(side=BOTTOM)
l2.pack(side=BOTTOM)
l3.pack(side=BOTTOM)
l4.pack(side=BOTTOM)
.........

.........
l1.pack(side=LEFT)
l2.pack(side=LEFT)
l3.pack(side=LEFT)
l4.pack(side=LEFT)
.........

.........
l1.pack(side=RIGHT)
l2.pack(side=RIGHT)
l3.pack(side=RIGHT)
l4.pack(side=RIGHT)
.........

.........
l1.pack(side=TOP)
l2.pack(side=BOTTOM)
l3.pack(side=RIGHT)
l4.pack(side=LEFT)
.........

.........
l1.pack(side=LEFT)
```

```
l2.pack(side=LEFT)
l3.pack(side=BOTTOM)
l4.pack(side=LEFT)
..........
```

С двумя последними вариантами размещения - проблема:

разместить виджеты "квадратом", т. е. два сверху, два снизу ровно под двумя верхними, - НЕ ПОЛУЧИТСЯ. В этом случае используется вспомогательный фрейм (рамка), который порождается от класса Frame. Фреймы размещаются на главном окне, а уже во фреймах – виджеты:

```
# ================================================================================
from tkinter import *
root = Tk()
# ================================================================================
frm_top = Frame(root)
frm_bot = Frame(root)
# frm_top, frm_bot - определяют на что ложить метку ======================
# метки: =========================================================================
# две метки в рамку (фрейм) frm_top
lbl1 = Label(frm_top, width=7, height=4, bg='yellow', text="1")
lbl2 = Label(frm_top, width=7, height=4, bg='orange', text="2")
# две метки в рамку (фрейм) frm_bot
lbl3 = Label(frm_bot, width=7, height=4, bg='lightgreen', text="3")
lbl4 = Label(frm_bot, width=7, height=4, bg='lightblue', text="4")
# рамки размещаются с помощью тандартного менеджера упаковки =============
frm_top.pack()
frm_bot.pack()
# метки пакуются в рамки ================================================
lbl1.pack(side=LEFT)
lbl2.pack(side=LEFT)
lbl3.pack(side=LEFT)
lbl4.pack(side=LEFT)
# вызов окна виджета ====================================================
root.mainloop()
# ================================================================================
```

- ***Свойства менеджера упаковки pack***

У менеджера упаковки pack кроме side есть ещё другие параметры-свойства.

Можно задать внутренние (ipadx и ipady) и внешние (padx и pady) отступы:

```
..........

frm_top.pack(padx=10, pady=10)
lbl1.pack(side=LEFT)
lbl2.pack(side=LEFT)

..........

frm_top.pack(ipadx=10, ipady=10)
lbl1.pack(side=LEFT)
lbl2.pack(side=LEFT)

..........
```

Когда устанавливаются внутренние отступы, то из-за того, что side прибивает виджет к левой границе, справа получается отступ в 20 пикселей, а слева – ничего. Можно частично решить проблему, если заменить внутренние отступы рамки на внешние отступы у меток.

```
.........

frm_top.pack()
lbl1.pack(side=LEFT, padx=10, pady=10)
lbl2.pack(side=LEFT, padx=10, pady=10)

.........
```

Но тут появляется промежуток между самими метками. Чтобы его убрать, пришлось бы каждый виджет укладывать в свой собственный фрейм. И вместо упрощения работ по реализации дизайна - лишняя суета и куча проблем. Таким образом, упаковщик Tkinter оказывается полезен только для разработки относительно простых интерфейсов.

- ***Свойства fill (заполнение) и expand (расширение)***

Ещё пара свойств – fill (заполнение) и expand (расширение).

expand по умолчанию установлен в ноль (его другое значение – единица),

fill по умолчанию – None (его другие значения BOTH, X, Y).

Пример применения (окно с одной меткой):

```
from tkinter import *

root = Tk()
lbl1 = Label(text="This is a label", width=30, height=10, bg="lightgreen")
lbl1.pack()

root.mainloop()
```

Если начать расширять окно с этой меткой или сразу раскрыть его на весь экран, то метка окажется вверху по вертикали и в середине по горизонтали. Причина, по которой метка не в середине по вертикали состоит в том, что side ПО УМОЛЧАНИЮ равен TOP, и метку прибивает(!) к верху окна.

Если же установить свойство expand в 1, то при расширении окна метка будет всегда в середине:

```
from tkinter import *

root = Tk()
lbl1 = Label(text="This is a label", width=30, height=10, bg="lightgreen")
lbl1.pack(expand=1)

root.mainloop()
```

Свойство fill заставляет виджет заполнять все доступное пространство. В зависимости от значения свойства fill, это пространство можно заполнить во всех направлениях или только по одной из осей:

```
from tkinter import *

root = Tk()
lbl1 = Label(text="This is a label", width=30, height=10, bg="lightgreen")
lbl1.pack(expand=1, fill=Y)

root.mainloop()
```

Таким образом, два свойства менеджера упаковки – fill (заполнение) и expand (расширение).

По-умолчанию expand равен 0 (другое значение – 1),

fill – NONE (другие значения BOTH, X, Y): NONE (default), which will keep the widget's original size; X, fill horizontally; Y, fill vertically; BOTH, fill horizontally and vertically.

- ***anchor (якорь)***

И ещё одна опция метода pack – anchor (якорь) – может принимать значения:

- N (north – север),
- S (south – юг),
- W (west – запад),
- E (east – восток),
- CENTER (center - без комментариев)

и их комбинации, представленные отдельными константами.

То есть, anchor must be: N, NE, E, SE, S, SW, W, NW, or CENTER

- **Метод place**

Это третий (последний) менеджер геометрии библиотеки tk – Place, который размещает виджеты по координатам.

В tkinter использование данного менеджера геометрии реализуется через метод place, реализованный в классах виджетов.

Метод place указывает виджету его положение либо в абсолютных значениях (в пикселях),

либо относительно (в долях родительского окна).

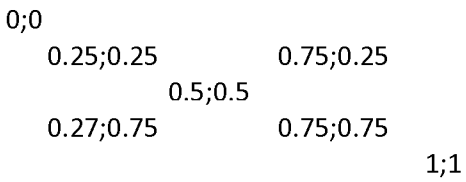
Также абсолютно и относительно можно задавать размер самого виджета.

- ***Метод place: основные параметры***

- anchor (якорь) – определяет часть виджета, для которой задаются координаты. Принимает значения N, NE, E, SE, SW, W, NW или CENTER. По умолчанию NW (верхний левый угол).
- relwidth, relheight (относительные ширина и высота) – определяют размер виджета в долях его родителя.

- relx, rely – определяют относительную позицию в родительском виджете.
  Координата (0; 0) – у левого верхнего угла, (1; 1) – у правого нижнего.
- width, height – абсолютный размер виджета в пикселях. Значения по умолчанию (когда данные опции опущены) приравниваются к естественному размеру виджета, то есть к тому, который определяется при его создании и конфигурировании.

- x, y – абсолютная позиция в пикселях. Значения по умолчанию приравниваются к нулю.
- Схема с указанием относительных координат:



разница между абсолютным и относительным позиционированием на примере:

```
# ================================================================================
# Кнопка, позиция которой была жестко задана абсолютным позиционированием,
# не изменяет своего положения при изменении размеров окна.
# Кнопка, позиция которой была задана с относительными координатами,
# смещается. Опции relx и rely для нее по-прежнему 0.3 и 0.5, но уже
# относительно нового размера окна.

from tkinter import *

root = Tk()
root.geometry('150x150')

Button(bg='red').place(x=75, y=20)
Button(bg='green').place(relx=0.3, rely=0.5)

root.mainloop()

# ================================================================================
```

С размерами виджетов то же самое. Если они относительны, то с изменением размера родительского виджета их размеры также будут изменяться. Если нужно чтобы виджет не менял свего размера, этот размер должен задаваться как абсолютный. Или не указываться вовсе! В таком случае он будет приравнен к естественному (???).

```
…
Label(bg='white').place(x=10, y=10, width=50, height=30)
Label(bg='green').place(x=10, y=50, relwidth=0.3, relheight=0.15)
…
```

Комбинируя различные варианты позиционирования и установки размеров, можно добиться неожиданных результатов. Поэтому метод place следует применять очень осторожно, с пониманием возможных последствий. Если при этом размер окна приложения не меняется, а интерфейс сложен, то управляющий размещением Place может оказаться наилучшим вариантом, так как с его помощью можно выполнить тонкую настройку и создать аккуратный интерфейс.

Text – многострочное текстовое поле

В tkinter многострочное текстовое поле создается от класса Text. По умолчанию его размер равен 80-ти знакоместам по горизонтали и 24-м по вертикали.

```
# ================================================================
from tkinter import *
root = Tk()
txt = Text()
print(txt['width'], txt['height'])
txt.pack()
root.mainloop()

# ================================================================
```

Эти свойства можно менять с помощью опций width и height. Также можно конфигурировать шрифт, цвета и другое.

```
# ================================================================

from tkinter import *
root = Tk()
txt = Text(width=25, height=5, bg="darkgreen", fg='white', wrap=WORD)
txt.pack()
root.mainloop()
# ================================================================
```

Значение WORD опции wrap позволяет переносить слова на новую строку целиком, а не по буквам.

▪ Text и Scrollbar

Если в многострочное текстовое поле вводится больше строк текста, чем его высота, то это поле само будет прокручиваться вниз. При просмотре введённый текст можно прокручивать вверх-вниз можно с помощью колеса мыши и стрелками на клавиатуре. Также удобно пользоваться скроллером – полосой прокрутки. В tkinter скроллеры производятся от класса Scrollbar. Объект-скроллер связывают с виджетом, которому он требуется.

И это не обязательно многострочное текстовое поле. Часто полосы прокрутки бывают нужны спискам, которые будут рассмотрены позже.

Далее создаётся скроллер, к которому с помощью опции command привязывается прокрутка текстового поля по оси y – text.yview. В свою очередь текстовому полю опцией yscrollcommand устанавливается ранее созданный скроллер – scroll.set.

```
# ================================================================
from tkinter import *
root = Tk()

text = Text(width=20, height=7)
text.pack(side=LEFT)

scroll = Scrollbar(command=text.yview)
scroll.pack(side=LEFT, fill=Y)

text.config(yscrollcommand=scroll.set)

root.mainloop()
# ================================================================
```

▪ Методы Text

Основные методы у Text такие же, как у Entry – get, insert, delete. Однако, если в случае однострочного текстового поля было достаточно указать один индекс элемента при вставке или удалении, то в случае многострочного надо указывать два – номер строки и номер символа в этой строке (номер столбца).

!!!!! При этом нумерация строк начинается с единицы, а столбцов – с нуля. !!!!!

Методы get и delete могут принимать не два, а один аргумент. В таком случае будет обрабатываться только один символ в указанной позиции.

```
# ================================================================
from tkinter import *

# обработчики событий
# ================================================
def insert_text():
    s = "Hello World"
    text.insert(1.0, s)
# ================================================
# ================================================
def get_text():
    s = text.get(1.0, END)
    label['text'] = s
# ================================================
# ================================================
def delete_text():
    text.delete(1.0, END)
# ================================================

root = Tk()

text = Text(width=25, height=5)
text.pack()

frame = Frame()
frame.pack()

# кнопки вставляются в рамку ======================================
# у кнопок собственные обработчики
Button(frame, text="Вставить", command=insert_text).pack(side=LEFT)
Button(frame, text="Взять",    command=get_text).pack(side=LEFT)
Button(frame, text="Удалить",  command=delete_text).pack(side=LEFT)

# метка. Применяется в функции обработки get_text
label = Label()
label.pack()

root.mainloop()
# ================================================================
```

○ Класс LabelFrame

Также существует класс LabelFrame – фрейм с подписью: у него есть свойство text.

```
.........
frm_top = LabelFrame(root, text = 'Верх')
```

```
frm_bot = LabelFrame(root, text = 'Низ')
..........

# ================================================================================
from tkinter import *
root = Tk()

# frm_top = Frame(root)
# frm_bot = Frame(root)

# frm_top = LabelFrame(root, text = 'Верх')
# frm_bot = LabelFrame(root, text = 'Низ')

frm_top = LabelFrame(text = 'Верх')
frm_bot = LabelFrame(text = 'Низ')
# Виджет размещается в главном окне приложения,
# поэтому аргумент root может быть по умолчанию.

# первый аргумент - информация куда метки предназначены
# frm_top, frm_bot определяют на что ложить метку

lbl1 = Label(frm_top, width=7, height=4, bg='yellow', text="1")
lbl2 = Label(frm_top, width=7, height=4, bg='orange', text="2")
lbl3 = Label(frm_bot, width=7, height=4, bg='lightgreen', text="3")
lbl4 = Label(frm_bot, width=7, height=4, bg='lightblue', text="4")

# далее - варианты применения менеджера упаковки pack с различными виджетами
frm_top.pack()
frm_bot.pack()

lbl1.pack(side=LEFT)
lbl2.pack(side=LEFT)
lbl3.pack(side=LEFT)
lbl4.pack(side=LEFT)
# ================================================================================

root.mainloop()

..........
lbl1.pack(expand=1, anchor=SE)  # SE - это отдельная константа
..........
```

- **Теги**

Текстовое поле библиотеки Tk позволяет форматировать в нём текст - то есть придавать его разным частям разное оформление.

Это делается это с помощью методов tag_add и tag_config.

tag_add добавляет тег, при этом надо указать его произвольное имя и отрезок текста, к которому он будет применяться.

tag_config настраивает тегу стили оформления.

```
# ================================================================
from tkinter import *
root = Tk()
```

```
text = Text(width=50, height=10)
text.pack()
text.insert(1.0, "Hello world!\nline two")

text.tag_add('title', 1.0, '1.end')
text.tag_config('title', justify=CENTER, font=("Verdana", 24, 'bold'))

root.mainloop()
# ==========================================================================
```

▪ Вставка виджетов в текстовое поле

С помощью метода window_creat в Text можно вставлять другие виджеты.

Это не очень-то и нужно, однако с объектами типа Canvas (этот класс будет описан позже) может быть интересно. Ниже вставляется метка в текущую (INSERT) позицию курсора.

```
# ==========================================================================
from tkinter import *

# ==========================================================================
# Размещение метки в функции позволяет каждый раз при вызове функции
# создавать новую метку. Иначе, если бы метка была в
# основной ветке программы, предыдущая метка исчезала бы.
def smile():
    label = Label(text=":)", bg="yellow")
    text.window_create(INSERT, window=label)
# ==========================================================================

root = Tk()

text = Text(width=50, height=10)
text.pack()

# кнопка. Обработчик события – функция smile
button = Button(text=":)", command=smile)
button.pack()

root.mainloop()

# ==========================================================================
```

○ Radiobutton и Checkbutton: переменные Tkinter

Радиокнопки и Флажки: в Tkinter от класса Radiobutton создаются радиокнопки, от класса Checkbutton – флажки.

Радиокнопки НЕ создаются по одной. Делать один такой виджет не имеет смысла, так как радиокнопка это инструмент выбора. Для выбора делается связанная группа радиокнопок, которая работает по принципу переключателей: когда включена одна, другие выключены.

Объекты (экземпляры) Checkbutton также могут быть визуально оформлены в группу. Но при этом каждый флажок в группе независим от остальных. Каждый может быть в состоянии "установлен" или "снят", независимо от состояний других флажков.

Таким образом, в группе Checkbutton можно сделать множественный выбор, в группе Radiobutton – нет.

- **Radiobutton – радиокнопка**

Если будут объявлены несколько (две) радиокнопки БЕЗ соответствующих настроек, то обе они по умолчанию переводятся во включённое состояние и выключить их будет невозможно. Интерпретатор рассматривает эти радиокнопки как вырожденные независимые группы радиокнопок. Эти переключатели никак не связаны друг с другом. Также для них не указано исходное значение, то есть, должны ли они быть в состоянии "вкл" или "выкл".

```
# ================================================================================
from tkinter import *

root = Tk()

r1 = Radiobutton(text='First')
r2 = Radiobutton(text='Second')

r1.pack(anchor=W)
r2.pack(anchor=W)

root.mainloop()
# ================================================================================
```

Для хранения состояний виджетов в Tkinter подходит НЕ ЛЮБАЯ переменная.

Для этих целей предусмотрены специальные классы – переменные пакета tkinter: BooleanVar, IntVar, DoubleVar, StringVar. Первый класс позволяет принимать своим экземплярам только булевы значения (0 или 1 и True или False), второй – целые, третий – дробные, четвертый – строковые. При этом класс переменной для хранения состояний важен только на начальном этапе выполнения приложения, непосредственно при формировании группы радиокнопок. При этом выполняется главная задача – ОГРАНИЧЕНИЕ ТИПОВ ПЕРЕМЕННЫХ, КОТОРЫЕ ПРИМЕНЯЮТСЯ ПРИ НАСТРОЙКЕ И РАБОТЕ РАДИОКНОПОК В ГРУППЕ. Для этого НЕ используются вещественные и комплексные значения.

Связь между радиокнопками в группе устанавливается через общую переменную, разные значения которой соответствуют включению разных радиокнопок группы. У всех кнопок одной группы свойство variable устанавливается в одно и то же значение – связанную с группой переменную. Свойству value присваиваются разные значения этой переменной. Радиокнопки в группе характеризуются значением атрибута value, который не обязательно должен различаться у каждой кнопки и одним ОБЩИМ (не важно каким!) значением атрибута variable, по которому радиокнопки объединяются в одну группу, независимо от конкретного значения этого атрибута. Таким образом, на момент формирования группы, значение атрибута variable может иметь любое значение, которое меняется в зависимости от значения атрибута value радиокнопки, выбранной в данной группе формируемых виджетов. Если формируется группа радиокнопок (rbtn_0, ..., rbtn_N), с переменной, var_rbtn, которая объединяет данное множество в группу, далее область значений var_rbtn определяется множеством значений группы радиокнопок (rbtn_0, ..., rbtn_N).

В группе ЗАЖИГАЮТСЯ ВСЕ ТЕ РАДИОКНОПКИ, ЧЬИ ЗНАЧЕНИЯ АТРИБУТОВ 'value' СОВПАДАЮТ СО ЗНАЧЕНИЕМ переменной var_rbtn, независимо от выбора радиокнопки. В одно и то же состояние (ЗАЖЖЕНО - ПОГАШЕНО) одновременно могут переключиться несколько радиокнопок: переменная var_rbtn ОДНА на всю группу радиокнопок; при переключении принимает значение 'value' одного из виджетов группы. Это значение может совпадать со значениями атрибутов 'value' СРАЗУ НЕСКОЛЬКИХ радиокнопок в группе;

в этом случае одновременно ЗАЖИГАТЬСЯ и ГАСИТЬСЯ будут несколько радиокнопок в группе.

```
r_var = BooleanVar()
```

```
r_var.set(0)
r1 = Radiobutton(text='First', variable=r_var, value=0)
r2 = Radiobutton(text='Second', variable=r_var, value=1)
```

Здесь переменной r_var присваивается объект типа BooleanVar. С помощью метода set он устанавливается в значение 0.

При запуске приложения с группой радиокнопок включенной оказывается первая радиокнопка, так как значение ее опции value совпадает с текущим значением переменной r_var. Если кликнуть по второй радиокнопке, то она включится, а первая выключится. При этом значение r_var станет равным 1. С помощью метода get, определенного в объектах Tkinter можно определить в программе, какая из кнопок в группе находится в состоянии включено.

В следующем примере в функции change в зависимости от прочитанного значения переменной var ход выполнения программы идет по одной из трех веток.

```
# =====Вариант 1=============================================================
from tkinter import *

def change():
    if var.get() == 0:
        label['bg'] = 'red'
    elif var.get() == 1:
        label['bg'] = 'green'
    elif var.get() == 2:
        label['bg'] = 'blue'

root = Tk()

var = IntVar()
var.set(0)
red = Radiobutton(text="Red", variable=var, value=0)
green = Radiobutton(text="Green", variable=var, value=1)
blue = Radiobutton(text="Blue", variable=var, value=2)

button = Button(text="Изменить", command=change)

label = Label(width=20, height=10)

red.pack()
green.pack()
blue.pack()
button.pack()
label.pack()

root.mainloop()

# ============================================================================
```

От кнопки "Изменить" можно избавиться. Для этого надо связать функцию change или любую другую непосредственно со свойством command радиокнопок. При этом НЕ обязательно, чтобы радиокнопки, объединенные в одну группу, вызывали одну и ту же функцию. В следующем примере таких функций ТРИ.

В любом случае, метка (объект класса Label) будет менять цвет при клике по радиокнопкам.

```
# =====Вариант 2=========================================================
from tkinter import *

# обработчики событий переключения радиокнопок ======================

def red_lbl():
    label['bg'] = 'red'

def green_lbl():
    label['bg'] = 'green'

def blue_lbl():
    label['bg'] = 'blue'

# ====================================================================

root = Tk()

var = IntVar()

var.set(0)

Radiobutton(text="Red", variable=var, value=0, command=red_lbl).pack()
Radiobutton(text="Green", variable=var, value=1, command=green_lbl).pack()
Radiobutton(text="Blue", variable=var, value=2, command=blue_lbl).pack()

label = Label(width=20, height=10)
label.pack()

# функция обработки запускается САМА ПО СЕБЕ!
red_lbl()

root.mainloop()
# ============================================================================
```

- **Ещё примеры**

Если создать пару радиокнопок без соответствующих настроек, то обе они будут включены и выключить их будет невозможно. Эти переключатели никак не будут связаны друг с другом. Кроме того для них не указано исходное значение, должны ли они быть в состоянии "вкл" или "выкл". По-умолчанию они включены.

Связь между ними устанавливается через общую переменную, разные значения которой соответствуют включению разных радиокнопок группы. У всех кнопок одной группы свойство variable устанавливается в одно и то же значение, которое задаётся связанную с группой переменную. Свойству value присваиваются разные значения этой переменной.

В Tkinter нельзя использовать любую переменную для хранения состояний виджетов. Для этих целей предусмотрены специальные классы-переменные пакета tkinter – BooleanVar, IntVar, DoubleVar, StringVar. Первый класс позволяет принимать своим экземплярам только булевы значения (0 или 1 и True или False), второй – целые, третий – дробные, четвертый – строковые.

```
r_var = BooleanVar()
r_var.set(0)
r1 = Radiobutton(text='First',
                 variable=r_var, value=0)
r2 = Radiobutton(text='Second',
```

```
                variable=r_var, value=1)
```

Здесь переменной r_var присваивается объект типа BooleanVar. С помощью метода set он устанавливается в значение 0. При запуске программы включенной окажется первая радиокнопка, так как значение ее опции value совпадает с текущим значением переменной r_var. Если кликнуть по второй радиокнопке, то она включится, а первая выключится. При этом значение r_var станет равным 1.

В коде обычно требуется прочитать данные о том, какая из кнопок включена.

```
from tkinter import *

def change():
    if var.get() == 0:
        label['bg'] = 'red'
    elif var.get() == 1:
        label['bg'] = 'green'
    elif var.get() == 2:
        label['bg'] = 'blue'

root = Tk()

var = IntVar()
var.set(0)
red = Radiobutton(text="Red",
                  variable=var, value=0)
green = Radiobutton(text="Green",
                    variable=var, value=1)
blue = Radiobutton(text="Blue",
                   variable=var, value=2)
button = Button(text="Изменить",
                command=change)
label = Label(width=20, height=10)
red.pack()
green.pack()
blue.pack()
button.pack()
label.pack()

root.mainloop()
```

В функции change в зависимости от считанного значения переменной var выполнение программы идет по одной из трех веток.

Можно избавиться от кнопки "Изменить", связав функцию change или любую другую со свойством command радиокнопок. При этом не обязательно, чтобы радиокнопки, объединенные в одну группу, вызывали одну и ту же функцию.

```
from tkinter import *

def red_label():
    label['bg'] = 'red'

def green_label():
    label['bg'] = 'green'
```

```
def blue_label():
    label['bg'] = 'blue'

root = Tk()

var = IntVar()
var.set(0)
Radiobutton(text="Red", command=red_label,
            variable=var, value=0).pack()
Radiobutton(text="Green", command=green_label,
            variable=var, value=1).pack()
Radiobutton(text="Blue", command=blue_label,
            variable=var, value=2).pack()
label = Label(width=20, height=10, bg='red')
label.pack()

root.mainloop()
```

Метка будет менять цвет при клике по радиокнопкам.

В этом примере код радиокнопок почти идентичный, как и код связанных с ними функций. В таких случаях однотипный код можно поместить в класс.

```
from tkinter import *

def paint(color):
    label['bg'] = color

class RBColor:
    def __init__(self, color, val):
        Radiobutton(
            text=color.capitalize(),
            command=lambda i=color: paint(i),
            variable=var, value=val).pack()

root = Tk()

var = IntVar()
var.set(0)
RBColor('red', 0)
RBColor('green', 1)
RBColor('blue', 2)
label = Label(width=20, height=10, bg='red')
label.pack()

root.mainloop()
```

В двух последних вариантах кода метод get, чтобы получить значение переменной var не используется. В данном случае этого не требуется, потому что цвет метки меняется в момент клика по соответствующей радиокнопке и не находится в зависимости от значения переменной. Несмотря на это использовать переменную в настройках радиокнопок необходимо, так как она обеспечивает связывание их в единую группу и выключение одной при включении другой.

Ещё пример. Здесь все радиокнопки (13 штук) объявляются в одном классе, ЕДИНСТВЕННЫЙ объект которого создаётся в функции main. В этом же классе размещается функция обработки события, связанного с кнопкой, которая запускает процесс обработки выбранной радиокнопки: её имя печатается в поле текстовой метки. Для создания графического интерфейса пользователя в приложении используются методы библиотечного модуля tkinter.

```
"""
Требуется создать проект "Консольный файловый менеджер"
В проекте предполагается реализовать функционал, для которого и было
реализовано данное меню. После запуска программы пользователь видит меню,
состоящее из следующих пунктов:
- создать папку;
- удалить папку;
- удалить файл;
- копировать папку;
- копировать файл;
- просмотр содержимого рабочей директории;
- посмотреть только папки;
- посмотреть только файлы;
- просмотр информации об операционной системе;
- создатель программы;
- играть в викторину;
- мой банковский счет;
- смена рабочей директории (*необязательный пункт);
- выход.
После выполнения какого либо из пунктов снова возвращаемся в меню, пока не
будет выбран
пункт меню выход
Последовательность действий при создании программы пориеняющей методы
библиотеки tkinter:
1. ! Создать главное окно!
2. Создать виджеты и выполнить конфигурацию их свойств (опций).
3. Определить события - то, на что будет реагировать программа.
4. Описать обработчики событий - то, как будет реагировать программа.
5. Расположить виджеты в главном окне.
6. ! Запустить цикл обработки событий!
Последовательность действий 2 - 5 не обязательно именно такая,
НО пункты 1 и 6 ВСЕГДА остаются на своих местах.

"""
# ========================================================================
from tkinter import *

class CFManager:

    def __init__(self, funName ):
        self.masterRoot = Tk()     # приложение заключается в окно,
                                   # которое называтся главным,
                                   # так как в нем располагаются все остальные
# виджеты. Это объект окна верхнего уровня.
# Этот объект создается от класса Tk модуля tkinter.
# Переменная, которая связывается с объектом, часто называется root (корень).
        self.var = IntVar()
        self.var.set(0)
        self.rbCreateFolder = Radiobutton(text="create folder",
                                          variable=self.var, value=0)
        self.rbCreateFolder.pack() #"create folder"
        self.rbDeleteFolder = Radiobutton(text="delete folder",
                                          variable=self.var, value=1)
        self.rbDeleteFolder.pack() #"delete folder"
        self.rbDeleteFile = Radiobutton(text="delete file",
                                        variable=self.var, value=2)
```

```
        self.rbDeleteFile.pack() # "delete file"
        self.rbCCopyFolder = Radiobutton(text="copy folder",
                                         variable=self.var, value=3)
        self.rbCCopyFolder.pack() #"copy folder"
        self.rbCopyFile = Radiobutton(text="copy file",
                                      variable=self.var, value=4)
        self.rbCopyFile.pack() #"copy file"
        self.rbViewTheContentsOfTheWorkingDirectory =
        Radiobutton(text="view the contents of the working directory",
                    variable=self.var, value=5)
        self.rbViewTheContentsOfTheWorkingDirectory.pack()
                        #"view the contents of the working directory"

        self.rbSeeOnlyFolders = Radiobutton(text="see only folders",
                                            variable=self.var, value=6)
        self.rbSeeOnlyFolders.pack() #"see only folders"
        self.rbSeeOnlyFiles = Radiobutton(text="see only files",
                                          variable=self.var, value=7)
        self.rbSeeOnlyFiles.pack() #"see only files"

        self.rbSeeOperationSystemInformation = Radiobutton(
                                    text="see operation system information",
                                    variable=self.var, value=8)
        self.rbSeeOperationSystemInformation.pack()
                                    #"see operation system information"

        self.rbProgramCreator = Radiobutton(text="program creator",
                                            variable=self.var, value=9)
        self.rbProgramCreator.pack() #"program creator"
        self.rbQuiz = Radiobutton(text="quiz", variable=self.var, value=10)
        self.rbQuiz.pack() # "quiz"
        self.rbBankAccount = Radiobutton(text="bank account",
                                         variable=self.var, value=11)
        self.rbBankAccount.pack() # "bank account"

        self.rbChangeWorkingDirectory = Radiobutton(
                                            text="change working directory",
                                            variable=self.var, value=12)
        self.rbChangeWorkingDirectory.pack() #"change working directory"

        self.rbExit = Radiobutton(text="exit", variable=self.var, value=13)
        self.rbExit.pack() #"exit"

        self.butDoIt = Button(text="do it", command=self.rbrScaner)
# Вызов метода rbrScaner для
# определения пункта меню, который соответствует выбранной радиокнопке
        self.butDoIt.pack() #"do it"

        self.lblMain = Label(self.masterRoot, width=50,
                             bg='white', fg='black')
        self.lblMain.pack()

        self.masterRoot.mainloop()
                              # здесь же запускается цикл обработки событий

# ========================================================================
    def rbrScaner(self):
```

```
        if self.var.get() == 0:
            self.lblMain['text'] = "create folder"
        elif self.var.get() == 1:
            self.lblMain['text'] = "delete folder"
        elif self.var.get() == 2:
            self.lblMain['text'] = "delete file"
        elif self.var.get() == 3:
            self.lblMain['text'] = "copy folder"
        elif self.var.get() == 4:
            self.lblMain['text'] = "copy file"
        elif self.var.get() == 5:
            self.lblMain['text'] =
                              "view the contents of the working directory"
        elif self.var.get() == 6:
            self.lblMain['text'] = "see only folders"
        elif self.var.get() == 7:
            self.lblMain['text'] = "see only files"
        elif self.var.get() == 8:
            self.lblMain['text'] = "see operation system information"
        elif self.var.get() == 9:
            self.lblMain['text'] = "program creator"
        elif self.var.get() == 10:
            self.lblMain['text'] = "quiz"
        elif self.var.get() == 11:
            self.lblMain['text'] = "bank account"
        elif self.var.get() == 12:
            self.lblMain['text'] = "change working directory"
        elif self.var.get() == 13:
            self.lblMain['text'] = "exit"

def main():
    cfm = CFManager("rbrScaner")

if __name__ == '__main__':
    main()
# ================================================================================
```

- **Checkbutton – флажок**

Флажки не требуют установки между собой связи, поэтому возникает вопрос, по поводу переменных Tkinter. Они нужны для получения сведений о состоянии флажков. По значению связанной с Checkbutton переменной можно определить, установлен флажок или снят, что в свою очередь повлияет на ход выполнения программы.

У каждого флажка должна быть своя переменная Tkinter. И при включении одного флажка, другой будет выключаться, так как значение общей tkinter-переменной изменится и не будет равно значению опции onvalue первого флажка.

```
from tkinter import *
root = Tk()

def show():
    s = f'{var1.get()},
        f'{var2.get()}'
    lab['text'] = s

frame = Frame()
```

```
frame.pack(side=LEFT)

var1 = BooleanVar()
var1.set(0)
c1 = Checkbutton(frame, text="First",
                 variable=var1,
                 onvalue=1, offvalue=0,
                 command=show)
c1.pack(anchor=W, padx=10)

var2 = IntVar()
var2.set(-1)
c2 = Checkbutton(frame, text="Second",
                 variable=var2,
                 onvalue=1, offvalue=0,
                 command=show)
c2.pack(anchor=W, padx=10)

lab = Label(width=25, height=5, bg="lightblue")
lab.pack(side=RIGHT)

root.mainloop()
```

С помощью опции onvalue устанавливается значение, которое принимает связанная переменная при включенном флажке. С помощью свойства offvalue – при выключенном. В данном случае оба флажка при запуске программы будут выключены, так как методом set были установлены отличные от onvalue значения.

Опцию offvalue можно не указывать. Однако при ее наличии можно отследить, выключался ли флажок.

С помощью методов select и deselect флажков можно их программно включать и выключать. То же самое относится к радиокнопкам.

```
from tkinter import *

class CheckButton:
    def __init__(self, master, title):
        self.var = BooleanVar()
        self.var.set(0)
        self.title = title
        self.cb = Checkbutton(
            master, text=title, variable=self.var,
            onvalue=1, offvalue=0)
        self.cb.pack(side=LEFT)

def ch_on():
    for ch in checks:
        ch.cb.select()

def ch_off():
    for ch in checks:
        ch.cb.deselect()

root = Tk()

f1 = Frame()
f1.pack(padx=10, pady=10)
checks = []
for i in range(10):
    checks.append(CheckButton(f1, i))
```

```
f2 = Frame()
f2.pack()
button_on = Button(f2, text="Все ВКЛ",
                   command=ch_on)
button_on.pack(side=LEFT)
button_off = Button(f2, text="Все ВЫКЛ",
                    command=ch_off)
button_off.pack(side=LEFT)

root.mainloop()


# ================================================================
from tkinter import *

# функция обработки событий переключения флажка ===========
def show(lblkey):
    s = f'{var1.get()}, ' \
        f'{var2.get()}'
    lblkey['text'] = s
# ==========================================================

root = Tk()

# рамка
frame = Frame()
frame.pack(side=LEFT)

# метка
lbl = Label(width=25, height=5, bg="lightblue")

lbl.pack(side=RIGHT)

# флажок c1 ================================================
# Checkbutton описывается двумя переменными: var1 и c1

var1 = BooleanVar()
var1.set(0)

c1 = Checkbutton(frame,
                 text="First",
                 variable=var1,
                 onvalue=1,
                 offvalue=0,
                 # обработчик события кодируется лямбда-выражением,
                 # которое позволяет создать анонимную функцию на основе
                 # функции обработки событий переключения флажка -
                 # функции show.
                 # В ОДНО ВЫРАЖЕНИЕ УДАЁТСЯ ПОЛУЧИТЬ ЗНАЧЕНИЕ АРГУМЕНТА (lbl)
                 # И ВЫЗВАТЬ ФУНКЦИЮ
                 command=lambda l=lbl: show(l)
                 )
# размещение флажка c1 методом упаковки pack
c1.pack(anchor=W, padx=10)

# ==========================================================

# флажок c2 ================================================
# Checkbutton описывается двумя переменными: var2 и c2
```

```
var2 = IntVar()  # целочисленные значения со знаком.
var2.set(-1)     # при начальной загрузке var2 устанавливается в -1

c2 = Checkbutton(frame,
                 text="Second",
                 variable=var2,
                 onvalue=1,
                 offvalue=0,
                 # лямбда обработчик события:
                 # получает значение аргумента (lbl)
                 # и обеспечивает вызов функции
                 command=lambda l=lbl: show(l)
                 )
# размещение флажка c2 методом упаковки pack
c2.pack(anchor=W, padx=10)

# ============================================================

root.mainloop()

# ================================================================
```

○ Класс RBColor

В следующем примере для получения значения переменной var метод get не используется.

Этого не требуется, потому что цвет метки меняется в момент клика по соответствующей радиокнопке и зависит от значения переменной. Несмотря на это в настройках радиокнопок использовать переменную необходимо, так как она обеспечивает связывание их в единую группу и выключение одной при включении другой.

Здесь в качестве обработчика события применяется lambda – выражение (анонимная функция): особая синтаксическая конструкция, которая позволяет преобразовать описание функции любой сложности (с произвольным количеством параметров) в простое (да неужели…) и компактное представление, состоящее из одного выражения со стандартной структурой, которая позволяет присвоить определённые значения аргументам ранее объявленной функции обработки события и представить соответствующее выражение вызова.

В определении лямбда – выражения (анонимной функции) нет оператора return, так как в этой конструкции может быть только одно выражение, которое всегда возвращает значение и завершает работу анонимной функции.

В анонимной функции

- может содержаться только одно выражение
- могут передаваться сколько угодно аргументов

```
# ================================================================================
"""
В данном случае метод get для получения значения переменной var
здесь не используется.
Этого не требуется, потому что цвет метки меняется в момент клика по
соответствующей радиокнопке и зависит от значения переменной.
Несмотря на это в настройках радиокнопок использовать переменную
```

```
необходимо, так как она обеспечивает связывание их в единую группу
и выключение одной при включении другой.
"""
# =====Вариант 3=============================================================
from tkinter import *

class RBColor:

    # var - статическая переменная класса RBColor
    var = None

  # RBColor.var обеспечивает связывание объектов RBColor в единую группу
  # и выключение одной кнопки при включении другой.
    def __init__(self, color, val, lblkey):
        Radiobutton(text=color.capitalize(),                 # надпись
                    value=val,                               # номер кнопки
                    variable=RBColor.var,       # управляющая переменная
                    command=lambda c=color: self.setColor(c, lblkey)# lambda
                            # обработчик события
                                             # (цвет, ссылка на метку)
                    ).pack()

    # функция (метод) обработчик события переключения кнопки
    def setColor(self, color, lblkey):
        lblkey['bg'] = color

root = Tk()

# метка изначально красная ==================================================
lbl = Label(width=25, height=10, bg='red')
lbl.pack()
# ===========================================================================

# Инициализация статической переменной класса RBColor
# (управляющей переменной группы радиокнопок) определённым значением в
# зависимости от класса tkinter
# RBColor.var = IntVar()
# RBColor.var.set(0)

# RBColor.var = StringVar()
# RBColor.var.set('XXX')

RBColor.var = BooleanVar()
RBColor.var.set(True)

# RBColor.var.set(795.123) # при инициализации значения такого типа
                           # НЕ допускаются

"""
При запуске приложения включенной окажется та радиокнопка, значение
опции value которой совпадает с текущим значением переменной RBColor.var.
Если кликнуть по любой другой кнопке, то она включится, а первая кнопка
выключится. При этом значение RBColor.var станет равным значению опции
value этой кнопки независимо от типа переменной RBColor.var .
"""
RBColor('red',   0,    lbl)
RBColor('green', '1',    lbl)
RBColor('blue',  1,    lbl)
```

```
# RBColor('blue',  False,    lbl)
RBColor('yellow', 100.25,  lbl)

# RBColor('red',   'XXX',    lbl)
# RBColor('green', 'YYY',    lbl)
# RBColor('blue',  'ZZZ',    lbl)
# RBColor('yellow', '100',  lbl)
         # цвет,    номер,  метка

root.mainloop()

# ============================================================================
```

Виджет Listbox

От класса Listbox создаются списки – виджеты, внутри которых в столбик перечисляются элементы. При этом можно выбирать один или множество элементов списка.

В Tkinter сначала создается экземпляр Listbox, после этого с помощью метода insert он заполняется.

Первый аргумент в insert - индекс места, куда будет вставлен элемент. Если нужно вставлять в конец списка, то индекс обозначают константой END.

Вторым аргументом передается вставляемый элемент. По умолчанию по мышиному клику в Listbox можно выбирать только один элемент.

Если нужно обеспечить множественный выбор, то для свойства selectmode можно установить значение EXTENDED. В этом режиме, зажав Ctrl или Shift, можно выбрать любое количество элементов.

Если для Listbox необходим скроллер, то он настраивается также как и для текстового поля. В программу добавляется виджет Scrollbar, который и связывается с экземпляром Listbox.

С помощью метода get из списка можно получить один элемент по индексу, или срез, если указать два индекса.

Метод delete удаляет один элемент или срез.

Метод curselection позволяет получить в виде кортежа индексы выбранных элементов экземпляра Listbox.

Далее приводится пример программы с применением методов get, insert, delete и curselection класса Listbox. Первая кнопка добавляет введенную пользователем в текстовое поле строку в список, вторая кнопка удаляет выбранные элементы из списка, третья – сохраняет список в файл. В функции del_list кортеж выбранных элементов преобразуется в список, после чего выполняется его реверс (переворот). Это делается для того, чтобы удаление элементов происходило с конца списка. Иначе приложение неверно бы работало, так как удаление элемента приводило бы к изменению индексов всех следующих за ним. Если же удалять с конца, то индексы впереди стоящих элементов не меняются.

Метод curselection возвращает кортеж. Кортежи не имеют метода reverse, поэтому он преобразуется в список.

В функции save_list кортеж строк-элементов, который вернул метод get, преобразуется в одну строку с помощью строкового метода join через разделитель '\n'. Это делается для того, чтобы элементы списка записались в файл столбиком.

```
# ========================================================================

from tkinter import *

# функции обработки событий нажатия на кнопки.
# В аргументах - ссылки на объект Listbox ================================

def add_item(_box):
    _box.insert(END, entry.get())
    entry.delete(0, END)

def del_list(_box):
    select = list(_box.curselection())  # кортеж индексов выбранных элементов
                                        # преобразуется в список,
    select.reverse()                    # затем переворачивается.

    for i in select:                    # Удаление элементов по индексу
        _box.delete(i)                  # производится с самого большого
                                        # значения из списка выбранных
                                        # индексов.
                                        # Зачем переворот?
                                        # Чтобы удаление выбранных
                                        # элементов ВСЕГДА производилось
                                        # единообразно: со 'старшего'
                                        # элемента списка.

def save_list(_box):
    file = open('list000.txt', 'w')
    file.writelines('\n'.join(_box.get(0, END)))
    file.close()
# ========================================================================

root = Tk()

box = Listbox(selectmode=EXTENDED)
box.pack(side=LEFT)

scroll = Scrollbar(command=box.yview)
scroll.pack(side=LEFT, fill=Y)

box.config(yscrollcommand=scroll.set)

f = Frame()
f.pack(side=LEFT, padx=10)

entry = Entry(f)
entry.pack(anchor=N)

# анонимные кнопки: создаются, размещяются на фрейме (рамке), получают
# надпись, им назначаются лямбда - обработчики события-нажимания,
# которые несут дополнительную информацию о выполняемой операции:
# ссылку на список Listbox (box). Поэтому при их назначении
# соответствующая функция (add_item, del_list, save_list) кодируется
# анонимной функцией с использованием лямбды. Затем эти объекты
# пакуются менеджерами упаковки pack.
Button(f, text='Add', command=lambda b=box: add_item(b)).pack(fill=X)
Button(f, text='Delete', command=lambda b=box: del_list(b)).pack(fill=X)
Button(f, text='Save', command=lambda b=box: save_list(b)).pack(fill=X)
```

```
root.mainloop()

# ========================================================================
```

Listbox – сложный виджет! Кроме рассмотренных, у него имеется ещё много другх методов и свойств.

○ Статические и нестатические методы. Свойства виджетов

Классы можно рассматривать как модули, содержащие переменные со значениями (поля) и функции (методы). Поля и методы называются атрибутами класса.

Когда метод (meth) применяется к объекту – экземпляру класса A (a) (meth(...) вызывается от имени объекта a – a.meth()), ссылка на объект a передается в метод в качестве первого аргумента (self). При этом выражение вызова нестатического метода a.meth() преобразуется к выражению A.meth(a), то есть метод вызывается из "модуля A" в пространстве имен которого находится объявление атрибута meth. В результате оказывается, что meth(...) – это функция, возможно со многими аргументами, которая содержит один обязательный аргумент a.

При объявлении класса в python также можно объявить метод без аргумента self и вообще без параметров параметров и вызывать его только через класс. Такой метод не принимает ссылку на объект данного класса в качестве аргумента.

В ряде языков программирования для таких ситуаций предназначены статические методы, объявляемые в данном классе. В таком случае весь код (объявления методов) находится в классах. Объявление методов (статических методов), которые не содержат объявления ссылки на объект данного класса в качестве обязательного аргумента играют роль обычных функций.

Статические методы в таких языках объявляются с ключевым словом static в заголовке объявления метода. Статические методы могут вызываться через имя класса, содержащего объявление статического метода, либо от имени объектов-представителей данного класса. Но ссылка на сам объект в качестве обязательного аргумента в статические методы не передаётся.

В python также можно объявлять статические методы, которые в этом случае объявляются с декоратором @staticmethod и не содержат объявления аргумента self. Статические методы в python представляют собой обычные функции, помещенные в объявление класса. Возможно, для удобства применения они располагаются в пространстве имен объявляемого класса. В python особой необходимости в статических методах нет, поскольку функция может быть объявлена вне класса, на основной ветке кода.

Объявление класса виджета в python содержит объявление нестатического (с аргументом self) метода с именем cget, который используется для получения настроенных значений объекта:

```
# print("the foreground of xbutton is", xbutton.cget('fg'))



# ========================================================================
from tkinter import *   # импортировать ВСЁ из модуля tkinter !

but0 = None
wAttrExp = None

# ========================================================================

class WidgetsAttributesExpert:
    @staticmethod
```

```
    def get_attributes(widget):

        wdg = widget    # копия аргумента. Это ссылки на виджет
        wdgkeys = wdg.keys()        # список атрибутов виджета

        # непонятно, что здесь даёт копирование ссылки.
        # Значение ссылки НЕ МЕНЯЕТСЯ.
        # Изменить атрибут виджета можно как через копию ссылки,
        # так и непосредственно через значение аргумента.
        # wdg['fg'] = 'blue'
        # widget['fg'] = 'blue'

        # перебор атрибутов виджета и представление
        # атрибутов и их значений в виде таблицы.
        for key in wdgkeys:
            print('Attribute {0:<20}'.format(key), end=' ')

            # получение значений атрибутов виджета и определение их типов
            value = wdg[key]
            wattrtype = type(value)
            # ============================================================

            print('Type: {0:<30} Value: {1}'.format(str(wattrtype), value))

# ============================================================================
# ============================================================================
def fun0(xEvent):

    # Каждый виджет имеет метод с именем cget, который можно использовать
    # для получения настроенных значений: ================================
    #
    # print("the foreground of xbutton is", xbutton.cget('fg'))
    #
    # ============================================================================

    attr0 = but0.cget('fg')
    attr1 = but0.cget('activebackground')
    attr2 = but0.cget('activeforeground')
    print('values... attr0: {0}, attr1: {1}, attr2: {2} '
          .format(attr0, attr1, attr2))

    if attr0 == 'SystemButtonText':
        print('---SystemButtonText---')
    elif attr0 == 'SystemWindowFrame':
        print('===SystemWindowFrame===')
    elif attr0 == 'red' or attr0 == 'darkslategray':
        print('~~~color~~~')


    if but0['fg'] == 'red':
        but0['fg'] = 'darkslategray'
    else:
        but0['fg'] = 'red'

    # область определения атрибута but0['fg']: строка с обозначением
    #                                          цвета (влияет на цвет !)
    #                                               строка со значением
    #                одного из атрибутов виджета (на что влияет ???)
```

```
    #but0['fg'] = 'red'
    #but0['activeforeground'] = 'red'

    if but0['fg'] == 'darkslategray':
       but0['fg'] = 'SystemWindowFrame'

    # Хотелось бы иметь полное представление о назначении каждого атрибута
    # виджета. Его (пока) нет! С ЭТИМ НАДО РАЗБИРАТЬСЯ КОНКРЕТНО !
    #but0['activeforeground'] = 'blue' # здесь должна быть строка с
                                       # обозначением цвета
    #but0['command'] = 'TkDefaultFont'
    but0['command'] = 'qwerty'   # а здесь может быть любая строка ?
    #but0['command'] = 125       # и даже целочисленное значение !

    # ====================================================================
    print('$$$$$$$$$$')
    wAttrExp.get_attributes(but0)
    print('$$$$$$$$$$')
    # ====================================================================

# ========================================================================
# ========================================================================
def fun1(xEvent):
    obj = xEvent.widget           # widget
    name = str(obj)               # name of widget
    foreground = obj.cget('fg')   # value of attribute fg of widget
    #      ВОТ МЕТОД cget ================================================


    print('the value of attribute fg of widget {0} is {1}'.
                                              format(name, foreground))
    #                                         имя виджета, значение
    #                                                      атрибута fg

    # Альтернативная форма записи вывода строки
    # print('the value of attribute fg of widget %s is %s'\
    #                                          %(name, foreground))
    #


# ========================================================================


if __name__ == '__main__':

    wAttrExp = WidgetsAttributesExpert()

    mainWindow = Tk()

    but0 = Button(text='setRED', width=10, height=3)

    but0.bind('<Button-1>',      fun1)
    #but0.bind('<Button-1>',     fun0)
    #but0.bind('<Return>',       fun0)
    #          <'имя_события'>:
    #          нажатие на кнопку, которая кодируется как but0;
    #          нажатие на клавишу Enter (только сначала надо нажать Tab)
    #                  действие (функция-обработчик - одна на оба события)
```

```
    but0.pack()

    # ================================================================

    wAttrExp.get_attributes(but0)

    # ================================================================

    mainWindow.mainloop()

# ====================================================================
```

- **Связать виджет, событие, действие**

Это можно сделать двумя способами:

1. при помощи метода bind (виджет.bind(событие, обработчик)),
2. при помощи свойства command (если это свойство имеется у виджета. В таком случае, соответствующий фрагмент кода имеет вид: Конструктор_виджета(command=lambda ...))

Далее приводится пример, который представляет изменение свойства кнопки (цвета надписи) по нажатию на эту кнопку, изменение шрифта метки по щелчку мышиными кнопками по этой метке:

- клик левой кнопкой мыши по метке устанавливает для нее один шрифт,
- клик правой кнопкой мыши – другой.

Как вариант, для каждого из шрифтов можно написать две разные функции (что не есть хорошо) ...

```
# ====================================================================
# связать виджет, событие и действие можно:
# 1. при помощи метода bind (виджет.bind(событие, обработчик)) или
# 2. при помощи свойства command (если это свойство имеется у виджета
#                                 Конструктор_виджета(command=lambda …))
# ====================================================================
# изменение свойства кнопки (цвета надписи) по нажатию на эту кнопку
# изменение шрифта метки по щелчку мышиными кнопками по этой метке
# клик левой кнопкой мыши по метке устанавливает для нее один шрифт,
# клик правой кнопкой мыши – другой.
# Как вариант, для каждого из шрифтов можно написать две разные функции
#                                                 (что не есть хорошо)

from tkinter import *

lbl = None

# Один виджет (одна метка) реагирует на ДВА СОБЫТИЯ:
# По щелчку мышиной кнопкой по метке меняется шрифт.
# функции обработчики событий.
# свойству ['font'] метки lbl присваивается строка с названием шрифта ====

def setFont1(event):
    lbl['font'] = 'Verdana'
    lbl['bg'] = 'darkseagreen'

def setFont2(event):
    lbl['font'] = 'Times'
```

```
    lbl['bg'] = 'brown'

# ========================================================================

if __name__ == '__main__':
    mainwindow = Tk()

    lbl = Label(text = 'Master Label')

    # Назначение виджету обработчиков событий ============================
    lbl.bind('<Button-1>', setFont1)
    #        по левому щелчку
    #                       функция setFont1
    lbl.bind('<Button-3>', setFont2)
    #        по правому щелчку
    #                       функция setFont2

    lbl.pack()

    mainwindow.mainloop()

# ========================================================================
```

Это работает, НО функции обработки здесь практически идентичны. А было бы хорошо иметь одну функцию обработки, в которую как аргумент передавалось бы название шрифта. Например:

```
def changeFont(event, font):
   lbl['font'] = font
```

Однако есть проблема! Метод bind имеет строго ДВА аргумента: '<имя_события>',

ссылка_на_функцию-обработчик

(НО НЕ НА ВЫРАЖЕНИЕ ВЫЗОВА)

поэтому НЕЛЬЗЯ написать так:

```
lbl.bind('<Button-3>', setFont2(event, 'Times'))# выражение вызова должно
                                                # выполняться.
```

При этом если в функции нет оператора return, то она возвращает None. Поэтому получается, что даже если в выражении вызова все аргументы передать правильно, то в метод bind попадёт None, а не объект-функция.

В анонимном объекте-функции Python, который объявляется с помощью инструкции lambda, можно запаковать функцию вместе с аргументом. В конкретном случае, как то так:

```
lbl.bind('<Button-1>', lambda e, f='Verdana':changeFont(event, font))
lbl.bind('<Button-3>', lambda e, f='Times':changeFont(event, font))
```

Лямбда-функции можно использовать не только с методом bind, но и опцией command, которая имеется у ряда виджет.

Если функция передаётся через опцию command, то ей не нужен параметр event. Так этот параметр и при использовании метода bind тоже не всегда (очень редко) применяется. А с аргументом command обрабатывается только одно основное событие для виджета – клик левой кнопкой мыши.

```
# ================================================================================

from tkinter import *

lbl = None

# функция обработки события
def changeFont(font):    # Эта функция передаётся через опцию command.
                         # Ей не нужен параметр event.
    lbl['font'] = font

if __name__ == '__main__':

    mainwindow = Tk()

    lbl = Label(text = 'Master Label')
    lbl.pack()

"""
две безымянных кнопки (одна над друой). События не обозначены
(предполагается клик левой мышиной кнопкой по виджету).
функция-обработчик - ранее объявленная changeFont с параметром fnt
(строка с именем фонта), которая кодируется анонимным
объектом-функцией Python.
Этот объект вместе с состветствующим значением аргумента
параметра fnt) объявляется с помощью инструкции lambda.

Метод pack, применяемый к определённым в приложении виджетам
(метка lbl и безымянные кнопки), обеспечивает размещение всех виджетов
в окне mainwindow. В случае с кнопками метод pack() применяется
(вызывается) непосредственно при создании виджетов.
"""

    Button(command=lambda fnt='Verdana': changeFont(fnt)).pack()

    Button(command=lambda fnt='Times': changeFont(fnt)).pack()

    # ============================================================================

    mainwindow.mainloop()    #  Далее от имени объекта mainWindow вызывается
    # функция mainloop. Эта функция вызывает бесконечный цикл окна, при
    # котором окно (пока оно не будет закрыто) будет ждать любого
    # (объявленного) взаимодействия с пользователем.
    # В данном случае - левого мышиного щелчка по кнопке приложения.

    # ============================================================================
```

○ События и типы событий

Чтобы приложение с графическим интерфейсом что-либо делало, должны происходить те или иные события. Обычно эти события представляют собой воздействие пользователя на элементы графического интерфейса.

Можно выделить три основных типа событий:

- производимые мышью,
- как результат нажатия клавиш на клавиатуре,
- возникающие в результате изменения виджетов.

Возможны комбинированные события, представляющие собой сочетания основных типов. Например, клик мышью с зажатой клавишей на клавиатуре.

- щелчки кнопками мыши,
- вход курсора в пределы виджета,
- выход курсора за пределы виджета,
- движение мышью.

- нажатие клавиши,
- нажатие комбинации клавиш,
- ОТЖАТИЕ клавиши.

- ввод данных,
- изменение размеров окна,
- покрутка,
- другие ...

Событие и метод bind

Метод bind, назначает обработчик события виджету, событие передается в качестве первого аргумента.

```
widget.bind(event, function)
```

Название события заключается в кавычки, а также в угловые скобки < и >. События описывается с помощью зарезервированных ключевых слов. Например, события, производимые мышью:

- <Button-1> – клик левой кнопкой мыши
- <Button-2> – клик средней кнопкой мыши
- <Button-3> – клик правой кнопкой мыши
- <Double-Button-1> – двойной клик левой кнопкой мыши
- <Motion> – движение мыши

и т. д. (полного списка на момент написания этого материала ПОКА нет)

У событий как объектов tkinter, как и у многих других объектов, есть атрибуты.

В следующем примере в функции move извлекаются значения атрибутов x и y объекта event, в которых хранятся координаты местоположения курсора мыши в пределах виджета, по отношению к которому было сгенерировано событие.

Пример:

в зависимости от того, двигается мышь, щелкают левой или правой кнопкой, меняется надпись в заголовке главного окна.

В данном случае виджетом является главное окно, а событием – <Motion>, т. е. перемещение мыши.

```
# ========================================================================
from tkinter import *

# функции обработки событий.
# Информация о действиях с мышиными кнопками (нажимания) и
# самой мышью (перемещения) отображаются в заголовке
# главного окна приложения
# ========================================================================
def b1(event):
    root.title("Левая кнопка мыши")

def b3(event):
    root.title("Правая кнопка мыши")

def move(event):
    x = event.x
    y = event.y
    s = "Движение мышью {}x{}".format(x, y)
    root.title(s)
# ========================================================================

root = Tk()
root.minsize(width=500, height=400)

# назначение простых функций-обработчиков непосредственно с помощью ссылки
# на функцию. При этом в обработчиках приходится явно указывать ссылку на
# главное окно приложения (объект root)

root.bind('<Button-1>', b1)
root.bind('<Button-3>', b3)
root.bind('<Motion>', move)

root.mainloop()
# ========================================================================
```

Выше рассмотренное приложение обладает одной особенностью: используемый здесь механизм назначения функции обработки события НЕ допускает дополнительных аргументов (НИЧЕГО КРОМЕ event !). Поэтому непосредственно в теле функций обработки приходится ЯВНЫМ ОБРАЗОМ использовать ссылку на главное окно приложения (объект root).

ЯВНО НЕ использовать эту ссылку в функциях обработки можно за счёт определения новых функций обработки с дополнительным аргументом, который включается в выражение назначения обработчика (аргумент метода bind) с помощью lambda-выражения.

```
# ========================================================================
from tkinter import *

# функции обработки событий.
# Информация о действиях с мышиными кнопками (нажимания) и
# самой мышью (перемещения) отображаются в заголовке
# главного окна приложения
# ========================================================================
def _b1(event, rt):
    rt.title("rt... Левая кнопка мыши")

def _b3(event, rt):
```

```
    rt.title("rt... Правая кнопка мыши")

def _move(event, rt):
    x = event.x
    y = event.y
    s = "rt... Движение мышью {}x{}".format(x, y)
    rt.title(s)
# ================================================================================

root = Tk()
root.minsize(width=500, height=400)

# назначение функций-обработчиков с дополнительным аргументом с помощью
# анонимной функции (lambda)

root.bind('<Button-1>', lambda event, rt=root: _b1(event, rt))
root.bind('<Button-3>', lambda event, rt=root: _b3(event, rt))
root.bind('<Motion>', lambda event, rt=root: _move(event, rt))

root.mainloop()
# ================================================================================
```

■ Событие (Event) как объект tkinter

Далее в приложении выводится информация об объекте Event и некоторых его свойствах. Все атрибуты события также можно посмотреть с помощью команды dir(event). У разных событий они одни и те же, меняются только значения. Для тех или иных событий часть атрибутов не имеет смысла, такие свойства имеют значения по умолчанию.

Объект event среди своих атрибутов имеет ссылку на координаты курсора в главном окне приложения (event.x_root, event.y_root). Поэтому можно НЕ беспокоиться относительно аргумента - ссылки на объект root (главное окно приложения)

В примере, хотя обрабатывается событие нажатия клавиши клавиатуры ('a'), координаты положения на экране мышиного курсора (x, y) сохраняются в поля объекта:

- event.x_root,
- event. y_root.

```
# ================================================================================
from tkinter import *

# обработчик события - информатор об аргументах события (объекта event,
# а на самом деле имя объекта - события НЕ важно!) Что event, что xevent -
# БЕЗ РАЗНИЦЫ ! ============================================================
def event_info(xevent):
    print(type(xevent))
    print(xevent)
    print(xevent.time)
    print(xevent.x_root)
    print(xevent.y_root)


root = Tk()
root.bind('a', event_info)     # здесь событие с клавиатуры. для них
                               # буквенные клавиши можно записывать
```

```
                                # без угловых скобок (например, 'a').
                                # Кроме того, объект event (xevent)
                                # среди своих атрибутов имеет ссылку на
                                # координаты курсора в главном окне
                                # приложения (event.x_root, event.y_root).
                                # Поэтому можно НЕ беспокоиться относительно
                                # аргумента - ссылки на объект root
                                # (главное окно приложения)
root.mainloop()

# ================================================================================
```

В следующем примере в функции обработки напрямую задействована ссылка на главное окно приложения (атрибут rt) поэтому при назначении обработчика применяется lambda, так как функция обработки получает дополнительный аргумент. Для определения значений координат курсора в окне rt применяются СПЕЦИАЛЬНЫЕ методы:

- rt.winfo_pointerx(),
- rt.winfo_pointery().

■

```
# ================================================================================
from tkinter import *

# обработчик события - информатор об аргументах события (объекта event)
def event_info(event, rt):
    print(type(event))
    print(event)
    print(event.time)
    print(rt.winfo_pointerx())
    print(rt.winfo_pointery())


root = Tk()
root.bind('a', lambda event, rt = root: event_info(event, rt))
                                # здесь событие с клавиатуры. для них
                                # буквенные клавиши можно записывать
                                # без угловых скобок (например, 'a').
                                # Кроме того, напрямую задействована ссылка
                                # на главное окно приложения (атрибут rt)
                                # поэтому при назначении обработчика
                                # используется lambda, а для определения
                                # значений координат курсора в окне
                                # tr - методы rt.winfo_pointerx(),
                                #             rt.winfo_pointery()

root.mainloop()

# ================================================================================
```

События с клавиатуры, которые НЕ имеют явного буквенного представления, кодируются названием события в угловых скобках, заключённым в кавычки (строка с названием события в угловых скобках).

Например, '<Return>' - нажатие клавиши Enter, '<space>' – нажатый пробел.

При этом имеется событие '<Enter>', которое НЕ имеет отношения к нажатию клавиши Enter, а происходит, когда курсор заходит в пределы виджета.

Далее в приложении две метки используют одну и ту же функцию, и каждая метка использует эту функцию для обработки разных событий: ввода курсора в пределы виджета и вывода за его границы. Функция обработки, в зависимости от того, по отношению к какому виджету было зафиксировано событие, изменяет свойства только этого виджета. Способ изменения зависит от произошедшего события.

Свойство event.widget содержит ссылку на сгенерировавший событие виджет.

Свойство event.type описывает, что это за событие.

```
# ================================================================================
from tkinter import *
# функция обработки нескольких событий для нескольких виджетов ===========
def enter_leave(event):
    # у event'а есть ссылка на widget, куда пишется текст 'In' или 'Out'
    if event.type == '7':     # Указатель мыши был помещён в видимую
                              # часть виджета.
        event.widget['text'] = 'In'
    elif event.type == '8':   # Указатель мыши был перемещён из виджета.
        event.widget['text'] = 'Out'
# ================================================================================

root = Tk()

# разные виджеты, разные события, одна общая функция обработки!

lab1 = Label(width=20, height=3, bg='white')

lab1.pack()

# назначение функции обработчика события
lab1.bind('<Enter>', enter_leave)
lab1.bind('<Leave>', enter_leave)

lab2 = Label(width=20, height=3, bg='black', fg='white')

lab2.pack()

# назначение функции обработчика события
lab2.bind('<Enter>', enter_leave)
lab2.bind('<Leave>', enter_leave)

root.mainloop()
# ================================================================================
```

У каждого события есть имя и номер (тип). С помощью выражения print(repr(event.type)) можно просмотреть его полное описание. При этом на одних платформах str(event.type) возвращает имя события (например, 'Enter'), на других – строковое представление номера события (например, '7').

▪ Применение событий

В следующем примере демонстрируется применение одного обработчика событий для нескольких виджетов.

У event'а есть ссылка на widget (разные объекты), куда пишется текст 'In' или 'Out' в зависимости от положения курсора. Указатель мыши может быть либо помещён в видимую часть виджета, либо успешно перемещён из неё.

В приложении задействована общая функция обработки для разных виджетов и разных событий.

```
# ======================================================================
from tkinter import *

# функция обработки нескольких событий для нескольких виджетов ==========
def enter_leave(event):
    # у event'а есть ссылка на widget, куда пишется текст 'In' или 'Out'
    if event.type == '7':     # Указатель мыши был помещён в видимую
                              # часть виджета.
        event.widget['text'] = 'In'
    elif event.type == '8':   # Указатель мыши был перемещён из виджета.
        event.widget['text'] = 'Out'
# ======================================================================

root = Tk()

# разные виджеты, разные события, одна общая функция обработки!

lab1 = Label(width=20, height=3, bg='white')

lab1.pack()

# назначение функции обработчика события
lab1.bind('<Enter>', enter_leave)
lab1.bind('<Leave>', enter_leave)

lab2 = Label(width=20, height=3, bg='black', fg='white')

lab2.pack()

# назначение функции обработчика события
lab2.bind('<Enter>', enter_leave)
lab2.bind('<Leave>', enter_leave)

root.mainloop()
# ======================================================================
```

- **root.after: альтернативные способы запуска и объявления**

Без root.after() в функции outfun приложение не работает.

Метод after запускает функцию, указанную во втором аргументе, через промежуток времени, указанный в первом аргументе, со значением параметра для выполняемой функции, заданного в третьем аргументе

```
# ======================================================================
"""
 Без root.after() в функции outfun приложение не работает.
 Метод after запускает функцию, указанную во втором аргументе,
 через промежуток времени, указанный в первом аргументе,
 со значением аргумента для выполняемой функции, заданного
"""
```

```
# в третьем аргументе ================================================

from tkinter import *

# функция externalfun объявлена во внешней области объявления
def externalfun(key):
    print('~~~~~externalfun({0})~~~~~'.format(key))

def outfun(outkey):
 # функция infun объявлена во внутренней области объявления
    def infun(inkey):
    print('=====infun({0})====='.format(inkey))

    # в функции outfun методом root.after запускаются две функции,
    # объявленные в разных областях объявления, с разными значениями
    # аргументов
    root.after(0, infun, outkey)
    root.after(0, externalfun, outkey*2)

# ====================================================================
root = Tk()
outfun(25)

root.mainloop()
# ====================================================================
```

Аналогичные способы объявления и запуска функций при разработке графического интерфейса.

Здесь сочетание клавиш Ctrl+a выделяет текст в поле. Метод after выполняет функцию, указанную во втором аргументе, через промежуток времени (?), указанный в первом аргументе. В третьем аргументе передаётся значение атрибута widget объекта event. В данном случае им будет поле ent. Именно оно будет передано как аргумент в функцию select_all2 и присвоено параметру widget.

```
"""
 Здесь сочетание клавиш Ctrl+a выделяет текст в поле.
 Метод root.after выполняет функцию, указанную во втором аргументе, через
 промежуток времени, указанный в первом аргументе. В третьем аргументе
 передается значение атрибута widget объекта event. В данном случае им
 будет значение поля ent. Именно оно будет передано как аргумент
 в функцию select_all2 и присвоено аргументу widget.
"""
# ====================================================================

from tkinter import *

# функции обработчики событий ========================================

# аргумент event в коде не применяется
def exit_win(event):
   root.destroy()

# аргумент event в коде не применяется
def to_label(event):
   t = ent.get()
   lbl.configure(text=t)
```

```
# И это ТОЖЕ функция обработки события ================================
# аргумент event применяется в коде функции ===========================
def select_all(event):

    # область объявления функции select_all2 -
    # тело функции select_all, в которой функция select_all2 объявляется
    # ====================================================================
   def select_all2(widget):
       widget.selection_range(0, END)
       widget.icursor(END)  # курсор в конец
   # ====================================================================

   """
    root.after выполняет
    функцию, указанную во втором аргументе, через промежуток времени,
    указанный в первом аргументе. В третьем аргументе передается
    значение атрибута widget объекта event.
    В данном случае это widget поле ввода ent.
   """
   root.after(10, select_all2, event.widget)
# ====================================================================

# Вариант объявления функции select_all2:
# общая область объявления ===========================================
# def select_all2(widget):
#     widget.selection_range(0, END)
#     widget.icursor(END)  # курсор в конец
# ====================================================================

root = Tk()

# поле ввода ==================================
ent = Entry(width=40)
ent.pack()
ent.focus_set() # установка курсора на поле ввода ent (фокус ввода на ent)

# метка =======================================
lbl = Label(height=3, fg='orange', bg='darkgreen', font="Verdana 24")
lbl.pack(fill=X)

# назначение функций обработки событий ================================
ent.bind('<Return>', to_label)
ent.bind('<Control-a>', select_all)
root.bind('<Control-q>', exit_win)
# ====================================================================

root.mainloop()

# ====================================================================
```

■ Модуль tkinter.ttk

В состав пакета tkinter входит модуль ttk, содержащий классы виджетов разных стилей. Их внешний вид по умолчанию зависит от конкретной операционной системы. Далее в коде для сравнения внешнего вида размещено несколько пар объявлений аналогичных виджетов из модулей tkinter и tkinter.ttk.

```
# ========================================================================
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()

tk.Button(text="Hello").pack()
ttk.Button(text="Hello").pack()

tk.Checkbutton(text="Hello").pack()
ttk.Checkbutton(text="Hello").pack()

tk.Radiobutton(text="Hello").pack()
ttk.Radiobutton(text="Hello").pack()

root.mainloop()
# ========================================================================
```

Так как имена классов аналогичных виджетов совпадают, модули импортируются так, чтобы их пространства имен не перекрывались. При этом конструкторы в классах при создании объектов (виджетов) вызываются через имена (в данном случае псевдонимы) модулей. Если же НЕ нужны виджеты модуля tkinter, подобные которым есть в tkinter.ttk, удобнее импортировать всё пространство имен как одного, так и второго модуля.

```
# ========================================================================
from tkinter import *
from tkinter.ttk import *

root = Tk()

b = Button(text="Hello")
b.pack()

t = Text(width=10, height=5)
t.pack()

print(root.__class__)
print(b.__class__)
print(t.__class__)

root.mainloop()

# ========================================================================
```

В данном случае пространство имен tkinter.ttk, который импортируется вторым, перекрывает часть имен tkinter. Поэтому выражение Button(text="Hello") вызывает конструктор класса Button, находящийся в модуле tkinter.ttk.

В то же время в этом модуле нет классов Tk и Text. Поэтому экземпляры (виджеты) этих классов создаются от базовых классов tkinter. В консоль будет выведено:

<class 'tkinter.Tk'>

<class 'tkinter.ttk.Button'>

<class 'tkinter.Text'>

Наборы виджетов обоих модулей перекрываются не полностью:

- есть общие (такие как Label, Button, Entry),
- есть характерные только для tkinter (например, Listbox и Canvas),
- и только для ttk (например, Combobox, Notebook).

Проблема состоит в том, что в ttk свойства виджетов программируются не совсем так, как в tkinter.

Если в приложении одновременно сочетаются элементы обоих модулей, код программы становится более запутанным и менее понятным.

Так, для виджетов из tkinter.ttk многие свойства задаются с помощью объекта, созданного от класса ttk.Style. Основная идея ttk – отделить оформление виджета от описания его поведения. Далее для сравнения установки свойств для двух кнопок импортируются модули без перекрытия пространства их имен.

```
# ================================================================================

from tkinter import *
from tkinter import ttk

root = Tk()

b_tk = Button(text="Hello Tk")
b_ttk = ttk.Button(text="Hello Ttk")

b_tk.config(background="#b00",
            foreground="#fff")

style = ttk.Style()
style.configure("TButton",
                background="#0b0",
                foreground="#fff")

b_tk.pack(padx=10, pady=10)
b_ttk.pack(padx=10, pady=10)
root.mainloop()

# ================================================================================
```

свойства для экземпляра Button из модуля tkinter задаются с помощью метода кнопки config. Однако то же самое можно было бы сделать при передаче значения в конструктор:

```
b_tk = Button(text="Hello Tk",
              background="#b00",
              foreground="#fff")
```

Настроить так кнопку из модуля ttk нельзя. Вместо этого нужно создать экземпляр от класса Style и уже через него изменять свойства по умолчанию. В коде выше был применён метод configure. Здесь первым аргументом в него передается имя стиля (TButton), связанного с классом объектов, для которых производится настройка. В данном случае это кнопки. Для уточнения имени стиля можно воспользоваться методом winfo_class:

```
print(ttk.Label().winfo_class())
```

```
print(ttk.Button().winfo_class())
...
```

Результат:

TLabel

TButton

▪ Создание собственного стиля

Что делать, если в программе нужны, например, кнопки разного стиля? При этом можно создать собственный стиль, унаследовав его от исходного, и изменять лишь отдельные его свойства. Далее в примере созданный стиль будет называться "XXX".

После точки указывается его родитель (в данном случае это стиль TButton). При создании первой кнопки через опцию style указывается применяемый стиль. Для второй кнопки по умолчанию используется стиль TButton. Он не менялся, хотя его можно было бы изменить.

На самом деле особой необходимости изменять внешний вид виджетов нет. Напротив, в приложениях приветствуется стандартный и единообразный внешний вид и поведение виджетов. Это дает пользователю интуитивно понятный интерфейс.

```
# ================================================================
from tkinter import *
from tkinter.ttk import *

root = Tk()

style = Style()
style.configure("XXX.TButton", foreground="green")

Button(text="First", style="XXX.TButton").pack()
Button(text="Second").pack()

root.mainloop()
# ================================================================
```

▪ Тема приложения

Кроме того, в ttk есть разные темы. Можно менять не отдельные виджеты, а целиком тему оформления приложения. С помощью методов theme_names и theme_use можно выяснить список тем (который зависит от данной ОС) и текущую тему:

```
# ================================================================
from tkinter.ttk import *

style = Style()
print(style.theme_names())
print(style.theme_use())
# ================================================================
```

Результат:

('clam', 'alt', 'default', 'classic')

default

Если в метод theme_use передать название темы, она будет применена. Далее в программе при нажатии клавиши Enter изменяется тема приложения:

```
from tkinter import *
from tkinter.ttk import *

root = Tk()
style = Style()

e = Entry(justify='center')
e.pack()
Button(text="Hello").pack()
Label(text="Hello").pack()

themes = style.theme_names()
i = 0

def theme_next(event):
    global i
    style.theme_use(themes[i])
    e.delete(0, END)
    e.insert(0, themes[i])
    if i < len(themes) - 1:
        i += 1
    else:
        i = 0

root.bind('<Return>', theme_next)
root.mainloop()
```

- **Свойства виджетов в ttk**

Описание свойств виджетов в ttk не всегда совпадает с тем, как это делается в базовом tkinter.

Ранее в примере с красной и зеленой кнопками с помощью опций foreground и background устанавливались цвета текста и фона кнопки, когда она не находилась под курсором мыши (то есть когда она не активна).

Чтобы переопределить цвета по умолчанию при наводке курсора, для кнопок tkinter надо использовать опции activeforeground и activebackground. Однако эти свойства не работают для виджетов из ttk:

```
b_tk.config(background="red",
            foreground="white",
            activebackground="orange",
            activeforeground="white")

style = ttk.Style()
style.configure("TButton",
                background="green",
                foreground="white",
                activebackground="lightgreen",
                activeforeground="black")
```

Этот код выполнится без ошибок. Но фон зеленой кнопки при нажатии на нее останется светло-серым, и не станет светло-зеленым. В данном случае вместо метода configure следует использовать метод map, который вызывается объектами типа Style:

```
# ================================================================================
from tkinter import *
from tkinter.ttk import *

root = Tk()

Button(text="Hello World").pack(
    padx=40, pady=40,
    ipadx=20, ipady=20
)

st = Style()
st.map('TButton',
       foreground=[('!active', 'purple'),
                   ('pressed', 'orange'),
                   ('active', 'red')],
       background=[
           ('pressed', 'brown'),
           ('active', 'white')]
       )

root.mainloop()
# ================================================================================
```

Создание графических интерфейсов с помощью библиотеки Tkinter

Tkinter графическая кроссплатформенная библиотека на основе средств Tk. Свободное ПО, включено в стандартную библиотеку языка программирования Python. В состав Tkinter входит много компонентов.

Один из них — Canvas («Холст»).

Canvas

В tkinter от класса Canvas создаются объекты-холсты, на которых можно "рисовать", размещая различные фигуры и объекты.

Canvas применяется для вывода графических примитивов:

- линий,
- прямоугольников,
- эллипсов,
- текста,
- окон,
- изображений.

Это делается это с помощью вызовов соответствующих методов. При создании объекта Canvas указывается его ширина и высота.

Методы позиционирования элементов

При работе с Tkinter для позиционирования элементов применяются разные методы:

- pack();
- place();
- grid().

При размещении (рисовании) геометрических примитивов и других объектов указываются их координаты на холсте. При этом точкой отсчета является верхний левый угол.

Линии

Линии - это примитивные геометрические элементы. В Canvas линия с нужным размером создаётся через метод create_line().

Параметры метода create_line() — это координаты x и y. Они обозначают начальные и конечные точки будущего линейного отрезка.

Опция dash рисует пунктирную линию. У этой опции есть собственные значения, помещенные в скобки. В примере это (4, 2).

- цифра 4 обозначает длину тире (точки) в пикселях;
- цифра 2 отвечает за ширину (width) пустого промежутка, существующего между тире (точками).

Если прописать dash=(1, 1), получается линия из точек. При рисовании обычной линии в качестве координат можно указать несколько конечных точек. Фрагмент кода ниже отвечает за отрисовку треугольника, который также состоит из простых линий.

Далее в программе создается холст. На нём с помощью метода create_line рисуются отрезки. Аргументы create_line() — это координаты x и y. Они обозначают начальные и конечные точки будущего линейного отрезка. Опция dash рисует пунктирную линию. У этой опции есть собственные значения, помещенные в скобки. В примере это (4, 2):

- цифра 4 обозначает длину тире (точки) в пикселях;
- цифра 2 отвечает за ширину (width) пустого промежутка, существующего между тире (точками).

Если прописать dash=(1, 1), получится линия из точек.

```
from tkinter import Tk, Canvas, Frame, BOTH

class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title('this is the lines')
        self.pack(fill=BOTH, expand=1)

        canvas=Canvas(self)
        canvas.create_line(15,25,  200,25)
        canvas.create_line(300, 35, 300, 200, dash=(4, 2))
        canvas.create_line(55, 85, 155, 85, 105, 180, 55, 85)

        canvas.pack(fill=BOTH, expand=1)

def main():
    root=Tk()
    ex=Example()
    root.geometry('400x250+300+300')
    root.mainloop()

if __name__=='__main__':
    main()
```

При рисовании обычной линии в качестве координат можно указать несколько конечных точек. Кусочек кода ниже отвечает за отрисовку треугольника, который также состоит из простых линий. Сначала указывают координаты начала (x1, y1), затем – конца (x2, y2) отрезка.

```
# ======================================================================
# в программе создается холст. На нём рисуются отрезки с помощью
# метода create_line. Сначала указываются координаты начала (x1, y1),
# затем - конца (x2, y2) отрезка. ======================================

from tkinter import *

root = Tk()
```

```
c = Canvas(root, width=200, height=200, bg='white')
c.pack()

c.create_line(10,
              10,
              190,
              50)

c.create_line(# НЕименованные аргументы:
              # их порядок следования в выражении вызова метода ВАЖЕН
              100,
              180,
              100,
              60,
              # именованные аргументы:
              # их порядок следования в выражении вызова метода НЕ ВАЖЕН
              width=5,
              arrow=LAST,
              dash=(10, 2),
              arrowshape="10 20 10",
              fill='green',
              activefill='lightgreen' # activefill определяет цвет
                                      # отрезка при наведении на него
                                      # мышиного курсора.
              )

root.mainloop()

# ======================================================================
```

Методы, которые позволяют создавать различные фигуры на канве:

- create_rectangle,
- create_polygon,
- create_oval,
- create_arc,
- create_text.

■ create_rectangle: создание прямоугольников

```
c.create_rectangle(10, 10, 190, 60) # Первые две координаты -
                                    # верхний левый угол, вторые - правый нижний.
```

create_rectangle создаёт прямоугольник.

```
# ======================================================================
from tkinter import Tk, Canvas, Frame, BOTH

class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title('this is the colors')
 self.pack(fill=BOTH, expand=1)
```

```
        canvas=Canvas(self)    # создается виджет Canvas

"""
 на канве методом create_rectangle() создаётся прямоугольник.
 Здесь надо прописать 4 параметра для определения
 координатного положения левого верхнего угла
 (верхней левой ограничительной точки) и
 координатного положения правого нижнего угла
 (нижней правой ограничительной точки),
 параметр outline задаёт цвет контура прямоугольника,
 параметр fill задаёт цвет
 внутренней области прямоугольника.
"""
        canvas.create_rectangle(30, 10,  120,80, outline='#fb0',
                                                 fill='#fb0')
        canvas.create_rectangle(150,10,  240,80, outline='#f50',
                                                 fill='#f50')
        canvas.create_rectangle(270,10,  370,80, outline='#05f',
                                                 fill='#05f')

        canvas.pack(fill=BOTH, expand=1)
        # ================================================================

def main():
    root=Tk()
    ex=Example()
    root.geometry('400x250+300+300')
    root.mainloop()

if __name__=='__main__':
    main()
# ======================================================================
```

■ Цветные прямоугольники

Цвет — это объект, отображающий комбинацию трех цветов (красного, зеленого, синего — RGB). Ниже на канве нарисованы прямоугольники. Они закрашены в разные цвета.

create_polygon создаёт произвольный многоугольник

```
# ======================================================================

from tkinter import *

root = Tk()

c = Canvas(root, width=200, height=200, bg='white')
c.pack()

# В этом примере, когда на прямоугольник попадает курсор мыши, его рамка
# становится пунктирной, и это определяется свойством activedash.
c.create_rectangle(60, 80, 140, 190,
                   fill='yellow',
                   outline='green',
                   width=3,
                  activedash=(5, 4)  # пунктирная рамка, когда на
```

```
                                # прямоугольник попадает мышиный курсор
                    )

# Методом create_polygon рисуется произвольный многоугольник путем задания
# координат каждой его точки:

c.create_polygon(100, 10, 20, 90, 180, 90)

c.create_polygon(40, 110, 160, 110,
                 190, 180, 10, 180,
                 fill='orange', outline='black')

# координаты точек можно заключать в скобки:

c.create_polygon((40, 110), (160, 110),
                 (190, 180), (10, 180),
                 fill='orange', outline='black')

root.mainloop()

# ================================================================================
```

■ create_oval: создание эллипсов

create_oval создает эллипсы. При этом задаются координаты гипотетического прямоугольника, описывающего эллипс. Если нужно получить круг, то соответственно описываемый прямоугольник должен быть квадратом.

```
c.create_oval(50, 10, 150, 110, width=2)
c.create_oval(10, 120, 190, 190, fill='grey70', outline='white')
```

метод create_arc: в зависимости от значения опции style можно получить

- сектор (по умолчанию),
- сегмент (CHORD),
- дугу (ARC).

Также как в случае create_oval координаты задают прямоугольник, в который вписана окружность (или эллипс), из которой "вырезается" сектор, сегмент или дугу. Опции start присваивается градус начала фигуры, extent определяет угол поворота.

```
# ================================================================================

from tkinter import *

root = Tk()

c = Canvas(root, width=200, height=200, bg='white')
c.pack()

# ================================================================================

c.create_oval(10, 10, 190, 190, fill='lightgrey', outline='white')
```

```
c.create_arc(10, 10, 190, 190, start=0, extent=45, fill='red')

c.create_arc(10, 10, 190, 190, start=180, extent=25, fill='orange')

c.create_arc(10, 10, 190, 190, start=240,
                               extent=100,
                               style=CHORD,
                               fill='green')

c.create_arc(10, 10, 190, 190, start=160,
                               extent=-70,
                               style=ARC,
                               outline='darkblue',
                               width=5)
# ============================================================================

root.mainloop()

# ============================================================================
```

■ Сложные формы

Можно без проблем нарисовать круг, овал и прочие фигуры, включая криволинейные. Именно для этой цели и предназначен код ниже:

```
# ============================================================================
from tkinter import Tk, Canvas, Frame, BOTH

class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title('this is the forms')
        self.pack(fill=BOTH, expand=1)

        canvas=Canvas(self)    # создается виджет Canvas

# у всех фигур контур (outline) окрасится в красный цвет, заливка (fill)
# будет зеленой. Ширина контура (width) — 2 px.
# круг - create_oval... Первые 4 параметра нужны для указания
# ограничивающих координат:
#       x0, y0, x1, y1 - являются координатами правой нижней и верхней
#       левой точек квадрата, где помещен круг.
#       Аналогично создается и овал — меняются лишь первые 4 параметра
#       ============================================================

        canvas.create_oval(10, 10, 100, 100,
                           outline='#f11',
                           fill='#1f1',
                           width=2)
        canvas.create_oval(110, 10, 210, 75,
                           outline='#f11',
                           fill='#1f1',
                           width=2)
```

```
#       прямоугольник тоже имеет координатные значения x и y, которые
#       являются ограничительными точками
        canvas.create_rectangle(250,10, 300,70,
                                outline='#f11',
                                fill='#1f1',
                                width=2)

#       дуга (арка) это часть круга. Надо указать ограничительные
#       координаты дуги:
#       первые 4 параметра x0, y0, x1, y1 - являются координатами
#       правой нижней и верхней левой точек квадрата,
#       где помещен дуга, параметр start определяет угол дуги,
#       параметр extent устанавливает размер угла ========================
        canvas.create_arc(25, 250, 100, 120,
                          start=0, extent=210,
                          outline='#f11', fill='#1f1',
                          width=2)

#       многоугольник: много углов, и соответствующее количество
#       координатных значений в массиве points
        points=[
        150,100,
        200,120,
        240,180,
        210,200,
        150,150,
        100,200
        ]
        canvas.create_polygon(points,
                              outline='#f11',
                              fill='#1f1',
                              width=2)

        canvas.pack(fill=BOTH, expand=1)

def main():
    root=Tk()
    ex=Example()
    root.geometry('400x250+300+300')
    root.mainloop()

if __name__=='__main__':
    main()

# ================================================================================
create_text: размещение текста на канве
```

С помощью метода create_text на канве (холсте) размещается текст:

```
c.create_text(100, 100, text="Hello World,\nPython\nand Tk",
                        justify=CENTER,
                        font="Verdana 14")
c.create_text(200, 200, text="About this",
                        anchor=SE,
                        fill="grey")
```

По умолчанию в заданной координате располагается центр текстовой надписи.

Чтобы изменить это и (например) разместить по указанной координате левую границу текста, надо использовать якорь со значением W (от англ. west – запад). Другие значения якоря:

N, NE, E, SE, S, SW, W, NW.

Если букв, задающих сторону привязки, две, то вторая определяет вертикальную привязку (вверх или вниз "уйдет" текст от заданной координаты). Свойство justify определяет выравнивание текста относительно себя самого.

```
# ================================================================

from tkinter import *

root = Tk()

c = Canvas(root, width=200, height=200, bg='white')
c.pack()

c.create_text(100, 100, text="Hello World,\nPython\nand Tk",
                        justify=CENTER,
                        font="Verdana 14")

c.create_text(200, 200, text="About this",
                        anchor=SE,
                        fill="grey")

root.mainloop()

# ================================================================
```

■ Обращение к созданным фигурам для изменения их свойств

Ранее было описано размещение геометрических примитивов на экземпляре Canvas, далее будут рассмотрены способы обращения к уже созданным фигурам для изменения их свойств, а также будет создана анимация.

В Tkinter существует два способа "пометить" фигуры, размещенные на холсте, – это идентификаторы и теги.

Идентификаторы всегда уникальны для каждого объекта. Два объекта не могут иметь одни и тот же идентификатор.

Теги НЕ уникальны. Группа объектов на холсте может иметь один и тот же тег. Это дает возможность менять свойства всей группы объектов.

Отдельно взятая фигура на Canvas может иметь как идентификатор, так и тег.

■ Идентификаторы

Методы, создающие фигуры на канве (на холсте), возвращают численные значения - идентификаторы этих объектов, через которые можно обращаться к созданным фигурам и которые можно присвоить переменным.

```
# ================================================================
from tkinter import *

root = Tk()
```

```
cnv = Canvas(width=300, height=300, bg='white')
cnv.focus_set()
cnv.pack()

# Зелёный круг
ball = cnv.create_oval(140, 140, 160, 160, fill='green')

# он двигается по холсту (канве) с помощью стрелок на клавиатуре.
# При создании круга методом create_oval от имени объекта Canvas,
# идентификатор этого круга присваивается переменной ball.



# функции обработки событий '<Up>','<Down>','<Left>','<Right>'
                                                    # объекта cnv (Canvas)
cnv.bind('<Up>',    lambda event, b = ball: cnv.move(b, 0, -2))
                                     # смещение по канве объекта ball вниз
cnv.bind('<Down>', lambda event, b = ball: cnv.move(b, 0, 2))
                                     # смещение по канве объекта ball вверх
cnv.bind('<Left>', lambda event, b = ball: cnv.move(b,-2, 0))
                                     # смещение по канве объекта ball влево
cnv.bind('<Right>',lambda event, b = ball: cnv.move(b, 2, 0))
                                     # смещение по канве объекта ball вправо
#          событие
#                   lambda функция обработки события:
#    Метод move объекта cnv Canvas в качестве аргумента
#    принимает переменную b (в lambda ей присваивается значение ball)
#    и смещение по осям.
#    Обязательный позиционный аргумент event (без него НИКАК)
#    в lambda функции обработки события не используется

root.mainloop()

# ========================================================================
```

■ Теги

Тег является свойством объекта и присваивается объекту при его создании. Дла этого атрибуту tag присваивается строка с именем тега. В отличие от идентификаторов, которые являются уникальными для каждого объекта, один и тот же тег может присваиваться разным объектам. Дальнейшее обращение к такому тегу позволяет изменять свойства всех объектов, которые были помечены этим тегом при их создании.

Далее в примере эллипс и линия помечены одним и тем же тегом, а функция setcolor изменяет цвет всех объектов с тегом group1.

В отличие от имени идентификатора, которое является переменной, имя тега представляет собой строковое значение и заключается в кавычки.

```
"""
 В отличие от идентификаторов, которые являются уникальными для каждого
 объекта, один и тот же тег может присваиваться разным объектам.
 Дальнейшее обращение к такому тегу позволяет изменять свойства
 всех объектов, которые были помечены этим тегом.
 Здесь линия и эллипс помечены одним и тем же тегом, а функция setcolor
 изменяет цвет всех объектов с тегом group1.
 В отличие от имени идентификатора (переменная), имя тега заключается
```

```
 в кавычки (строковое значение).
"""

from tkinter import *

# функция обработки события ==========================================
def setcolor(event):
    c.itemconfig('group1', width=3, fill="red")
    # для всех объектов по тегу 'group1' на данной канве c...
# ====================================================================

root = Tk()

c = Canvas(width=460, height=150, bg='white')
c.pack()

# объекты под одним тегом "group1" =====================
oval = c.create_oval(30, 10, 130, 80, tag="group1")
c.create_line(10, 100, 450, 100,      tag="group1")
# ======================================================

# ====================================================================
c.bind('<Button-3>', setcolor)
#   щелчок правой мышиной кнопкой по канве - функция обработки события

root.mainloop()
# ====================================================================
```

■ Привязка события

Метод tag_bind позволяет привязать событие (например, щелчок кнопкой мыши) к определенной фигуре на Canvas. Таким образом, можно реализовать обращение к различным областям холста с помощью одного и того же события.

Далее пример иллюстрирует, как изменения на холсте зависят от того, где произведен клик. Метод delete удаляет объект. Если нужно очистить холст, то вместо идентификаторов или тегов используется константа ALL.

```
"""
 Метод tag_bind позволяет привязать событие (например, щелчок кнопкой
 мыши) к определенной фигуре на Canvas. Таким образом, с помощью одного и
 того же события, можно реализовать обращение к различным областям
 холста. Далее пример иллюстрирует, как изменения на холсте зависят от
 того, где произведен клик.
 Метод delete удаляет объект. Если нужно очистить холст, вместо
 идентификаторов или тегов используется константа ALL.
"""


from tkinter import *

# ====================================================================
# функции обработки событий (щелчки по разным областям канвы)

def blue_oval_func(event):
    c.delete(ALL)
    c.create_text(100, 50, text="Очистить канву")

def oval_func(event):
    c.delete(oval)
    c.create_text(80, 50, text="Круг")

def rect_func(event):
    c.delete("rect")
```

```
    c.create_text(230, 50, text="Прямоугольник")

def triangle_func(event):
    c.delete(trian)
    c.create_text(380, 50, text="Треугольник")
# ================================================================

# создание главного окна приложения
# root = Tk()  # вариант_1

c = Canvas(width=460, height=100, bg='grey80')
c.pack()

# ================================================================================
oval = c.create_oval(30, 10, 130, 80,fill="orange")

c.create_rectangle(180, 10, 280, 80, fill="lightgreen",
                                     tag="rect")

trian = c.create_polygon(330, 80, 380, 10, 430, 80, fill='white',
                                                    outline="black")

c.create_oval(5, 5, 25, 25, fill="blue",
                            tag="blue_oval")
# ================================================================================

# привязка по ссылке на объект (идентификатор oval)
c.tag_bind(oval, '<Button-1>', oval_func)
# привязка по тегу (tag="rect")
c.tag_bind("rect", '<Button-1>', rect_func)
# привязка по ссылке на объект (идентификатор trian)
c.tag_bind(trian, '<Button-1>', triangle_func)

# привязка по тегу (tag="blue_oval") 'Гасить на холсте ВСЁ!'
c.tag_bind("blue_oval", '<Button-1>', blue_oval_func)

# цикл обработки событий запускался от имени главного окна приложения
# (вариант_1).
# Этот объект надо было явно создавать, а затем от его имени вызывать
# метод .mainloop
# Это приводило к 'неявному' вызову функции mainloop() и запуску цикла.
# root.mainloop()  # вариант_1

# В качестве альтернативы (вариант_2) возможен непосредственный вызов
# функции mainloop() и запуск цикла обработки событий. Главное окно
# приложения явным образом при этом НЕ создаётся.
mainloop()    # вариант_2

# ================================================================================
```

■ Настройки окон

Обычные окна, в которых располагаются виджеты, в Tkinter порождаются не только от класса Tk, но и от Toplevel. От Tk принято создавать главное окно. Если создается многооконное приложение, то остальные окна создаются от Toplevel. Методы обоих классов схожи.

Размер и положение окна: по умолчанию окно приложения появляется в верхнем левом углу экрана. Его размер (ширина и высота) определяется совокупностью размеров расположенных в нем виджетов. В случае если окно пустое, то tkinter устанавливает по умолчанию его размер в 200 на 200 пикселей. С помощью метода geometry можно изменить как размер окна, так и его положение. Метод принимает строку определенного формата.

```
# ================================================================================
from tkinter import *
```

```
root = Tk()

root.geometry('600x400+200+100')

root.mainloop()
# ============================================================================
```

Первые два числа в строке-аргументе geometry задают ширину и высоту окна. Вторая пара чисел обозначает смещение на экране по осям x и y.

В примере окно размером 600 на 400 будет смещено от верхней левой точки экрана на 200 пикселей вправо и на 100 пикселей вниз.

Если перед обоими смещениями вместо плюса указывается минус, то расчет происходит от нижних правых углов экрана и окна.

При выполнении выражения root.geometry('600x400-0-0') окно появляется в нижнем правом углу.

В аргументе метода geometry можно не указывать либо размер, либо смещение.

Например, чтобы сместить окно, но при этом не менять его размер, достаточно написать root.geometry('+200+100').

Бывает удобно, чтобы окно приложения размещалось в центре экрана. Далее показан способ размещения окна с определёнными размерами в заданной точке относительно центра экрана. Методы winfo_screenwidth и winfo_screenheight возвращают размеры (в пикселях по ширине и высоте) фрагмента экрана, занимаемого окном приложения:

```
w = root.winfo_screenwidth()
h = root.winfo_screenheight()
w = w//2 # середина экрана
h = h//2
w = w - 200 # смещение от середины
h = h - 200
root.geometry('400x400+{}+{}'.format(w, h))
```

Здесь из определённых значений, равных половине ширины и высоты фрагмента экрана вычитается по 200 пикселей. Иначе в центре экрана оказывается верхний левый угол окна, а не его середина.

Если размер окна неизвестен, его можно получить с помощью того же метода geometry, но без аргументов. В этом случае метод возвращает строку, содержащую сведения о размерах и смещении, из которой можно определить ширину и высоту окна приложения.

```
# ============================================================================
# Если размер окна неизвестен, то его можно получить с помощью того же
# метода geometry, но без аргументов. В этом случае метод возвращает
# строку, содержащую сведения о размерах и смещении, из которой можно
# извлечь ширину и высоту окна.

from tkinter import *
# в этом приложении очень многое завязано на главное окно приложения
# и поэтому без явного упоминания объекта root никак не обойтись!

root = Tk()   # создание окна

# Кнопка - Метка - Кнопка ==================================
Button(text="Button", width=20).pack()
Label(text="Label", width=20, height=3).pack()
Button(text="Button", width=20).pack()
# ==========================================================
```

```
# update_idletasks позволяет перезагрузить данные об окне после размещения
# на нем виджетов. Иначе geometry вернет строку, где ширина и высота
# равняются по одному пикселю.

root.update_idletasks()
s = root.geometry()

# разбор строки с инфой об окне
s = s.split('+')           # ширина x высота + смещение на экране по X +
                           # смещение на экране по Y
s = s[0].split('x')        # s[0](ширина) s[1](высота)
width_root = int(s[0])     # ширина окна
height_root = int(s[1])    # высота окна

#  количество пикселей экрана, на котором появляется окно.
w = root.winfo_screenwidth()
h = root.winfo_screenheight()

#  половина количества пикселей экрана, на котором появляется окно.
w = w // 2
h = h // 2

# новые значения w и h
w = w - width_root // 2
h = h - height_root // 2

#  чтобы сместить окно, но не менять его размер, надо сформировать строку
#  - str_pos и подать её как аргумент в метод geometry...
str_pos = '+{0}+{1}'.format(w, h)
root.geometry(str_pos)
# либо сделать то же самое в одном операторе
# root.geometry('+{0}+{1}'.format(w, h))

root.mainloop()
# ============================================================================
```

■ *Заголовок окна*

По умолчанию окно может разворачиваться на весь экран, а также изменять его размер, раздвигая границы. Эти возможности можно отключить с помощью метода resizable.

Так root.resizable(False, False) запретит изменение размеров главного окна как по горизонтали, так и вертикали. Развернуть на весь экран его также будет невозможно, при этом соответствующая кнопка разворота исчезает.

По умолчанию в строке заголовка окна находится надпись "tk". Для установки собственного названия используется метод title.

```
root.title("Главное окно")
```

Заголовок окна можно вообще убрать. В приложении ниже второе окно (Toplevel) открывается при клике на кнопку, оно не имеет заголовка, так как к нему был применен метод overrideredirect с аргументом True. Через несколько секунд это окно закрывается с помощью метода destroy.

```
# ============================================================================
from tkinter import *

# функция обработки события ==================================================
# по нажатии на кнопку 'about' будет создано второе окно и зарегистрирована
# анонимная lambda функция (обработчик события 5000 тиков), которая это
# самое окно закроет через назначенные 5000 тиков ========================
```

```
def about():
    a = Toplevel()                          # второе окно приложения
    a.geometry('200x150')
    a['bg'] = 'grey'
    a.overrideredirect(True)                # без заголовка, но с меткой
    Label(a, text="About this").pack(expand=1)
    # Метод after() позволяет регистрировать lambda функцию
    # (функцию обратного вызова),
    # которая вызывается после задержки, заданной в миллисекундах
    # в основном цикле Tkinter и закрывает это самое второе окно.
    a.after(5000, lambda: a.destroy())
    #       событие (5000 тиков)
    #            анонимная функция - обработчик события (закрывает окно)
# ============================================================================

root = Tk()
root.title("Главное окно")

# виджеты ====================================================================
Button(text="Button", width=20).pack()
Label(text="Label", width=20, height=3).pack()
Button(text="About", width=20, command=about).pack() # кнопка с
                                                     # обработкой события

# метод after в действии =====================================================
# Этот код выведет              «-----1-----»,
#                               «-----2-----» и,
# наконец, через некоторое время «-----3-----» .
# Все это время графический интерфейс будет оставаться доступным,
# и пользователи смогут взаимодействовать с приложением. =================
print('-----1-----')
root.after(3000, lambda: print('-----3-----'))
print('-----2-----')
# ============================================================================

# цикл обработки событий
root.mainloop()

# ============================================================================
```

■ Менеджер геометрии grid

Менеджеры геометрии в Tkinter: ранее рассмотренный Pack, Place, Grid. "Grid" - это "сетка" или таблица. У всех виджетов есть соответствующий данному менеджеру метод, который обеспечивает табличный способ размещения виджетов.

При разработке сложных интерфейсов табличный способ размещения виджетов предпочтителен из-за его гибкости и удобства (?). Он позволяет избежать использования множества фреймов, что неизбежно в случае упаковщика Pack.

■ Правила применения метода grid

При размещении виджетов методом grid родительский контейнер (обычно это окно) условно разделяется на ячейки подобно таблице. Адрес каждой ячейки состоит из номера строки и номера столбца. Нумерация начинается с нуля. Ячейки можно объединять как по вертикали, так и по горизонтали. При объединении ячеек общая ячейка обозначается адресом первой.

Никаких предварительных команд по разбиению родительского виджета на ячейки не выполняется. Tkinter делает это сам (!), исходя из указанных позиций виджетов. Размещение виджета в той или иной ячейке задается через аргументы row (номер строки) и column (номер столбца).

Для объединения ячеек по горизонтали используется атрибут columnspan, которому присваивается количество объединяемых ячеек. Для объединения ячеек по вертикали используется атрибут rowspan.

Применение менеджера геометрии Grid. Общий подход:

- общий дизайн с перечнем виджетов и предполагаемой их локализацией,
- представление схемы в виде таблицы с (возможно объединёнными ячейками),
- нумерация ячеек (возможно объединённых), в которых предполагается размещать виджеты,
- код.

```
# ================================================================================
from tkinter import *

root = Tk()

Label(text="Имя:").grid(row=0, column=0)
Entry(width=30).grid(row=0, column=1, columnspan=3)

Label(text="Столбцов:").grid(row=1, column=0)
Spinbox(width=7, from_=1, to=50).grid(row=1, column=1)
Label(text="Строк:").grid(row=1, column=2)
Spinbox(width=7, from_=1, to=100).grid(row=1, column=3)

Button(text="Справка").grid(row=2, column=0)
Button(text="Вставить").grid(row=2, column=2)
Button(text="Отменить").grid(row=2, column=3)

root.mainloop()
# ================================================================================
```

■ Другие аргументы метода grid

У метода grid (как и у метода pack) имеются атрибуты для задания внешних и внутренних отступов (padx, pady, ipadx, ipady).

Имеется атрибут sticky (липкий), который принимает значения направлений сторон света (N, S, W, E, NW, NE, SW, SE). Если, например, указать NW, то виджет прилипнет к верхнему левому углу ячейки. Виджеты можно растягивать на весь объем ячейки (sticky=N+S+W+E) или только по одной из осей (N+S или W+E). Эффект от аргумента sticky (липкий) заметен, только если виджет меньше ячейки.

Далее в примере используются виджеты класса Spinbox. Spinbox похож на Entry, но для него задается список принимаемых значений и скроллер для перебора значений из списка.

```
from tkinter import *

# вариант_1 создание главного окна приложения
#root = Tk()

Label(text="Имя:").grid(row=0, column=0, sticky=W, pady=10, padx=10)
table_name = Entry()
table_name.grid(row=0, column=1, columnspan=3, sticky=W + E, padx=10)

Label(text="Столбцов:").grid(row=1, column=0, sticky=W, padx=10, pady=10)
Spinbox(width=7, from_=1, to=50).grid(row=1, column=1, padx=10)
Label(text="Строк:").grid(row=1, column=2, sticky=E)
Spinbox(width=7, from_=1, to=100).grid(row=1, column=3, sticky=E, padx=10)
```

```
Button(text="Справка").grid(row=2, column=0, pady=10, padx=10)
Button(text="Вставить").grid(row=2, column=2)
Button(text="Отменить").grid(row=2, column=3, padx=10)

# вариант_1 запуск цикла обработки событий от имени root
#root.mainloop()

# вариант_2 непосредственный вызов функции mainloop
mainloop()
```

С помощью методов grid_remove и grid_forget можно сделать виджет невидимым. Отличие между этими методами в том, что grid_remove запоминает прежнее положение виджета. Для его отображения в прежней ячейке достаточно применить grid без аргументов. После grid_forget нужно заново конфигурировать положение виджета.

```
# ========================================================================
# С помощью методов grid_remove и grid_forget можно сделать виджет
# невидимым. Отличие между методами:
# grid_remove запоминает прежнее положение виджета.
#             Для его отображения в прежней ячейке достаточно применить
#             grid без аргументов.
# grid_forget забывает прежнее положение виджета.
#             Для отображения нужно виджета нужно заново конфигурировать
#             его положение.
# ========================================================================

from tkinter import *

# функции - обработчики событий

def rem():
    global l1_flag
    if l1_flag == 1:
        l1.grid_remove()
        l1_flag = 0
    else:
        l1.grid()
        l1_flag = 1

def forg():
    global l2_flag
    if l2_flag == 1:
        l2.grid_forget()
 l2_flag = 0
    else:
        l2.grid(row=1)
        l2_flag = 1

# ========================================================================

# root = Tk() # вариант_1

l1_flag = 1
l2_flag = 1

l1 = Label(width=5, height=3, bg='blue')
l2 = Label(width=5, height=3, bg='green')

# кнопки с назначенными обработчиками событий
b1 = Button(bg='lightblue', command=rem)
b2 = Button(bg='lightgreen', command=forg)
```

```
# применение менеджера геометрии grid
l1.grid(row=0)
l2.grid(row=1)
b1.grid(row=2)
b2.grid(row=3)

# root.mainloop() # вариант_1
mainloop() # вариант_2 непосредственный вызов
           # функции запуска цикла обработки
# ========================================================================
```

В приложении голубая метка после удаления будет появляться в том же месте, где была до этого, несмотря на то, что в функции rem к ней применяется метод grid без настроек.

В отличие от нее зеленая метка, если ей не указать место размещения после удаления, появлялась бы под кнопками. Скрытие виджетов бывает необходимо в тех случаях, когда, например, от выбора пользователя в одной части интерфейса зависит, какие виджеты появятся в другой.

■ Метод grid: применение

Метод позволяет поместить виджет в конкретную ячейку условной сетки либо грида. При этом используются параметры:

- column — это номер столбца, отсчитывается с нуля;
- row — это номер строки, отсчитывается с нуля;
- columnspan — указывает число столбцов, занимаемых элементом;
- rowspan — указывает число строк;
- ipadx и ipady — подразумеваются отступы по горизонтали и вертикали от границ компонента до текста компонента;
- padx и pady — аналогичные отступы, но от границ ячейки грида до границ компонента;
- sticky — определяет выравнивание элемента в ячейке в случае, когда ячейка больше компонента.

Для начала работы с Tkinter надо импортировать библиотеку.

Далее – грид из 9 кнопок:

```
# ========================================================================
from tkinter import *
# Далее - грид из 9 кнопок:
root = Tk()
root.title('Gui on python')
root.geometry('300x250')
for x in range(3):
    for y in range(3):
        btn = Button(text="{0}-{1}".format(x, y))
        btn.grid(row = x,
                 column = y,
                 ipadx = 10,
                 ipady = 5,
                 padx = 10,
                 pady = 10)

root.mainloop()

# ========================================================================
```

■ Диалоговые окна

Пакет tkinter содержит несколько модулей, предоставляющих доступ к уже готовым диалоговым окнам. Это окна различных сообщений, выбора по принципу "да-нет", открытия и сохранения файлов и др.

Далее будут рассмотрены примеры окон из модулей messagebox и filedialog пакета tkinter.

Модули пакета нужно импортировать отдельно. То есть импортируется содержимое tkinter (например, from tkinter import *) и отдельно входящий в состав пакета tkinter модуль.

■ Импорт модуля и вызов функций модуля

Способы импорта на примере модуля messagebox и пример вызова одной из функций модуля в зависимости от способа импорта:

import tkinter.messagebox	tkinter.messagebox.askyesno()
from tkinter.messagebox import *	askyesno()
from tkinter import messagebox	messagebox.askyesno()
from tkinter import messagebox as mb (вместо mb может быть любой идентификатор)	mb.askyesno()

Далее будет использован последний вариант.

```
# ======================================================================
# Модуль messagebox - стандартные диалоговые окна
# Окно выбора "да" или "нет" - askyesno:
# ======================================================================

from tkinter import *
# импорт модуля с диалоговыми окнами
from tkinter import messagebox as mb

# обработчик нажимания кнопки ==========================================
def check():
    # диалоговое окно ДА-НЕТ
    answer = mb.askyesno(
        title="Вопрос",
        message="Перенести данные?")

    # если был получен положительный ответ (нажата кнопка ДА):
    if answer:
        s = entry.get()         # в окошке entry прочитать введённый текст
        entry.delete(0, END)    # очистить виджет Entry
        label['text'] = s       # прочитанный в Entry поместить в метке
# ======================================================================

root = Tk()             # главное окно приложения

entry = Entry()         # виджет Entry
entry.pack(pady=10)     # размещение виджета

# кнопка - вопрос о передаче: свойство command связано
# с вызовом функции check.
Button(text='Передать', command=check).pack()

label = Label(height=3)   # метка для размещения передаваемого текста
```

```
label.pack()

root.mainloop()
# ================================================================
```

"Да" в диалоговом окне возвращает в программу True,

"Нет" вернет False (также как закрытие окна через крестик).

Таким образом, в коде можно обработать выбор пользователя. В данном случае если пользователь соглашается, то данные переносятся из поля в метку.

Опции title и message являются позиционными, так что можно указывать только значения. Подобные окна генерируются при использовании функции askokcancel с надписями на кнопках "ОК" и "Отмена":

- askyesno("Вопрос", "Перенести данные?"),
- askquestion (возвращает не True или False, а строки 'yes' или 'no'),
- askretrycancel ("Повторить", "Отмена"),
- askyesnocancel ("Да", "Нет", "Отмена").

Далее окна с одной кнопкой, которые служат для вывода сообщений различного характера.

Это showerror, showinfo и showwarning.

```
…
def check():
    s = entry.get()
    if not s.isdigit():
        mb.showerror(
            "Ошибка",
            "Должно быть введено число")
    else:
        entry.delete(0, END)
        label['text'] = s
…
```

Следующее приложение основано на применении функций из модуля filedialog – askopenfilename и asksaveasfilename. Первая предоставляет диалоговое окно для открытия файла, вторая – для сохранения. Обе возвращают имя файла, который должен быть открыт или сохранен, но сами они его в диалоговых окнах для открытия и сохранения файлов НЕ открывают и НЕ сохраняют.

Это делается программными средствами самого python.

```
# ================================================================

from tkinter import *

# импорт модуля с диалоговыми окнами открытия и сохранения файлов
from tkinter import filedialog as fd


# функция обработчик сохранения текста ================================
def insert_text():
    file_name = fd.askopenfilename()
    f = open(file_name)
    s = f.read()
    text.insert(1.0, s)
    f.close()
# ================================================================
```

```
# функция обработчик прочтения текста из файла ============================
def extract_text():
    file_name = fd.asksaveasfilename(
        # Опция filetypes позволяет перечислить типы файлов, которые будут
        # сохраняться или открываться, и их расширения.
        filetypes=(("TXT files", "*.txt"),
                   ("HTML files", "*.html;*.htm"),
                   ("All files", "*.*")))
    f = open(file_name, 'w')
    s = text.get(1.0, END)
    f.write(s)
    f.close()
# =========================================================================

root = Tk()     # главное окно приложения

# окошко для ввода текста ================================================
text = Text(width=50, height=25)
# при размещении текстового поля методом grid не указаны аргументы
# row и column. В таких случаях по умолчанию предполагается, что их
# значения установлены по нулям.
text.grid(columnspan=2)

# понизу (row=1) кнопки 'Открыть' и 'Сохранить' ==========================
b1 = Button(text="Открыть", command=insert_text)
b1.grid(row=1, sticky=E)

b2 = Button(text="Сохранить", command=extract_text)
b2.grid(row=1, column=1, sticky=W)
# =========================================================================

root.mainloop()

# =========================================================================
```

■ Виджет Menu

Меню – это виджет, который присутствует во многих пользовательских приложениях. Это выпадающее окно, содержащее различные команды. Оно располагается под строкой заголовка и представляет собой выпадающие списки под словами-пунктами меню. Пункты конечных списков представляют собой команды, открывающие диалоговые окна, либо обеспечивающие выполнение в приложении каких-либо действий.

В tkinter экземпляр меню создается от класса Menu, далее его надо привязать к виджету, на котором оно будет расположено. Обычно это главное окно приложения. Опции menu главного окна приложения присваивается экземпляр (объект) Menu через имя связанной с объектом переменной.

```
from tkinter import *

root = Tk()
mainmenu = Menu(root)
root.config(menu=mainmenu)

root.mainloop()
```

Выполнение этого кода никакого меню не покажет. Так как ни одного пункта меню не было создано. Предполагается, что "Файл" и "Справка" – это метки для обозначения команд. Метод add_command добавляет пункт меню:

```
from tkinter import *

root = Tk()
```

```
mainmenu = Menu(root)
root.config(menu=mainmenu)

mainmenu.add_command(label='Файл')
mainmenu.add_command(label='Справка')

root.mainloop()
```

Обычно панель меню содержит выпадающие списки меток команд, а сами по себе метки (пункты на панели) командами не являются. Событие смены значения в выпадающем списке – '<<ComboboxSelected>>'. Клик по выпадающим спискам меток команд приводит лишь к раскрытию соответствующего списка. К ним можно добавить опцию command и связать их тем самым с какой-либо функцией-обработчиком выбора меню (клика). Можно создавать новые экземпляры Menu и подвязывать их к главному меню с помощью метода add_cascade.

```
# Меню - это выпадающее окно, содержащее различные команды.
# пример меню с одним элементом.
# При нажатии на «Выход» из меню, приложение закрывается. ================

from tkinter import Tk, Frame, Menu

# ============================================================================
class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    # создание экземпляра меню ==============================================
    def initUI(self):
        # master - корневое окно приложения по умолчанию
        self.master.title("Простое меню")

        # создаётся панель меню. Для этого применяется виджет Menu,
        # который настраивается для отображения в качестве меню для
        # корневого окна.
        menubar = Menu(self.master)
        self.master.config(menu=menubar)

        # создаётся объект меню для секции «Файл».
        fileMenu = Menu(menubar)
        # добавляется новый элемент (команда) в меню «Файл».
        # Эта команда будет вызывать метод onExit().
        fileMenu.add_command(label="Выход", command=self.onExit)
        # Меню «Файл» добавляется на панель меню
        # при помощи метода add_cascade().
        menubar.add_cascade(label="Файл", menu=fileMenu)

    # обработчик события 'выбор пункта меню Выход' ======================
    def onExit(self):
        self.quit()
# ============================================================================

def main():
    root = Tk()
    root.geometry('300x200+400+400')
    app = Example()
    root.mainloop()

if __name__ == '__main__':
    main()
```

■ Добавление подменю для основного меню

В следующем примере будет продемонстрировано добавление подменю в Tkinter. Подменю – это меню, встроенные в другие элементы меню. Это обычное меню, которое иерархически входит в другое меню. Далее представлено подменю с тремя командами.

```
# ================================================================
from tkinter import Tk, Frame, Menu

class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    # создание экземпляра меню ===============================================
    def initUI(self):
        # master - корневое окно приложения по умолчанию
        self.master.title("Добавление подменю")

        # создаётся панель меню. Для этого применяется виджет Menu,
        # который настраивается для отображения в качестве меню для
        # корневого окна.
        menubar = Menu(self.master)
        self.master.config(menu=menubar)

        # создаётся объект меню для секции «Файл».
        fileMenu = Menu(menubar)

        # далее в подменю от основного меню «Файл» реализовны три опции.
        # также создаётся разделитель и горячие клавиши.
        submenu = Menu(fileMenu)
        submenu.add_command(label="Новый источник")
        submenu.add_command(label="Закладки")
        submenu.add_command(label="Почта")

        # Меню «Импортировать» добавляется на панель меню
        # при помощи метода add_cascade().
        fileMenu.add_cascade(label='Импортировать',
                             menu=submenu,
                             underline=0)
        # Разделитель - это горизонтальная линия, которая визуально
        # разделяет элементы меню. Благодаря разделителям можно
        # группировать элементы в меню.
        fileMenu.add_separator()

        fileMenu.add_command(label="Выход",
                        underline=0,
                             command=self.onExit)
        menubar.add_cascade(label="Файл", underline=0, menu=fileMenu)

    # обработчик события 'выбор пункта меню Выход' ========================
    def onExit(self):
        self.quit()
    # ================================================================

def main():
    root = Tk()
    root.geometry("300x200+400+400")
    app = Example()
    root.mainloop()

if __name__ == '__main__':
    main()
# ================================================================
```

Подменю прикрепляется к основному меню «Файл», но не к самой панели меню. Так и создается подменю. При помощи параметра underline можно установить горячие клавиши. Этот параметр подчеркивает символ, обозначающий горячую клавишу команды. Позиции символов начинаются с нуля. При нажатии на меню «Файл», появляется контекстное окно. В меню «Импортировать» является подчеркнутым также только один символ. Этот пункт меню можно выбрать при помощи курсора или сочетанием горячих клавиш Alt+I.

■ Всплывающее меню в Tkinter

В следующем примере создаётся всплывающее меню. Его можно вызвать в любой части рабочей области окна.

```
# ==================================================================
# всплывающее меню с двумя командами
from tkinter import Tk, Frame, Menu


class Example(Frame):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title("Всплывающее меню")
        # Для создания всплывающего меню используется обычный виджет Menu.
        # У него отключается свойство tearoff. И теперь есть возможность
        # показать меню в новом всплывающем окне.
        self.menu = Menu(self.master, tearoff=0)

        self.menu.add_command(label="Сигнал!", command=self.onBell)
        self.menu.add_command(label="Выход", command=self.onExit)

        # методу showMenu() назначатся событие
        # 'клик правой кнопкой мыши <Button-3>' .
        # Событие срабатывает тогда, когда по рабочей области окна
        # нажимается правая мышиная кнопка.
        self.master.bind("<Button-3>", self.showMenu)
        self.pack()

    # Метод showMenu() отображает всплывающее меню. Это меню открывается
    # по x и y координатам от расположения мышиного курсора
    def showMenu(self, e):
        self.menu.post(e.x_root, e.y_root)

    # метод обработки выбора пункта меню Выход
    def onExit(self):
        print('=quit=')
        self.quit()

    # метод отработки выбора путкта меню Сигнал!
    def onBell(self):
        print('=bell=')
        self.bell()
        self.bell()
        self.bell()
        self.bell()


def main():
    root = Tk()
    root.geometry("300x200+400+400")
    app = Example()
```

```
    root.mainloop()

if __name__ == '__main__':
    main()
# ================================================================================
```

■ Панель инструментов в Tkinter

Меню объединяют команды, которые можно использовать в приложении. Панель инструментов позволяет получить быстрый доступ к наиболее популярным командам. В Tkinter нет виджета панели инструментов, его надо создать. Представленный ниже код использует иконку которую нужно будет сохранить рядом с файлом code_56.py: exit.png

```
# ================================================================================
from PIL import Image, ImageTk
from tkinter import Tk, Frame, Menu, Button
from tkinter import LEFT, TOP, X, FLAT, RAISED


class Example(Frame):

    # метод обработки события применение инструмента 'Выход' =============
    def onExit(self):
        self.quit()

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.master.title("Панель инструментов")

        menubar = Menu(self.master)
        self.fileMenu = Menu(self.master, tearoff=0)

        # добавление пункта меню с назначением метода обработки события
        self.fileMenu.add_command(label="Выход", command=self.onExit)
        # fileMenu подвязывается к главому меню menubar
        menubar.add_cascade(label="Файл", menu=self.fileMenu)

        # панелью инструментов будет рамка, в которой помещается кнопка
        toolbar = Frame(self.master, bd=1, relief=RAISED)

        # создана панель инструментов. Это обычная рамка.
        # Созданы границы панели инструментов.
        self.img = Image.open("exit.png")
        eimg = ImageTk.PhotoImage(self.img)

        #  создано изображение и фотоизображение для панели инструментов
        # виджет кнопки выхода
        exitButton = Button(
                             toolbar,
                             image=eimg,
                             relief=FLAT,
                             command=self.quit
                            )

        exitButton.image = eimg
        exitButton.pack(side=LEFT, padx=2, pady=2)
        # Панель инструментов - это рамка, а рамка - это виджет контейнера.
        # Кнопка закрепляется у левого края. Далее добавляется
        # небольшой отступ в 2 пикселя
```

```
        # Сама панель инструментов закреплена к верхней части
        # главного окна и растянута горизонтально.
        toolbar.pack(side=TOP, fill=X)
        self.master.config(menu=menubar)
        self.pack()

    # ================================================================


def main():
    root = Tk()
    root.geometry("250x150+300+300")
    app = Example()
    root.mainloop()


if __name__ == '__main__':
    main()
# ================================================================================
```

■ Научная графика в Python

Библиотека matplotlib - это библиотека двумерной графики для языка программирования python с помощью которой можно создавать рисунки различных форматов.После установки библиотеки matplotlib вызывается в консоли или в скрипте как модуль:

```
# ================================================================================
import matplotlib as mpl
# Вывод текущей версии библиотеки matplotlib
print('Current version on matplotlib library is', mpl.__version__)
# ================================================================================
```

Matplotlib представляет собой модуль-пакет для python. Matplotlib состоит из множества модулей. В модулях множество классов и функций, которые иерархически связаны между собой.

Создание рисунка в matplotlib: нужно иметь представление о будущем рисунке (что именно он будет рисовать), взять основу (холст или бумагу), инструменты (кисти или карандаши), нарисовать рисунок (деталь за деталью).

В matplotlib все эти этапы также существуют, и в качестве художника-исполнителя здесь выступает сама библиотека. Пользователь должен управлять модулями matplotlib, и для этого он определяет что именно нужно нарисовать и какими инструментами.

Обычно matplotlib является основным средством рисования и обеспечивает создание основы и процесс отображения рисунка. Отображение графиков обеспечивается функциями из пакета pyplot библиотеки matplotlib, обозначенного как plt. При импорте пакета pyplot создается готовый к работе объект plt (???) со всеми его графическими возможностями. Нужно всего лишь использовать функцию plot() и передать ему массив (возможно кортеж) значений, по которым строится график.

В результате будет создан объект Line2D. Это линия, которая представляет собой последовательность точек, нанесённую на график. С помощью функции plt.show() остаётся показать этот график.

■ Структура matplotlib

Библиотека matplotlib организована иерархически. Её описание проще всего начинать с функций высокого уровня, с высокоуровневого интерфейса matplotlib.pyplot.

Для рисования гистограммы средствами matplotlib.pyplot достаточно вызывать всего один метод:

matplotlib.pyplot.hist(arr)

И не важно, как именно была нарисована эта диаграмма.

Главное - это общая структура рисунка: рисунок иерархичен.

Итоговая картинка (диаграмма высокого уровня) строится из простых геометрических фигур (линий - средний уровень рисунка), которые создаются несколькими универсальными методами рисования (низкий уровень).

■ Применение модуля matplotlib.pyplot

```
# ======================================================================

import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
plt.show()

# ======================================================================
```

■ Рисунок (Figure)

Рисунок является объектом самого верхнего уровня, на котором располагаются основа-холст (Canvas), одна или несколько областей рисования (Axes), элементы рисунка Artisits (заголовки, легенда и т.д.). На рисунке может быть несколько областей рисования Axes, но данная область рисования Axes может принадлежать только одному рисунку Figure.

■ Иерархическая структура рисунка в matplotlib

Главной единицей (объектом самого высокого уровня) при работе с matplotlib является рисунок (Figure). Любой рисунок в matplotlib имеет вложенную структуру. Работа пользователя предполагает операции с разными уровнями рисунка:

Figure(Рисунок) -> Canvas(Холст) -> Axes(Область рисования) -> Axis(Координатная ось)

■ Область рисования (Axes)

Область рисования является объектом среднего уровня. Это главный объект работы с графикой matplotlib в объектно-ориентированом стиле. Это часть изображения с пространством данных. Каждая область рисования Axes содержит две (или три в случае трёхмерных данных) координатных оси (Axis объектов), которые упорядочивают отображение данных.

■ Координатная ось (Axis)

Координатная ось - объект среднего уровня. Определяет область изменения данных, на них наносятся деления ticks и подписи к делениям ticklabels. Расположение делений определяется объектом Locator, а подписи делений обрабатывает объект Formatter. Конфигурация

координатных осей заключается в комбинировании различных свойств объектов Locator и Formatter.

Элементы рисунка (Artists)

Элементы рисунка Artists составляют сам рисунок и на рисунке нет ничего, что бы не было элементом рисунка на всех его уровнях. Практически всё, что отображается на рисунке является элементом рисунка (Artist), даже объекты Figure, Axes и Axis. Также элементы рисунка Artists включают в себя такие (простые) объекты как текст (Text), плоская линия (Line2D), фигура (Patch) и другие.

Когда происходит отображение рисунка (figure rendering), все элементы рисунка Artists наносятся на основу-холст (Canvas). Большая часть из них связывается с областью рисования Axes. Элемент рисунка не может совместно использоваться несколькими областями Axes или быть перемещён с одной на другую.

Интерфейс прикладного программирования matplotlib API

В matplotlib изобразительные функции логически разделены между несколькими объектами, причём каждый из них сам имеет довольно сложную структуру. Можно выделить три уровня интерфейса прикладного программирования(matplotlib API):

- matplotlib.backend_bases.FigureCanvas - абстрактный базовый класс, который позволяетрисовать и визуализировать результаты команд;
- matplotlib.backend_bases.Renderer - объект (абстрактный класс), который знает как рисовать на FigureCanvas;
- matplotlib.artist.Artist - объект, который знает, как использовать визуализатор (renderer),чтобы рисовать на холсте (canvas).

FigureCanvas и Renderer обрабатывают детали, необходимые для взаимодействия со средствами пользовательского интерфейса, а Artist обрабатывает все конструкции высокого уровня такие как представление и расположение рисунка, текста и линий.

Классы Artists

Существует два типа объектов-классов Artists:

- примитивы (primitives);
- контейнеры (containers).

Примитивы

Примитивы представляют собой стандартные графические объекты:

плоскую линию (Line2D),
прямоугольник (Rectangle),
текст (Text),
изображение (AxesImage)
и т.д.

Контейнеры

Контейнеры - это объекты-хранилища, на которые можно наносить графические примитивы.

К контейнерам относятся:

- рисунок (Figure),
- область рисования (Axes),
- координатная ось (Axis),

- деления (Ticks).

Пользовательская настройка рисунков обеспечивается при помощи обращений к различным контейнерам класса Artists, объединённых логически в единую структуру.

Всего существует 4 вида Artists контейнеров:

Figure - это контейнер самого высокого уровня. Контейнеры рисунка (Figure containers). На нём располагаются все другие контейнеры и графические примитивы.

Axes - Контейнеры областей рисования (Axes containers). Именно с ними чаще всего работает пользователь. Экземпляры Axes - это области, располагающиеся в контейнере Figure, для которых можно задавать координатную систему (декартовая или полярная). На нём (экземпляр Axes) располагаются все другие контейнеры, кроме Figure, и графические примитивы. Это области на рисунке, на которых располагаются графики и диаграммы, в которые вставляются изображения и т.д. Мультиоконные рисунки состоят из набора областей Axes.

Axis - Контейнеры осей (Axis containers). Этот контейнер обслуживает экземпляры Axes. Он отвечает за создание координатных осей, на которые будут наноситься деления осей, подписи делений и линий вспомогательной сетки. Его специализация (и отличие от контейнера Tick) - это расположение делений и линий, их позиционирование и форматирование подписей делений, их отображение.

Tick - Контейнеры делений (Tick containers). Это контейнер низшего уровня. Он задаёт характеристики (цвет, толщина линий) линий сетки, делений и их подписей (размеры и типы шрифтов).

Обычная последовательность действий при создании рисунка в matplotlib: создаётся экземпляр класса Figure (Figure instance), на котором выделяют одну или нескольких областей Axes (или экземпляров Subplot), и используют вспомогательные методы экземпляра класса Axes (Axes instance) для создания графических примитивов (primitives). Если автоматически подобранные характеристики координатной сетки, делений и их подписей не устраивают пользователя, то они настраиваются с помощью экземпляров контейнеров Axis (оси) и Tick (деления осей), которые всегда присутствуют на созданной области рисования Axes.

Pyplot

Интерфейс pyplot позволяет сосредоточиться на выборе готовых решений и настройке базовых параметров рисунка. Существует стандарт вызова pyplot в python:

```
# ========================================================================
# фактически стандарт вызова pyplot в python
import matplotlib.pyplot as plt
# ========================================================================
```

Рисунки в matplotlib либо в интерактивном режиме (в консоли), либо в скрипте (текстовый файл с python-кодом), создаются путём последовательного вызова команд.

Графические элементы (точки, линии, фигуры и т.д.) наслаиваются друг друга последовательно. При этом последующие перекрывают предыдущие, если они занимают общее участки на рисунке (регулируется параметром zorder).

В matplotlib работает правило "текущей области" ("current axes"), которое означает, что все графические элементы наносятся на текущую область рисования. Несмотря на то, что областей рисования может быть несколько, одна из них всегда является текущей.

Самым главным объектом в matplotlib является рисунок Figure. Поэтому создание научной графики начинается именно с создания рисунка.

Создать рисунок в matplotlib означает задать форму, размеры и свойства основы-холста (canvas), на котором будет создаваться будущий график. Создать рисунок figure позволяет метод plt.figure(). После вызова любой графической команды, то есть функции, которая создаёт какой-либо графический объект, например, plt.scatter() или plt.plot(), всегда существует хотя бы одна область для рисования (по умолчанию прямоугольной формы).

Чтобы результат рисования, то есть текущее состояние рисунка, отразилось на экране, можно воспользоваться командой plt.show(). При этом будут показаны все рисунки (figures), которые были созданы.

```
# ========================================================================
import matplotlib.pyplot as plt
fig = plt.figure()   # Создание объекта Figure
print(fig.axes)   # Список текущих областей рисования пуст
print(type(fig))   # тип объекта Figure
plt.scatter(1.0, 1.0)   # scatter - метод для нанесения маркера
                        # в точке (1.0, 1.0)
# После нанесения графического элемента в виде маркера
# список текущих областей состоит из одной области
print (fig.axes)

plt.show()
# ========================================================================
```

В matplotlib возможно создание ВЛОЖЕННЫХ контейнеров одного типа. Далее в контейнере рисунка figure создаются вложенные контейнеры областей рисования, в каждом из которых создаются 'свои' рисунки и на 'своих' осях (объектах Axis) наносятся 'собственные' деления (масштабирование рисунка).

```
import matplotlib.pyplot as plt
import numpy as np
def set_data(start, stop, step):
    x = np.arange(start, stop, step)
    y = np.sin(x) * np.exp(x)
    z = np.cos(x) * np.sin(x)
    return(x, y, z)
def do_it(fig, y, z):
    # Создание экземпляра Axes (контейнера области рисования)
    # с помощью Figure-метода add_subplot()
    ax0 = fig.add_subplot(111)
    # Методы plot() вызываются через объекты ax, а не plt (интерфейс pyplot)
    ax0.plot(y)
    ax0.plot(z)
    ax0.grid(True)     # grid - каждому экземпляру Axes непосредственно
                       # при его создании
    # или так
    zz = z*z
    # Создание экземпляра Axes (контейнера области рисования)
    # с помощью Figure-метода add_axes()
    box = [0.25, 0.5, 0.25, 0.25]
    ax1 = fig.add_axes(box)
    ax1.plot(zz)
    ax1.grid(True)     # grid - каждому экземпляру Axes непосредственно
                       # при его создании
    # перебор ВСЕХ объектов Axes и назначение grid КАЖДОМУ из них
    # for ax in fig.axes:
    #     ax.grid(True)
```

```
# ============================================================================
if __name__ == '__main__':
    fig = plt.figure()
    print(type(fig))
    x, y, z = set_data(0.0, 1.0, 0.1)
    do_it(fig, y, z)
    plt.show()
```

■ Библиотека matplotlib

Библиотека matplotlib - это библиотека двумерной графики для языка программирования python с помощью которой можно создавать высококачественные рисунки различных форматов. Matplotlib представляет собой модуль-пакет для python.

Скачать и установить matplotlib можно с официального сайта библиотеки. В некоторых python дистрибутивах matplotlib уже предустановлен. Например, в Anaconda от Continuum Analytics.

Обычно рисунок в matplotlib представляет собой прямоугольную область, заданную в относительных координатах по обеим осям.

Второй распространённый вариант типа рисунка - круглая область (polar plot). Подробнее о таких типах графиков в главе "Графики в полярных координатах".

Чтобы сохранить получившийся рисунок нужно применить метод plt.savefig(). Он сохраняет текущую конфигурацию текущего рисунка в графический файл с некоторым расширением (png, jpeg, pdf и др.), который можно задать через параметр fmt. Поэтому её (???) нужно вызывать в конце исходного кода, после всех вызова всех других команд. Если в python-скрипте создать несколько рисунков figure и попытаться сохранить их одной командой plt.savefig(), то будет сохранён последний рисунок figure.

■ Интерфейс Pyplot

Элементы рисунка Artists...

Свойства графических элементов...

```
# ============================================================================
# Различные по форме области рисования
# ============================================================================
import matplotlib.pyplot as plt
# ============================================================================
fig = plt.figure()
# Добавление на рисунок прямоугольной (по умолчанию) области рисования
ax = fig.add_axes([0, 0, 1, 1])
print(type(ax))
plt.scatter(1.0, 1.0)
plt.savefig('example 142a.png', fmt='png')
# ============================================================================
fig = plt.figure()
# Добавление на рисунок круговой области рисования
ax = fig.add_axes([0, 0, 1, 1], polar=True)
plt.scatter(0.0, 0.5)
plt.savefig('example 142b.png', fmt='png')
# ============================================================================
plt.show()
# ============================================================================
```

Иерархическая структура рисунка в matplotlib

Всё пространство рисунка Figure (прямоугольной или иной формы) можно использовать для нанесения других элементов рисунка, например, контейнеров Axes, графических примитивов в виде линий, фигур, текста и так далее.

В любом случае каждый рисунок можно структурно представить следующим образом:

- Область рисования Axes
- Заголовок области рисования -> plt.title();
- Ось Xaxis
- Подпись оси абсцисс OX -> plt.xlabel();
- Ось Yaxis
- Подпись оси абсцисс OY -> plt.ylabel();
- Легенда -> plt.legend()
- Цветовая шкала -> plt.colorbar()
- Подпись горизонтальной оси OX -> cbar.ax.set_xlabel();
- Подпись вертикальной оси OY -> cbar.ax.set_ylabel();
- Деления на оси OX -> plt.xticks()
- Деления на оси OY -> plt.yticks()
- Для каждого из перечисленных уровней-контейнеров есть возможность нанести заголовок (title) или подпись (label).

Подписи к рисунку облегчают понимание того, в каких единицах представлены данные на графике или диаграмме. Также часто на рисунок наносятся линии вспомогательной сетки (grid).

В pyplot она вызывается командой plt.grid(). Вспомогательная сетка связана с делениями координатных осей (ticks), которые определяются автоматически исходя из значений выборки. В дальнейшем будет показано как определять положение и задавать значения делений на координатных осях.

В matplotlib существуют главные деления (major ticks) и

вспомогательные (minor ticks)

для каждой координатной оси. По умолчанию рисуются только главные деления и связанные с ними линии сетки grid.

В плане настройки главные деления ничем не отличаются от вспомогательных.

Если на рисунке присутствует так называемый "mappable object", то на рисунке может быть нарисована цветовая шкала (colorbar). К шкале также можно делать подписи вдоль разных сторон. При этом сама цветовая может быть расположена как на текущей области рисования

axes, отбирая у неё некоторую долю, либо может быть размещена на самостоятельной области рисования. Подробнее о цветовой шкале в главе "Цветовая шкала".

Элементы рисунка Artists

Список элементов рисунка (Artists) для контейнера Figure:

- axes - список экземпляров Axes (включая Subplot);
- images - список FigureImages patches (эффективно для отображения пикселей);
- legends - список экземпляров Figure Legend (отличается от Axes.legends);
- lines - список экземпляров Figure Line2D (редко используется, см. Axes.lines);
- patch - фон Rectangle;
- patches - список Figure patches (редко используется, см. Axes.patches);
- texts - список экземпляров Figure Text.

```
# ========================================================================
# Элементы простого рисунка
import matplotlib.pyplot as plt
import numpy as np
lag = 0.1
x = np.arange(0.0, 2*np.pi+lag, lag)
y = np.cos(x)
fig = plt.figure()
plt.plot(x, y)
plt.text(np.pi-0.5, 0,  '1 Axes',
                        fontsize=26
        )
plt.text(0.1, 0, '3 Yaxis',
                        fontsize=18,
                        rotation=90)
plt.text(5, -0.9, '2 Xaxis',
                        fontsize=18
        )
plt.title('1a TITLE')
plt.ylabel('3a Ylabel')
plt.xlabel('2a Xlabel ')
plt.text(5, 0.85, '6 Xticks',
                        fontsize=12,
                        rotation=90)
plt.text(0.95, -0.55, '6 Xticks',
                        fontsize=12,
                        rotation=90)
plt.text(5.75, -0.5, '7 Yticks',
                        fontsize=12
        )
plt.text(0.15, 0.475, '7 Yticks',
                        fontsize=12
        )
plt.grid(True)
plt.show()
# ========================================================================


# ========================================================================
# Элементы более сложного рисунка
import matplotlib.pyplot as plt
import numpy as np
N = 100
n = int(np.sqrt(N))
x = np.arange(n)
# Задаёся выборка из Гамма-распредления с параметрами
# формы=1. и масштаба=0.
z = np.random.random(N).reshape(n, n)
y = z[5, :]
fig = plt.figure()
cc = plt.contourf(z, alpha=0.5)  # трёхмерное поле
plt.plot(x, y, label='line', color='red')  # красная линия
plt.title('1a. Title')    # заголовок
plt.xlabel('2a. Xlabel')  # подпись оси OX
plt.ylabel('3a. Ylabel')  # подпись оси OY
plt.legend()              # легенда
cbar = plt.colorbar(cc)   # цветовая шкала
plt.text(2.5, 7, '1. Axes', fontsize=26)
plt.text(4, -0.5, '2. XAxis', fontsize=18)
plt.text(-0.5, 3.8, '3. YAxis', fontsize=18, rotation=90)
plt.text(6.3, 7.2, '4. Legend', fontsize=16)
plt.text(9.1, 5., '5. Colorbar', fontsize=16, rotation=90)
plt.text(7., 0.8, '6. Xticks', fontsize=12)
plt.text(0.8, 8.4, '7. Yticks', fontsize=12, rotation=90)
# Подписи для цветовых шкал имеют отличный от остальных подписей синтаксис
```

```
cbar.ax.set_xlabel('5a. Colorbar Xlabel', color='k', rotation=30)
cbar.ax.set_ylabel('5b. Colorbar Ylabel', color='k')
plt.text(2.8, 4.8, '6. Grid lines', fontsize=14)
plt.grid(True)
plt.show()
# ========================================================================
```

■ Свойства графических элементов

Многообразие и удобство создания графики в matplotlib обеспечивается не только за счёт созданных графических команд, но и за счёт богатого набора средств по конфигурации типовых форм. Эта настройка включает в себя работу с цветом, формой, типом линии или маркера, толщиной линий, степенью прозрачности элементов, размером и типом шрифта и другими свойствами.

Параметры, которые определяют эти свойства в различных графических командах, обычно имеют одинаковый синтаксис, то есть называются одинаково.

Стандартным способом задания свойств какого либо создаваемого объекта (или методу) является передача по ключу:

ключ = значение.

Названия параметров изменения свойств графических объектов, которые чаще всего встречаются в приложениях, перечислены ниже:

- color / colors / c - цвет;
- linewidth / linewidths - толщина линии;
- linestyle - тип линии;
- alpha - степень прозрачности (от полностью прозрачного 0 до непрозрачного 1);
- fontsize - размер шрифта;
- marker - тип маркера;
- s - размер маркера в методе plt.scatter(только цифры);
- rotation - поворот строки на X градусов.

Если в описании графического метода указано примерно так, как в plt.plot(*args, **kwargs), то это значит, что в качестве входных данных требуется сначала список / кортеж параметров (чаще всего в последовательности аргументов достаточно хотя бы одного символа, например, Y), а после этого в функцию (метод) передаются параметры, представленные парами имя-значение с фиксированными именами этих параметров (color, linewidth и т.д.).

```
# ========================================================================
import numpy as np
import matplotlib.pyplot as plt
# Пример функции с объединением в кортеж *args
def f_sums(*args):
    list1 = []
    for arg in args:
        a = 0
        for i in arg:
            a += i
        list1.append(a)
    return list1
# Пример функции с объединением в словарь **kwargs
def f_words(**kwargs):
    """
    Функция pop возвращает значение для заданного ключа
```

```
    (если он есть в словаре) и удаляет из словаря пару "ключ - значение"
    """
    print('Словарь kwargs ДО вызова метода pop:', kwargs)
    per1 = kwargs.pop('solo', 'Han')
    per2 = kwargs.pop('wookie', 'Chubbaca')
    act = kwargs.pop('loves', 'loves')
    str1 = '%s %s %s' % (per1, act, per2)
    print('Словарь kwargs ПОСЛЕ вызова метода pop:', kwargs)
    return str1
# Пример функции с объединением и в кортеж args и в словарь **kwargs =====
def f_plot(*args, **kwargs):
    xlist = []
    ylist = []
    for i, arg in enumerate(args):
        if (i % 2 == 0):
            xlist.append(arg)
        else:
            ylist.append(arg)
    colors = kwargs.pop('colors', 'k')
    linewidth = kwargs.pop('linewidth', 1.)
    fig = plt.figure()
    ax = fig.add_subplot(111)
    i = 0
    for x, y, color in zip(xlist, ylist, colors):
        i += 1
        ax.plot(x, y, color=color, linewidth=linewidth, label=str(i))
    ax.grid(True)
    ax.legend()
# ============================================================================
# MAIN SCRIPT BODY
x = np.arange(10)
y = np.random.random(20)
z = np.linspace(-15, -7.5, 37)
xyz = [x, y, z]
abc = {'solo': 1, 'wookie': 'green', 'friend': True}
res1 = f_sums(*xyz)
res2 = f_words(**abc)
print('res1', type(res1), res1)
print('res2', type(res2), res2)

"""
Т.к. в plt.plot нет обязательных параметров, то переданные
в эту функцию через зпт последовательности или массивы будут обработаны
Здесь пример передачи двух линий - две последовательности из пары OX-OY
(x, y2) и (x, y3). Им в соответствие представлена последовательность
цветов colors.
"""
colors = ['red', 'blue']
N = 10
x = np.arange(N)
y2 = np.random.random(N)
y3 = np.random.random(N)
f_plot(x, y2, x, y3, colors=colors, linewidth=2.)
# ============================================================================
```

Ещё пример задания свойств графических элементов. Создаются 3 картинки (фигуры).

```
# ============================================================================
#Свойства графических элементов ========================================
import matplotlib.pyplot as plt
import numpy as np
# ============================================================================
N = 100
```

```
x = np.arange(N)
# Выборка из Гамма-распредления с параметрами формы=1. и масштаба=0. =====
z = np.random.gamma(2., 1., N)
y = z.reshape(10, 10)
# matplotlib.pyplot ===================================================
# ========== fig_0 ==========
fig_0 = plt.figure()
cc = plt.contourf(y)
cbar = plt.colorbar(cc)
plt.title('1. TITLE', color='green')
plt.xlabel('2. X - LABEL')
plt.ylabel('3. Y - LABEL', fontsize=16)
# Подписи для цветовых шкал имеют отличный от остальных подписей синтаксис
cbar.ax.set_xlabel('4. COLORBAR X-LABEL', color='b')
cbar.ax.set_ylabel('5. COLORBAR Y-LABEL', color='r')
plt.grid(True)
# ======================================================================
# ========== fig_1 ==========
fig_1 = plt.figure()
# создание словаря для fig_1
dict_1 = {'color' : 'grey', 'linewidth' : 2.5, 'linestyle' : '--'}
xz_1 = [x, z]
# передача параметров через список xz и словарь my_dict.
# Наличие знаков * и ** обязательно!
cc = plt.plot(*xz_1, **dict_1)
# результат аналогичен такой записи
#cc = plt.plot(x, z, color='grey', linewidth=2.5, linestyle='--')
plt.scatter(
            x,
            y + 2.0,
            marker='v',
            s=10,
            color='red'
            )
plt.title('Sample from Gamma distribution')
plt.xlabel('Gamma sample values')
plt.ylabel('Sample numbers')
# Подписи для цветовых шкал
# (имеют отличный от остальных подписей синтаксис)
cbar.ax.set_xlabel('4. COLORBAR X-LABEL', fontsize=8)
cbar.ax.set_ylabel('5. COLORBAR Y-LABEL', color='r')
plt.grid(True, color='blue', linewidth=1.0)    # grid for fig_1
# ======================================================================
# ========== fig_2 ==========
fig_2 = plt.figure()
z_1_2 = np.cos(x/10.)
z_2_2 = np.cos(x/20.)
# создание словаря для fig_2
dict_2 = {'color' : 'green', 'linewidth' : 4.0, 'alpha' : 0.5}
plt.fill_between(x, z_2_2, z_1_2, color='green', alpha=0.25)
plt.plot(x, z_1_2, color='green', linewidth=4.0)
plt.plot(x, z_2_2, **dict_2)
plt.title('Different alpha values')
plt.grid(True)    # grid for fig_2
# ======================================================================
plt.show()
# ======================================================================
```

■ Figure: конфигурация рисунка

Стандартный метод создания экземпляра Figure:

```
import matplotlib.pyplot as  plt
fig = plt.figure()
print(u'Тип Figure %s' % type(fig))
```

Основные параметры созданного экземпляра Figure определяются в конфигурационном файле matplotlibrc.

Атрибуты Figure можно изменить во время создания объекта:

- figsize : кортеж из целых или действительных чисел, который определяет ширину и высоту в дюймах. По умолчанию равен значению figure.figsize из настроек rcParams;
- dpi : разрешение рисунка. По умолчанию равен значению figure.dpi из настроек rcParams;
- facecolor : цвет фона. По умолчанию равен значению figure.facecolor из настроек rcParams;
- edgecolor : цвет границы. По умолчанию равен значению figure.edgecolor из настроек rcParams.

```
import matplotlib.pyplot as  plt
from matplotlib import rcParams

figsize = (8,6)
fig = plt.figure(figsize=figsize,
                 facecolor='pink',
                 frameon=True)

plt.plot([[0.0,0.0],[1.,1.]],'k')
plt.grid(True)

rcParams['figure.edgecolor'] = 'blue'

plt.show()
```

По умолчанию цвета координатных осей, линий вспомогательной сетки отрисовываются чёрным цветом, а цвет фона (основы рисунка) - белым. Заменив параметры, отвечающие за соответствующие цвета, можно создать, например, инвертированный бело-чёрный рисунок.

```
from matplotlib import rcParams
import matplotlib.pyplot as plt
import numpy as np
# Изменение параметров рисования
# (смена чёрного по белому на белое по чёрному)
def set_rcParams(color0, color1):
    rcParams['font.family'] = 'Times New Roman', 'Arial', 'Tahoma'
    rcParams['font.fantasy'] = 'Times New Roman'
    rcParams['figure.edgecolor'] = color0
    rcParams['figure.facecolor'] = color0
    rcParams['axes.facecolor'] = color0
    #rcParams['axes.edgecolor'] = color0
    rcParams['grid.color'] = color1
    rcParams['xtick.color'] = color1
    rcParams['ytick.color'] = color1
    rcParams['axes.labelcolor'] = color1
def set_data(start, stop, step):
    x = np.arange(start, stop, step)     #(0, 4*np.pi, 0.12)
    y = np.tan(x)
    return (x, y)
# def do_it():
```

```
#
#     set_rcParams('k', 'w')
#     x, y = set_data(0, 4*np.pi, 0.12)
#
#     fig = plt.figure(facecolor='k')
#
#     plt.plot(x, y, 'w')
#     plt.grid(True, linestyle=':', color='w')
#
#     plt.xlim(x[0], x[-1])
#     plt.ylim(np.min(y), np.max(y))
#     plt.xlabel(u'Время [с]', fontsize=12)
#     plt.ylabel(u'Амплитуда [м]', fontsize=12)
def do_it():
    set_rcParams('k', 'w')
    x, y = set_data(0, 4*np.pi, 0.12)
    fig = plt.figure(facecolor='k')
    ax = fig.add_subplot()
    ax.plot(x, y, 'w')
    ax.grid(True, linestyle=':', color='w')
    plt.xlim(x[0], x[-1])
    plt.ylim(np.min(y), np.max(y))
    plt.xlabel(u'Время [с]', fontsize=12)
    plt.ylabel(u'Амплитуда [м]', fontsize=12)
if __name__ == '__main__':
    do_it()
    plt.show()
```

■ Настройки линии сетки Matplotlib.pyplot

Линии сетки (plt.grid ()) — средство визуализации.

Разница между Matplotlib gca и gcf:

gca	gcf
get current axis	get current figure
дает ссылку на текущие оси	дает ссылку на текущую картинку

```
import matplotlib.pyplot as plt
import numpy as np
plt.gcf()   # Размер холста
plt.grid()  # Создать сетку
plt.show()
```

Есть много параметров метода grid(), вот лишь некоторые из них

Оси: значение равно «оба»(?), «x», «y». Просто нарисовать линию сетки, в каком направлении. Если вводится x или y, какая ось введена, какая ось будет скрыта.

Цвет: установить цвет линий сетки. Или использовать напрямую параметр 'c' вместо цвета.

Линейный стиль: также можно использовать ls вместо обозначения линейного стиля. Установить стиль линии сетки, чтобы она была непрерывной сплошной линией, пунктирной линией или другими различными линиями.

| '-' | '--' | '-.' | ':' | 'None' | ' ' |

Ширина линии: установить ширину линии сетки

Установить ось = 'x'

Установить ось = 'y'

Установить цвет линии сетки = 'r'

Установить стиль линии linestyle = | '-' | '--' | '-.' | ':' | 'None' | ' ' |

ls = | '-' | '--' | '-.' | ':' | 'None' | ' ' |

- Картинка (figure) с разноцветной сеткой разных стилей:

```
import numpy as np
import numpymatplotlib.pyplot as plt

plt.gcf()   # Ссылка на текущую картинку form
plt.grid(axis='y', ls = '--', color='r')
plt.grid(axis='x', ls = ':',  color='b')  # Создать сетку
plt.show()
```

Примеры

Прямые и квадратичные линии

Приложение открывает несколько окон, в которых рисуются прямые и квадратичные линии.

```
# Рисование прямых и квадратичных кривых
import matplotlib.pyplot as plt
import numpy as np
def get_data(start, stop, shape):
# В указанном интервале [start, stop] возвращает множество чисел
# с равным интервалом в количестве shape.
    x=np.linspace(start, stop, shape)    # (-3,3,50)
    y1=2*x+1
    y2=x**2
    return(x, y1, y2)
"""
plt.figure(
      num = None, // номер или имя изображения, number - число, строка - имя
             // Первый по умолчанию равен 1, а затем по очереди 1, 2, 3;
             // из операции IDLE видно, что он не может быть здесь отображен.

         figsize = None, // Устанавливается размер всей области

         dpi = None,     // Указание разрешения рисованного объекта,
                         // то есть сколько пикселей на дюйм,
                         // значение по умолчанию - 80; 1 дюйм равен 2,5 см

         facecolor = None, // Цвет фона

         edgecolor = None, // цвет границы

         frameon = True, // Показывать ли фрейм

         FigureClass=<class 'matplotlib.figure.Figure'>,

         clear=False,

         **kwargs,
        )
```

```
"""
def do_it_0(x, y):
    plt.figure(facecolor='y', dpi=100, num='csd')
    plt.plot(x, y)
    plt.grid(ls=':', color='g')
def do_it_1(x, y1, y2):
    plt.figure(figsize=(8,5), facecolor='y')
    plt.plot(x, y2)
    plt.plot(x, y1, color='red', linestyle='--', linewidth=0.5)
    plt.grid(ls='-', color='b')
# ============================================================================
if __name__ == '__main__':
    x, y1, y2 = get_data(-3, 3, 50)
    do_it_0(x, y1)
    do_it_1(x, y1, y2)
    plt.show()
```

■ Оптимизация осей координат

```
# Оптимизация осей координат
import matplotlib.pyplot as plt
import numpy as np
def get_data(start, stop, shape):
# В указанном интервале [start, stop] возвращает множество чисел
# с равным интервалом в количестве shape.
    x=np.linspace(start, stop, shape)    # (-3,3,50)
    y1=2*x+1
    y2=x**2
    return(x, y1, y2)
x, y1, y2 = get_data(-3, 3, 50)
plt.figure(facecolor='y')
plt.plot(x,y2)
plt.plot(x,y1,color='red',linestyle='--',linewidth=1.0)
# Отображение диапазона значений размера изображения (координат),
plt.xlim((-1,2))
plt.ylim((-2,3))
# Добавление тегов
plt.xlabel('I am x')
plt.ylabel('I am y')
# Равномерно разделить 5 координатных точек в диапазоне отображения оси x,
# указанном в plt.xlim ((- 1,2))
new_ticks = np.linspace(-1,2,5)
print(new_ticks)
plt.xticks(new_ticks)
"""
#plt.yticks([],[],)
 Первые два параметра - это массивы,
 Вторая репрезентативная метка (может быть не числовой) заменит первую
цифровую метку по умолчанию,
 Третий параметр изменяет окончательную метку (например, форму, цвет)
"""
# Регулярные выражения показывают красивые шрифты, при печати пробелов
используется перевод '/', </ + space>
#plt.yticks([ -2,          -1,   1,        2,     3],
#           ['really bad','bad','normal','good','really good'])
plt.yticks([-2,                        -1,       1,           2,         3],
           [r'$really\ bad\ \alpha$', r'$bad$', r'$normal$', r'$good$',
r'$really\ good$'],
           rotation=30)
plt.grid()
plt.show()
```

■ Явное определение области рисования

Явное определение области рисования предполагает оптимизацию осей координат.

```
# Оптимизация осей координат
import matplotlib.pyplot as plt
import numpy as np
def get_data(start, stop, shape):
# В указанном интервале [start, stop] возвращает множество чисел
# с равным интервалом в количестве shape.
    x=np.linspace(start, stop, shape)    # (-3,3,50)
    y1=2*x+1
    y2=x**2
    return(x, y1, y2)
x, y1, y2 = get_data(-3, 3, 50)
# варианты получения ссылки на область рисования
#fig = plt.figure(facecolor='y')    # рисунок
#ax = fig.add_subplot(111)     # добавить область рисования
#ax = fig.gca()                # получить текущую область рисования от fig
ax = plt.gca()                 # получить текущую область рисования от plt
# линии на области рисования ============================================
ax.plot(x, y1, color='red', linestyle='--', linewidth=1.0)
ax.plot(x, y2)
# Отображение диапазона значений размера изображения (координат)
plt.xlim((-1, 2))
plt.ylim((-2, 3))
# Добавление тегов
plt.xlabel('I am x')
plt.ylabel('I am y')
# Равномерно разделить 5 координатных точек в диапазоне отображения оси x,
# указанном в plt.xlim ((- 1, 2))
new_ticks = np.linspace(-1, 2, 5)
print(new_ticks)          # распечатать значения списка новых координат
plt.xticks(new_ticks)     # нанести новые координаты
"""
#plt.yticks([],[],)
 Первые два параметра - это массивы,
 Вторая репрезентативная метка (может быть не числовой) заменит первую
цифровую метку по умолчанию,
 Третий параметр изменяет окончательную метку (например, форму, цвет)
"""
# Регулярные выражения показывают красивые шрифты, при печати пробелов
используется перевод '/', </ + space>
#plt.yticks([ -2,          -1,   1,        2,     3],
#           ['really bad','bad','normal','good','really good'])
plt.yticks([-2,                        -1,       1,           2,          3],
           [r'$really\ bad\ \alpha$', r'$bad$', r'$normal$', r'$good$',
r'$really\ good$'],
           rotation=30)
plt.grid()
plt.show()
```

■ Нормальная общая система координат

Нормальная общая система координат применяется при решении математических задач.

```
# нормальные координаты x, y
# (общая система координат, которая
# применяется для решения математических задач)
import matplotlib.pyplot as plt
import numpy as np
def get_data(start, stop, shape):
# В указанном интервале [start, stop] возвращает множество чисел
# с равным интервалом в количестве shape.
```

```
    x=np.linspace(start, stop, shape)    # (-3,3,50)
    y1=2*x+1
    y2=x**2
    return(x, y1, y2)
x, y1, y2 = get_data(-3, 3, 50)
plt.figure()
plt.plot(x, y2)
plt.plot(x, y1, color='red', linestyle='--', linewidth=1.0)
# Отображение диапазона значений размера изображения (координат),
plt.xlim((-1, 2))
plt.ylim((-2, 3))
# Добавление тегов
plt.xlabel('I am x')
plt.ylabel('I am y')
# Равномерно разделить 5 координатных точек в диапазоне отображения оси x,
# указанном в plt.xlim ((- 1, 2))
new_ticks=np.linspace(-1, 2, 5)
print(new_ticks)            # распечатать значения списка новых координат
plt.xticks(new_ticks)       # нанести новые координаты
#plt.yticks([-2,-1,1,2,3],['really bad','bad','normal','good','really good'])
# Регулярные выражения показывают красивые шрифты
plt.yticks([-2,-1,1,2,3],[r'$really\ bad\
\alpha$',r'$bad$',r'$normal$',r'$good$',r'$really\ good$'])
# gca = 'получить текущую ось'
ax=plt.gca()
# Сделать прозрачными правую и верхнюю оси, скрыть spines
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
# Установить ось xaxis, полученную с помощью ax, как ось 'bottom'
# Установить ось yaxis, полученную с помощью ay, на ось 'left'
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Установить начальную точку x, y как: (0, 0) origin
ax.spines['bottom'].set_position(('data', 0))
ax.spines['left'].set_position(('data', 0))
ax.grid()
plt.show()
```

■ Легенда

```
# нормальные координаты x, y
# (общая система координат, которая
# применяется для решения математических задач) и легенда
import matplotlib.pyplot as plt
import numpy as np
def get_data(start, stop, shape):
# В указанном интервале [start, stop] возвращает множество чисел
# с равным интервалом в количестве shape.
    x=np.linspace(start, stop, shape)    # (-3,3,50)
    y1=2*x+1
    y2=x**2
    return(x, y1, y2)
x, y1, y2 = get_data(-3, 3, 50)
plt.figure()
plt.plot(x, y2, color='blue')
l1 = plt.plot(0, 0, label=u'x**2', color='blue')
plt.plot(x, y1, color='red', linestyle='--', linewidth=1.0)
l2 = plt.plot(0, 0, label=u'2*x + 1', color='red')
#legend: Описание значка
# handles: Размер рамки может быть автоматически установлен в соответствии
#          с размером этикеток.
# Параметр loc: определяет положение значка при отображении по умолчанию
#               как «наилучшее», дополнительные параметры:
```

```
# «верхний центр», «нижний центр», «центральный левый», «центральный правый»
# и т. д.
legends=l1 + l2
labels = [l.get_label() for l in legends]
plt.legend(legends,
           labels,
           loc=4)
# Отображение диапазона значений размера изображения (координат),
plt.xlim((-1, 2))
plt.ylim((-2, 3))
# Добавление тегов
plt.xlabel('I am x')
plt.ylabel('I am y')
# Равномерно разделить 5 координатных точек в диапазоне отображения оси x,
# указанном в plt.xlim ((- 1, 2))
new_ticks = np.linspace(-1, 2, 5)
print(new_ticks)            # распечатать значения списка новых координат
plt.xticks(new_ticks)       # нанести новые координаты
#plt.yticks([-2,-1,1,2,3],['really bad','bad','normal','good','really good'])
# Регулярные выражения показывают красивые шрифты
plt.yticks([-2,                        -1,       1,           2,         3],
           [r'$really\ bad\ \alpha$', r'$bad$', r'$normal$', r'$good$',
r'$really\ good$'])
# gca = 'получить текущую ось'
ax = plt.gca()
# Сделать прозрачными правую и верхнюю оси, скрыть spines
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
# Установить ось xaxis, полученную с помощью ax, как ось 'bottom'
# Установить ось yaxis, полученную с помощью ay, на ось 'left'
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Установить начальную точку x, y как: (0, 0) origin
ax.spines['bottom'].set_position(('data', 0))
ax.spines['left'].set_position(('data', 0))
ax.grid()
plt.show()
```

■ Аннотации и собственный текст

Добавление аннотаций к особым точкам на изображении, добавление текста относительно изображения на области рисования

```
# Добавление аннотаций к особым точкам на изображении
# Добавление текста относительно изображения на области рисования
import matplotlib.pyplot as plt
import numpy as np
# ============================================================================
def get_data_0(start, stop, shape):
# В указанном интервале [start, stop] возвращает множество чисел
# с равным интервалом в количестве shape.
    x=np.linspace(start, stop, shape)    # (-3,3,50)
    y=2*x+1
    return(x, y)
# ============================================================================
# x = np.linspace(-3, 3, 50)
# y = 2*x+1
x, y = get_data_0(-3, 3, 50)
plt.figure(num=1,figsize=(8,5))
#plt.plot(x, y)
ax=plt.gca()
ax.plot(x, y)
```

```
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data', 0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data', 0))
ax.grid()
# x0=1
# y0=2*x0+1
x0, y0 = get_data_0(1, 1, 1)     # get_data_0 возвращает два массива
# получение пары значений из массивов x0, y0
_x0 = int(x0[0])
_y0 = int(y0[0])
# Задать точку рассеивания, s: размер точки.
ax.scatter(_x0, _y0, s=50, color='b')
#plt.scatter(_x0, _y0, s=50, color='b')
# Между двумя точками [(_x0, _y0), (_x0, 0)]
#                   нарисовать черную пунктирную линию шириной 0.5.
ax.plot([_x0, _x0], [_y0, 0],              'k--',              lw=0.5)
#plt.plot([_x0, _x0], [_y0, 0],              'k--',              lw=0.5)
# method 1 ============================================================
#annotate: аннотация
# % y: эквивалентно символу адреса & в языке C и присваивает значение % s.
# xy: Установка относительного положения текста (2x + 1 = 3) относительно
#     точки аннотации.
# textcoords = 'offset points': значение по умолчанию,
#              параноидальная точка (отклонение от значения настройки
#              xytext (x0, y0)) и, наконец, использование ее в качестве
#              точки привязки для рисования кривой аннотации до (x0, y0)
#              xytext = (x1, y1) задание значения для точки настройки
#arrowprops = установка стиля: стрелка, угол, радиус
# ====================================================================
# plt.annotate(r'$2x+1=%s$'%_y0,
#              xy=(_x0, _y0),
#              xycoords='data',
#              xytext=(+30,-30),
#              textcoords='offset points',
#              fontsize=16,
#              arrowprops=dict(arrowstyle='->',
#                              connectionstyle='arc3, rad= .2')
#              )
ax.annotate(r'$2x+1=%s$'%_y0,
            xy=(_x0, _y0),
            xycoords='data',
            xytext=(+30,-30),
            textcoords='offset points',
            fontsize=16,
            arrowprops=dict(arrowstyle='->',
                            connectionstyle='arc3, rad= .2')
            )
# method 2 ============================================================
"""
plt.text(x,y,s,fontdict,)
 x, y: где разместить текст
 s: str текстовое содержимое.
 fontdict: словарь, используемый для переопределения атрибутов текста
 по умолчанию (пользовательский стиль текста)
"""
# ====================================================================
plt.text(-3.7,
         5,
         r'$This\ is\ the\ some\ text.\ \mu\ \sigma_i\ \alpha_t$',
         fontdict={'size':16,'color':'r'})
plt.show()
```

■ Прозрачность линии

Координаты иногда скрыты линиями графика — установить прозрачность линии, чтобы они отображалась.

```
# Координаты иногда скрыты линиями графика — установить
# прозрачность линии, чтобы они отображалась.
import matplotlib.pyplot as plt
import numpy as np
# ============================================================================
def get_data_0(start, stop, shape):
# В указанном интервале [start, stop] возвращает множество чисел
# с равным интервалом в количестве shape.
    x=np.linspace(start, stop, shape)    # (-3,3,50)
    y=0.1*x
    return(x, y)
# ============================================================================
x, y = get_data_0(-3, 3, 50)
plt.figure()
plt.ylim(-2, 2)
ax=plt.gca()
#alpha: коэффициент прозрачности
ax.plot(x, y, linewidth=10, color='green', alpha=0.5)
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data', 0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data', 0))
ax.grid()
# Изменить метки (координаты) осей x и y,
# установить размер шрифта 12,
# метки поместить в рамки (красные с зелёной каёмочкой, полупрозрачные)
for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_fontsize(12)
    #pass
    label.set_bbox(dict(facecolor='red',
                        edgecolor='green',
                        #edgecolor='None',
                        alpha=0.5)
                   )
plt.show()
```

■ Точечная диаграмма

```
# Точечная диаграмма ========================================================
import matplotlib.pyplot as plt
import numpy as np
n=1024
X=np.random.normal(0,1,n)
Y=np.random.normal(0,1,n)
T=np.arctan2(Y,X)  #for color value
"""
np.arange()
Для параметра значение параметра - это конечная точка,
начальная точка - значение по умолчанию 0,
а длина шага - значение по умолчанию 1.
В случае двух параметров первый параметр является начальной точкой,
второй параметр - конечной точкой, длина шага - значением по умолчанию 1.
При наличии трех параметров первый параметр является начальной точкой,
второй параметр - конечной точкой, а третий параметр - длиной шага.
Если размер шага поддерживает десятичные дроби (?)
```

```
"""
plt.scatter(X,Y,s=75,c=T,alpha=0.5)
# plt.scatter Настроить точечную диаграмму.
#plt.scatter(np.arange(5),np.arange(5))
#plt.xlim((-1.5,1.5))
#plt.ylim((-1.5,1.5))
# Не передавать параметры, удалить периферийные координаты
# plt.xticks(())
# plt.yticks(())
plt.grid(True)
plt.show()
```

■ Барная гистограмма

Барная гистограмма случайная выборка из равномерного распределения [низкий, высокий), важно, что домен является закрытым слева и открытым справа, то есть содержит низкий, а не высокий (?), n - количество точек выборки.

```
# Барная гистограмма ====================================================
import matplotlib.pyplot as plt
import numpy as np
def get_data(n):
    n = n
    X = np.arange(12)
    # 1/float(2)=0.5
    # np.random.uniform: Случайная выборка из равномерного распределения
    # [низкий, высокий), важно, что домен является закрытым слева
    # и открытым справа, то есть содержит низкий, а не высокий,
    # n – количество точек выборки
    Y1 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
    Y2 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
    return(X, Y1, Y2)
X, Y1, Y2 = get_data(12)
# bar: Репрезентативная гистограмма
ax = plt.gca()
# plt.bar(X, +Y1, facecolor='red', edgecolor='white')
# plt.bar(X, -Y2, facecolor='blue', edgecolor='white')
ax.bar(X, +Y1, facecolor='red', edgecolor='white')
ax.bar(X, -Y2, facecolor='blue', edgecolor='white')
# zip: x, y выражаются в единицах кортежей,
# а значения последовательно берутся в массивах X и Y.
for x, y in zip(X, Y1):
    # ha:horizontal alignment
    # plt.text(x + 0.4, y + 0.05, '%.2f' % y, ha='center', va='bottom')
    ax.text(x + 0.4, y + 0.05, '%.2f' % y, ha='center', va='bottom')
for x, y in zip(X, Y2):
    # ha:horizontal alignment
    # plt.text(x + 0.4, -y - 0.05, '%.2f' % y, ha='center', va='top')
    ax.text(x + 0.4, -y - 0.05, '%.2f' % y, ha='center', va='top')
# plt.xticks(())
# plt.yticks(())
ax.grid()
plt.show()
```

■ Диаграмма высокого и низкого потенциала

```
"""
 Диаграмма высокого и низкого потенциала энергии
```

```
 np.linspace (x, y, n): делить поровну на n как общее
 np.arange (x, y, n): делить поровну на n как размер шага
"""

import matplotlib.pyplot as plt
import numpy as np
# ====================================================================
def f(x,y,r1,r2):
    #b = (1-x/2+x**5+y**3)*np.exp(-x**2-y**2)
    b = (r2-x/r1+x**5+y**3)*np.exp(-x**2-y**2)
    #the height function
    return b
# ====================================================================
n = 256
x = np.linspace(-3,3,n)
y = np.linspace(-3,3,n)
#meshgrid: Сетка
"""
 np.meshgrid(x, y) возвращает координаты двумерной сетки
                   на основе координат, содержащихся
                   в векторе x и векторе y.
 x - вектор [1 2 3], y - вектор [1 2 3 4 5];
 Каждая строка матрицы X равна x, то есть [1 2 3],
                        общая длина (y) = 5 строк;
 Каждый столбец матрицы Y равен y, то есть [1 2 3 4 5],
                        общая длина (x) = 3 столбца.
X =                                 Y =
    1 2 3                               1    1    1
    1 2 3                               2    2    2
    1 2 3                               3    3    3
    1 2 3                               4    4    4
    1 2 3                               5    5    5
"""
X, Y = np.meshgrid(x, y)
# Контур (contour) ===================================================
# use plt.contourf to filling contours
# X,Y and value for( X,Y) point
# 8: 8 делений
# cmap = color_map contourf: заливка контура.
ax = plt.gca()
r1 = (np.random.uniform(3, 9, 1))
r1= int(r1)
r2 = (np.random.uniform(2, 6, 1))
r2= int(r2)
# plt.contourf(X,Y,f(X,Y),8,alpha=0.75,cmap=plt.cm.hot)
ax.contourf(X, Y, f(X, Y, r1, r2), 8, alpha=0.75, cmap=plt.cm.hot)
# использовать plt.contour, чтобы добавить контурные линии contour:
# позиционирование линий
# C=plt.contour(X,Y,f(X,Y),8,colors='black',linewidths=0.5)
C = ax.contour(X, Y, f(X, Y, r1, r2), 8, colors='black', linewidths=0.5)
#adding label
#inline: Добавить ярлык онлайн
#plt.clabel(C,inline=True,fontsize=10)
ax.clabel(C, inline=True, fontsize=10)
#plt.grid()
ax.grid()
# plt.xticks(())
# plt.yticks(())
plt.show()
```

■ Цветовой блок и аннотация градиентной гистограммы

```
# Цветовой блок и аннотация градиентной гистограммы ====================
```

```
import matplotlib.pyplot as plt
import numpy as np
#image data
# Указать цвет по номеру, передть массив, а затем изменить форму (3,3),
# чтобы изменить форму матрицы 3X3
a=np.array([0.313660827978,0.365348418405,0.423733120134,
           0.365348418405,0.439599930621,0.525083754405,
           0.423733120134,0.525083754405,0.651536351379]).reshape(3,3)
plt.figure(figsize=(8,5),facecolor='white')
"""
fornthe value of"interpolation",check this:
http://matplotlib.org/examples/images_contours_and_fields/interpolation_metho
ds.html
"""
#cmap:color map;
# cmap = 'bone': установите белый цвет изображения.
#origin отображается относительно начала и конца исходной матрицы 3X3, origin
= 'upper'
#interpolation='nearest'
plt.imshow(a,cmap='bone',origin='lower')
#shrink: определение коэффициента сжатия панели комментариев относительно
цветового блока.
plt.colorbar(shrink=0.9)
plt.grid()
# plt.xticks(())
# plt.yticks(())
plt.show()
```

■ 3D изображение и отображение плоскостей

Для рисования трехмерных графиков функций (поверхностей) и применения средств настройки внешнего вида графиков нужны библиотека Matplotlib и математическая библиотека numpy.

Библиотека numpy также позволяет значительно сократить количество строк кода, а некоторые методы классов рисования трехмерных графиков в качестве параметров ожидают экземпляры класса numpy.array.

Для трехмерного графика в первую очередь надо создать трехмерные оси. Чтобы создать трехмерные оси, сначала надо загрузить модуль (модуль ???) Axes3D. Затем с помощью функции figure() создаётся экземпляр класса matplotlib.figure.Figure.

Дальше у экземпляра этого класса вызывается метод add_subplot(), который может принимать различное количество параметров, но в данном случае ему передаётся один именованный параметр - projection, который описывает тип осей. Для создания трехмерных осей значение параметра projection должно быть строкой "3d". В результате получается экземпляр (объект) класса осей, с которым можно в дальнейшем работать.

Этот экземпляр принадлежит к классу, производному от Axes3D (он включает класс matplotlib.axes._subplots.Axes3DSubplot, но этот класс создается динамически в библиотеке, и его описания нет в документации). С помощью объекта этого класса будут рисоваться графики и настраиватся их внешний вид.

```
# И ничего кроме осей! ========================================================
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
if __name__ == '__main__':
    fig = plt.figure()
    # возможны 2 варианта создания трехмерных осей от экземпляра =========
    # класса Figure. Но сначала надо загрузить модуль (модуль ???) Axes3D
```

```
    axes = fig.add_subplot(projection='3d')
    #axes = Axes3D(fig)
    # ============================================================================
    plt.show()

# Полученные оси можно вращать мышкой.
```

Полученные оси можно вращать мышкой. Также должен быть график. Для этого примера — функция от двух координат, которая будет рисоваться на осях:

f(x, y) = sin(x) * sin(y) / x*y

Нужно подготовить данные для рисования. Для этого нужны три двумерные матрицы: матрицы X и Y, которые будут хранить координаты сетки точек, в которых будет вычисляться приведенная выше функция, а матрица Z будет хранить значения этой функции в соответствующей точке (x,y).

Дальше нужно подготовить прямоугольную сетку на плоскости XY, в узлах которой будут рассчитаны значения значения по оси Z (значения отображаемой функции). Для создания такой сетки можно применить функцию numpy.meshgrid(). В простейшем случае эта функция, принимает несколько одномерных массивов. Здесь нужны ДВА массива для осей X и Y, которые содержат значения координат узлов вдоль соответствующей оси, и могут иметь разный размер. Эта функция возвратит две двумерные матрицы, описывающие координаты X и Y на двумерной сетке.

Первая матрица будет создана 'размножением' первого переданного параметра в функцию numpy.meshgrid() вдоль СТРОК первой возвращаемой матрицы, вторая матрица создается 'размножением' второго переданного одномерного массива вдоль СТОЛБЦОВ второй матрицы. Пример:

```
#
# X, Y = numpy.meshgrid([1, 2, 3], [4, 5, 6, 7])
#
# X = array([
#             [1,2,3],
#             [1,2,3],
#             [1,2,3],
#             [1,2,3]
# ])
#
# Y = array([
#             [4,4,4],
#             [5,5,5],
#             [6,6,6],
#             [7,7,7]
# ])
#
#  по индексу узла сетки можно узнать реальные координаты:
#  X[0][0] = 1, Y[0][0] = 4 и т.д.
```

Такие двумерные матрицы позволяют легко (???) рассчитывать значения функций от двух аргументов, используя поэлементные операции (векторизацию). Эти матрицы требуются для рисования трехмерных поверхностей в Matplotlib.

Функция makeData возвращает три двумерные матрицы: x, y, z.

Координаты x и y лежат на отрезке от -10 до 10 с шагом 0.1.

Для задания интервала по осям X и Y использована функци numpy.linspace(), которая создает одномерный массив со значениями из заданного интервала [-10; 10] с указанным количеством отсчетов в ней (в данном случае - 100). По умолчанию правый конец интервала также включается в результат (если правый конец не нужно включать в результат, то можно передать параметр endpoint, равный False). Для простоты по осям берется четное количество отсчетов, благодаря этому особая (!) точка с координатами (0, 0) не попадает в создаваемую сетку, поэтому дополнительная проверка не проводится.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy
from array import *


def makeData(x, y):
    xgrid, ygrid = numpy.meshgrid(x, y)
    z = numpy.sin(xgrid) * numpy.sin(ygrid) / (xgrid * ygrid)
    return xgrid, ygrid, z
# Чтобы отобразить полученные данные, достаточно вызвать ================
# метод plot_surface() экземпляра класса Axes3D, в который надо передать
# полученные с помощью функции makeData() двумерные матрицы.
if __name__ == '__main__':
    # Строится сетка в интервале от -10 до 10, имеющая 100 отсчетов по
    # обоим координатам
    x = numpy.linspace(-10, 10, 100)
    y = numpy.linspace(-10, 10, 100)
    X, Y, Z = makeData(x, y)
    fig = plt.figure()
    axes = Axes3D(fig)
    #axes.plot_surface(X, Y, Z)
    # Шаг сетки
    # Есть два инструмента для изменения прореживания данных:
    # (rcount, ccount), (rstride, cstride)
    # По умолчанию метод plot_surface() использует разрежение исходных
    # данных, чтобы ускорить отображение. Регулировать степень разрежения
    # (или даже отключить его) можно с помощью пар именованных параметров,
    # передаваемых в метод plot_surface().
    # С помощью параметров rcount и ccount можно задать количество
    # отсчетов по двум осям (по строкам и по столбцам в исходных данных).
    # По умолчанию используются значения rcount = ccount = 50.
    # Здесь rcount = ccount = 100.
    # Пара параметров rstride и cstride задают степень
    # прореженности (децимации) по осям, то есть сколько отсчетов
    # надо пропустить.
    axes.plot_surface(X, Y, Z, rcount = 100, ccount = 100)
    plt.show()
    # ========================================================================
```

■ Изменение цвета графика

Можно изменить цвет поверхности с помощью параметра color. Значение этого параметра представляет собой строку, которая описывает цвет. Строка цвета может задаваться разными способами.

Цвет можно определить английским словом для соответствующего цвета или одной буквой:

'b' или 'blue'

'g' или 'green'

'r' или 'red'

'c' или 'cyan'

'm' или 'magenta'

'y' или 'yellow'

'k' или 'black'

'w' или 'white'

серый цвет, его яркость можно задать с помощью строки, содержащей число в интервале от 0.0 до 1.0 (0 - белый, 1 - черный). Например, можно написать следующую строку:

axes.plot_surface(x, y, z, color='0.7')

Можно задавать цвет так, как это принято в HTML после символа решетки ('#'). Например:

axes.plot_surface(x, y, z, color='#11aa55')

■ Применение цветовых карт (colormap)

Цветовые карты используются, если нужно указать в какие цвета должны окрашиваться участки трехмерной поверхности в зависимости от значения Z в этой области (задание цветового градиента). Это применение градиентов.

Чтобы при выводе графика использовался градиент, в качестве значения параметра cmap (от colormap - цветовая карта) нужно передать экземпляр класса matplotlib.colors.Colormap или производного от него.

```
from matplotlib.colors import LinearSegmentedColormap
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy
from array import *
def makeData(x, y):
    xgrid, ygrid = numpy.meshgrid(x, y)
    z = numpy.sin(xgrid) * numpy.sin(ygrid) / (xgrid * ygrid)
    return xgrid, ygrid, z
if __name__ == '__main__':
    # Строится сетка в интервале от -10 до 10, имеющая 100 отсчетов по
    # обоим координатам
    x = numpy.linspace(-10, 10, 100)
    y = numpy.linspace(-10, 10, 100)
    X, Y, Z = makeData(x, y)
    fig = plt.figure()
    axes = Axes3D(fig)
    # !!! для создания цветовой карты используется статический метод
    # LinearSegmentedColormap.from_list(). Принимает три параметра:
    # = Имя создаваемой карты
    # = Список цветов, начиная с цвета для минимального значения
    #   на графике (голубой - 'b'), через промежуточные цвета
    #   (здесь это белый - 'w') к цвету для максимального значения
    #   функции (красный - 'r').
    # = Количество цветовых переходов. Чем это число больше,
    #                                  тем более плавный градиент,
    #                                  но тем больше памяти он занимает.
    cmap = LinearSegmentedColormap.from_list('red_blue', ['b','w','r'], 256)
```

```
    axes.plot_surface(X,Y,Z,color='#11aa55',
                              cmap=cmap,
                              rcount = 100,
                              ccount=100)
    plt.show()
```

■ Список встроенных цветовых карт и их применение

['Accent', 'Accent_r', 'Blues', 'Blues_r', 'BrBG', 'BrBG_r', 'BuGn',

'BuGn_r', 'BuPu', 'BuPu_r', 'CMRmap', 'CMRmap_r', 'Dark2', 'Dark2_r',

'GnBu', 'GnBu_r', 'Greens', 'Greens_r', 'Greys', 'Greys_r', 'LUTSIZE',

'MutableMapping', 'OrRd', 'OrRd_r', 'Oranges', 'Oranges_r', 'PRGn',

'PRGn_r', 'Paired', 'Paired_r', 'Pastel1', 'Pastel1_r', 'Pastel2',

'Pastel2_r', 'PiYG', 'PiYG_r', 'PuBu', 'PuBuGn', 'PuBuGn_r', 'PuBu_r',

'PuOr', 'PuOr_r', 'PuRd', 'PuRd_r', 'Purples', 'Purples_r', 'RdBu',

'RdBu_r', 'RdGy', 'RdGy_r', 'RdPu', 'RdPu_r', 'RdYlBu', 'RdYlBu_r',

'RdYlGn', 'RdYlGn_r', 'Reds', 'Reds_r', 'ScalarMappable', 'Set1',

'Set1_r', 'Set2', 'Set2_r', 'Set3', 'Set3_r', 'Spectral', 'Spectral_r',

'Wistia', 'Wistia_r', 'YlGn', 'YlGnBu', 'YlGnBu_r', 'YlGn_r', 'YlOrBr',

'YlOrBr_r', 'YlOrRd', 'YlOrRd_r',

'_api', '_cmap_registry', '_gen_cmap_registry', 'afmhot', 'afmhot_r',

'autumn', 'autumn_r', 'binary', 'binary_r', 'bone', 'bone_r', 'brg',

'brg_r', 'bwr', 'bwr_r', 'cbook', 'cividis', 'cividis_r', 'cmap_d',

'cmaps_listed', 'colors', 'cool', 'cool_r', 'coolwarm', 'coolwarm_r',

'copper', 'copper_r', 'cubehelix', 'cubehelix_r', 'datad', 'flag',

'flag_r', 'get_cmap', 'gist_earth', 'gist_earth_r', 'gist_gray',

'gist_gray_r', 'gist_heat', 'gist_heat_r', 'gist_ncar', 'gist_ncar_r',

'gist_rainbow', 'gist_rainbow_r', 'gist_stern', 'gist_stern_r',

'gist_yarg', 'gist_yarg_r', 'gnuplot', 'gnuplot2', 'gnuplot2_r',

'gnuplot_r', 'gray', 'gray_r', 'hot', 'hot_r', 'hsv', 'hsv_r',

'inferno', 'inferno_r', 'jet', 'jet_r', 'ma', 'magma', 'magma_r', 'mpl',

'nipy_spectral', 'nipy_spectral_r', 'np', 'ocean', 'ocean_r', 'pink',

'pink_r', 'plasma', 'plasma_r', 'prism', 'prism_r', 'rainbow',

'rainbow_r', 'register_cmap', 'seismic', 'seismic_r', 'spring',

'spring_r', 'summer', 'summer_r', 'tab10', 'tab10_r', 'tab20',

'tab20_r', 'tab20b', 'tab20b_r', 'tab20c', 'tab20c_r', 'terrain',

'terrain_r', 'turbo', 'turbo_r', 'twilight', 'twilight_r',

'twilight_shifted', 'twilight_shifted_r', 'unregister_cmap', 'viridis',

'viridis_r', 'winter', 'winter_r']

```
# В примере строится два трехмерных графика в одном окне.
# Для левого графика цветовая карта jet, а для правого - Spectral.
from matplotlib.colors import LinearSegmentedColormap, LightSource
# from matplotlib.colors import LightSource
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy
from array import *
def makeData():
    # Сетка в интервале от -10 до 10, имеет 100 отсчетов по обоим координатам
    x = numpy.linspace(-10, 10, 100)
    y = numpy.linspace(-10, 10, 100)
    # Создаётся двумерная матрица-сетка
    xgrid, ygrid = numpy.meshgrid(x, y)
    # В узлах рассчитываем значение функции
    zgrid = numpy.sin(xgrid) * numpy.sin(ygrid) / (xgrid * ygrid)
    return xgrid, ygrid, zgrid
if __name__ == '__main__':
    x, y, z = makeData()
    fig = plt.figure(figsize=(10, 8))
    axes_1 = fig.add_subplot(1, 2, 1, projection='3d')
    axes_2 = fig.add_subplot(1, 2, 2, projection='3d')
    # !!!
    axes_1.plot_surface(x, y, z, cmap='jet')
    axes_1.set_title("cmap='jet'")
    # !!!
    axes_2.plot_surface(x, y, z, cmap='Spectral')
    axes_2.set_title("cmap='Spectral'")
    plt.show()
```

■ Цвета и толщины линий линии сетки трехмерной поверхности

Можно сделать более выделяющимися линии сетки трехмерной поверхности. Цвет линий в методе plot_surface() задаётся с помощью именованного параметра edgecolors, а толщина линий - с помощью параметра linewidth.

Далее линии сетки можно сделать более жирными и черными. Для наглядности сетка может быть сделана более редкой с помощью параметров rcount и ccount, а для раскраски поверхности с помощью параметра cmap задается цветовая карта "jet".

```
from matplotlib.colors import LinearSegmentedColormap, LightSource
# from matplotlib.colors import LightSource
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy
from array import *
def makeData():
    # Строится сетка в интервале от -10 до 10,
    # имеющая 100 отсчетов по обоим координатам
```

```
    x = numpy.linspace(-10, 10, 100)
    y = numpy.linspace(-10, 10, 100)
    # Создаётся двумерная матрицу-сетку
    xgrid, ygrid = numpy.meshgrid(x, y)
    # В узлах рассчитываются значения функции
    z = numpy.sin(xgrid) * numpy.sin(ygrid) / (xgrid * ygrid)
    return cxgrid, ygrid, z
if __name__ == '__main__':
    x, y, z = makeData()
    fig = plt.figure()
    axes = fig.add_subplot(projection='3d')
    # !!!
    axes.plot_surface(x, y, z, rcount=40,
                              ccount=40,
                              cmap='jet',
                              linewidth=0.5,
                              edgecolors='k')
    plt.show()
```

■ Пример 3d изображения и отображения плоскостей

```
# 3D изображение и отображение плоскостей ================================
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(facecolor='white')
ax = Axes3D(fig)
#X,Y value
x=np.arange(-4,4,0.25)
y=np.arange(-4,4,0.25)
X,Y=np.meshgrid(x,y)
R=np.sqrt(X**2+Y**2)
#height value
Z=np.cos(R)
# r, cstride: плотность черных линий
# rstride: интервал по горизонтальной оси;
# cstride: интервал по вертикальной оси
# радуга: цвета радуги
ax.plot_surface(X,
                Y,
                Z,
                rstride=1,
                cstride=1,
                edgecolor='black',
                cmap=plt.get_cmap('rainbow')
                )
ax.contourf(X,Y,Z,
            zdir='z',
            offset=-2,
            cmap='rainbow'
            )
ax.set_zlim(-2, 2)
plt.show()
```

■ Анимация

Далее приводится несколько примеров отображения графиков, которые обеспечиваются функциями из пакета pyplot библиотеки matplotlib, обозначенного как plt.

■ Анимация в цикле

В этом модуле отображение графиков (результат выполнения функции gaussian) обеспечивается функциями из пакета pyplot библиотеки matplotlib, обозначенного как plt. При импорте пакета pyplot создается готовый к работе объект plt (???) со всеми его графическими возможностями. Нужно всего лишь использовать функцию plot() и передать ей массив (возможно кортеж) значений, по которым строится график. В результате будет создан объект Line2D. Это линия, которая представляет собой последовательность точек, нанесённую на график. Далее с помощью функции plt.show() нужно показать этот график.

```
import matplotlib.pyplot as plt
import numpy
import time
# ======================================================================
def gaussian(x, delay, sigma):
    """
    Функция, график которой будет отображаться в процессе анимации
    """
    return numpy.exp(-((x - delay) / sigma) ** 2)
# ======================================================================
def gaussmake(data, delay, sigma, maxSize, plt):
    y = gaussian(data, delay, sigma)
    # новый гауссов график на чистую ТЕКУЩУЮ фигуру =
    plt.clf()  # !!! Очистить текущую (НЕ НОВУЮ !)
    # фигуру. Использование новой фигуры приводит к
    # неограниченному росту окон приложения ==========
    plt.plot(data, y)    # Отображение графика
    # Установка отображаемых интервалов по осям ======
    plt.xlim(0, maxSize)    # по оси X
    plt.ylim(-1.1, 1.1)     # по оси Y
    # добавление сетки по осям =======================
    plt.axis([0, maxSize, -1.1, 1.1])
    plt.grid(True)
    # ================================================
    # !!! Вызовы для обновления графика
    plt.draw()
    plt.gcf().canvas.flush_events()    # Метод plt.gcf()
                                       # возвращает объект типа Figure,
                                       # который отвечает за окно графика.
    return y   # возвращаемое значение
# ======================================================================
if __name__ == '__main__':
    # Параметры отображаемой функции
    maxSize = 200
    sigma = 10.0
    # Диапазон точек для расчета графика функции [0, maxSize)
    xforward = numpy.arange(0, maxSize, 1)
    xbackward = numpy.arange(maxSize, 0, -1)
    # Значения графика функции
    y = numpy.zeros(maxSize)    # 200 нулей
# ======================================================================
    # !!! Включить интерактивный режим для анимации
    plt.ion()    # ==================================
    # У функции gaussian будет меняться параметр delay
    factor = 1.0
    for i in range(0, 10):
        if factor > 0:
            for delay in numpy.arange(0.0, 200.0, factor):
                y = gaussmake(xforward, delay, sigma, maxSize, plt)
                # возвращаемое значение здесь не применяется !
                # Задержка перед следующим обновлением
                time.sleep(0.01)
            factor = factor * -1
        if factor < 0.0:
```

```
            for delay in numpy.arange(200.0, 0.0, factor):
                y = gaussmake(xbackward, delay, sigma, maxSize, plt)
                # возвращаемое значение здесь не применяется !
                # Задержка перед следующим обновлением
                time.sleep(0.01)
            factor = factor * -1.0
    # Отключить интерактивный режим по завершению анимации
    plt.ioff()    # ================================
    # Нужно, чтобы график не закрывался после завершения анимации
    plt.show()
```

В примере до начала цикла график matplotlib с помощью функции ion() переводится в интерактивный режим. Затем в цикле производится очистка графика и добавление новой кривой с помощью функции plot(). В результате выполнения приложения график последовательно перемещается в обе стороны.

Отличие от простого рисования графиков заключается в том, что вместо функции show() вызывается функция draw(), после которой нужно дать возможность matplotlib обработать внутренние события, в том числе и для отображения графиков. Для этого используется вызов plt.gcf().canvas.flush_events().

Функция gcf() возвращает экземпляр типа Figure, который отвечает за окно графика. После завершения цикла с помощью функции ioff() выключается интерактивный режим, а затем вызывается функция show(), чтобы показать пользователю окно с последним кадром анимации.

Этот пример на каждом шаге анимации он удаляет все объекты кривых с графика, а затем создаетновую кривую.

■ Оптимизация работы приложения

Процесс обновления графика можно оптимизировать, если создать объект, представляющий единственную кривую с графиком, для которой затем изменять данные.

Далее в примере для создания графиков применяется объектно-ориентированный подход.

```
import time
import matplotlib.pyplot as plt
import numpy
# ======================================================================
def gaussian(x, delay, sigma):
    """
    Функция, график которой будет отображаться в процессе анимации
    """
    return numpy.exp(-((x - delay) / sigma) ** 2)
# ======================================================================
def gaussmake(data, delay, sigma, line, fig):
    y = gaussian(data, delay, sigma)
    # Обновить данные на графике
    # (старые данные заменить на новые)
    line.set_ydata(y)
    try:
        # !!! Вызовы для обновления графика
        fig.canvas.draw()
        fig.canvas.flush_events()
        return line
    except:
        return None
# ======================================================================
if __name__ == '__main__':
```

```
# Параметры отображаемой функции
maxSize = 200
sigma = 10.0
# Диапазон точек для расчета графика функции
x = numpy.arange(0, maxSize)
# Значения графика функции
y = numpy.zeros(maxSize)
# !!! Включить интерактивный режим для анимации
plt.ion()
# Создание окна и области рисования для графика. Здесь деления
# на осях и сетка устанавливаются по другому ====================
fig, ax = plt.subplots()
# ax.grid() # можно и так: сначала сетка, затем интервалы
# Установка отображаемых интервалов по осям
ax.set_xlim(0, maxSize)
ax.set_ylim(-1.1, 1.1)
ax.grid()   # установка сетки
# Отобразить график фукнции в начальный момент
# времени (возвращается кортеж линий)
line, = ax.plot(x, y)
factor = 1.0
for i in range(0, 25, 1):
# У функции gaussian будет меняться параметр delay (задержка)
    if factor > 0.0:
        for delay in numpy.arange(-50.0, 200.0, factor):
            line = gaussmake(x, delay, sigma, line, fig)
            if line == None:
                exit()
        # Задержка перед следующим обновлением
        time.sleep(0.01)
        factor *= -1
    if factor < 0:
        for delay in numpy.arange(200.0, -50.0, factor):
            line = gaussmake(x, delay, sigma, line, fig)
            if line == None:
                exit()
        # Задержка перед следующим обновлением
        time.sleep(0.01)
        factor *= -1
# Отключить интерактивный режим по завершению анимации
plt.ioff()
plt.show()
```

Сначала с помощью функции subplots() создается окно с графиком. Эта функция возвращает объекты Figure (окно графика) и Axes (область рисования в окне графика). Затем с помощью метода plot() класса Axes создается график. Этот метод возвращает список объектов Line2D. Поскольку известно, что добавляется только один график, то первый элемент сразу распаковывается в переменную line. В цикле вместо очистки графика и добавления новой кривой используется метод set_ydata() класса Line2D. Этот метод обновляет данные для отображаемой кривой. После этого с помощь методов draw() и flush_events() обновляется график.

■ Применение класса Animation

Для упрощения создания анимаций в Matplotlib применяются два специальных класса, производные от базового класса matplotlib.animation.Animation:

- matplotlib.animation.FuncAnimation,
- matplotlib.animation.ArtistAnimation.

Имеется также промежуточный класс matplotlib.animation.TimedAnimation.

В документации к Matplotlib есть такая схема наследования:

ArtistAnimation

Animation TimedAnimation

FuncAnimation

FuncAnimation используется, если нужно последовательно рассчитывать и обновлять график.

ArtistAnimation используется, если все кривые (или другие анимированные объекты) рассчитываются заранее, а затем в каждом кадре последовательно сменяют друг друга

■ Создание анимации с помощью класса FuncAnimation

Идея использования класса FuncAnimation состоит в том, что конструктору этого класса помимо других параметров, о которых будет сказано позднее, передаются экземпляр класса matplotlib.figure.Figure (он отвечает за окно с графиком) и функция, которая будет вызываться для каждого кадра анимации. Эта функция должна производить какие-то вычисления и возвращать список объектов, которые изменились в новом кадре. Эта функция должна возвращать список объектов, производных от базового класса matplotlib.artist.Artist. Все объекты на графике являются производными от этого класса.

■ Пример использования класса FuncAnimation

Функция main_func() принимает как минимум один параметр frame, который будет меняться от кадра к кадру. В данном примере это смещение функции Гаусса по оси X. Эта функция передается в качестве ВТОРОГО параметра конструктору класса FuncAnimation. Остальные далее перечисленные параметры являются необязательными:

- frames - задает изменяемую последовательность, каждый элемент которой для каждого кадра передается в функцию создания кадра. Этот параметр может быть:
  списком любых объектов;
  итератором;
  целым числом (это равносильно значению range(N));
  функцией-генератором;
  None (равносильно передаче в качестве параметра itertools.count).
- fargs - кортеж с дополнительными параметрами, которые передаются в функцию создания кадра. В данном примере, чтобы функция main_func() не использовала глобальные переменные (как это часто делают в примерах в документации), ей передаются те параметры, которые ей понадобятся для расчета и обновления кривой. Если не требуется передавать дополнительные параметры, этот параметр можно опустить, что будет равносильно передаче значения None.
- interval - задержка между кадрами в миллисекундах.
- blit - использовать ли буферизацию для уменьшения моргания графика при обновлении.
- repeat - если этот параметр равен True, то анимация начнется заново после достижения конца последовательности frames.

Кроме перечисленных параметров есть еще и другие необязательные параметры, которые позволяют изменять параметры кэширования данных и задержку между перезапуском анимации (параметр repeat_delay), если значение repeat равно True.

Результат этого примера не отличается от предыдущих.

```
# ====================================================================
#import time
import matplotlib.pyplot as plt
```

```
from matplotlib.animation import FuncAnimation
import numpy
# ======================================================================
def gaussian(x, delay, sigma):
    """
    Функция, график которой будет отображаться в процессе анимации.
    """
    return numpy.exp(-((x - delay) / sigma) ** 2)
# ======================================================================
# Функция, вызываемая для каждого кадра ================================
def main_func(frame, line, x, sigma):
    """
    frame - параметр, который изменяется от кадра к кадру.
    line - кривая, для которой изменяются данные.
    x - список точек по оси X, для которых рассчитывается функция Гаусса.
    sigma - отвечает за ширину функции Гаусса.
    """
    y = gaussian(x, frame, sigma)   # Обновить данные на графике
                                    # (старые данные заменить на новые)
    line.set_ydata(y)
    return [line]
# ======================================================================
if __name__ == '__main__':
    # Параметры отображаемой функции ===================================
    maxSize = 200
    sigma = 10.0
    # Диапазон точек для расчета графика функции
    x = numpy.arange(maxSize)
    # Значения графика функции
    y = numpy.zeros(maxSize)
    # ==================================================================
    # Создание окна для графика ========================================
    fig, ax = plt.subplots()
    ax.grid()
    # Установка отображаемых интервалов по осям
    ax.set_xlim(0, maxSize)
    ax.set_ylim(-1.1, 1.1)
    # ==================================================================
    # Создание линии, которая будет анимироваться
    line, = ax.plot(x, y)
    # !!! Параметр, который будет меняться от кадра к кадру
    frames = numpy.arange(-50.0, 200.0, 1.0)
    # !!! Задержка между кадрами в мс
    interval = 30
    # !!! Использовать ли буферизацию для устранения мерцания
    blit = True
    # !!! Будет ли анимация циклической
    repeat = True
    # !!! Создание анимации
    animation = FuncAnimation(
            fig,                # Figure (это ссылка на глобальную fig)
                                # Объект рисунка, используемый для получения
                                # необходимых событий, таких как
                                # рисование или изменение размера.
            func=main_func,     # Обновить данные на графике
            frames=frames,      # Источник данных для передачи
                                # в func - функцию (main_func),
                                # которая вызывается в каждом кадре
                                # анимации.
                                # numpy.arrange() используется
                                # для генерации последовательности
                                # чисел в линейном пространстве с
                                # одинаковым размером шага.
            fargs=(line, x, sigma),# fargs: tuple or None (optional)
                                   # Additional arguments to pass to
                                   # each call to func.
```

```
            interval=interval,      # Временной интервал между кадрами
            blit=blit,
            repeat=repeat
    )
    plt.show()
```

■ Создание анимации с помощью класса ArtistAnimation

При использовании класса matplotlib.animation.ArtistAnimation нужно заранее создать список списков объектов для каждого кадра. При этом длина списка первого уровня соответствует количеству кадров, а каждый элемент этого списка - список объектов, которые будут показаны в данном кадре.

При смене кадра объекты, предназначенные для других кадров скрываются.

В примере создается список списков frames, каждый элемент которого - список из одного экземпляра класса matplotlib.lines.Line2D, поскольку от кадра к кадру меняется только один объект на графике.

■ *Применение класса ArtistAnimation*

Конструктор ArtistAnimation кроме экземпляра класса Figure и списка объектов для кадров принимает параметры, аналогичные параметрам класса FuncAnimation: interval, repeat, repeat_delay и blit.

Пример выглядит практически так же, как и предыдущие за исключением того, что цвет линии графика будет отличаться. Это связано с тем, что голубой цвет, заданный параметром '-b' метода plot() отличается от цвета графика по умолчанию.

Но если этот параметр не указывать, каждая новая линия будет создаваться с новым цветом и при выполнении анимации цвет линии будет меняться.

```
#import time
import matplotlib.pyplot as plt
from matplotlib.animation import ArtistAnimation
import numpy
def gaussian(x, delay, sigma):
    """
    Функция, график которой будет отображаться процессе анимации
    """
    return numpy.exp(-((x - delay) / sigma) ** 2)
if __name__ == '__main__':
    # Параметры отображаемой функции  ====================================
    maxSize = 200
    sigma = 10.0
    # Диапазон точек для расчета графика функции
    x = numpy.arange(maxSize)
    # Значения графика функции
    y = numpy.zeros(maxSize)
    # ====================================================================
    # Создание окна для графика ==========================================
    fig, ax = plt.subplots()
    ax.grid()
    # Установка отображаемых интервалов по осям
    ax.set_xlim(0, maxSize)
    ax.set_ylim(-1.1, 1.1)
    # ====================================================================
```

```
# !!! Создание списка линий, которые будут последовательно  ==========
# переключаться и предъявляться при изменении номера кадра
frames = []
for delay in numpy.arange(-50.0, 200.0, 1.0):
    y = gaussian(x, delay, sigma)
    # список рисунков линий ========================================
    # голубой цвет линии задаётся параметром '-b'
    line, = ax.plot(x, y, '-b')
    # !!! Поскольку на каждом кадре может меняться несколько объектов,
    # каждый элемент списка - это список изменяемых объектов (?????)
    frames.append([line])
# ============================================================================
# Задержка между кадрами в мс
interval = 30
# Использовать ли буферизацию для устранения мерцания
blit = True
# Будет ли анимация циклической
repeat = True
# !!! Создание анимации
animation = ArtistAnimation(
        fig,                # Figure (это ссылка на глобальную fig)
                            # Объект рисунка, используемый для получения
                            # необходимых событий, таких как
                            # рисование или изменение размера.
        frames,             # Источник данных - список линий, которые
                            # будут последовательно переключаться и
                            # предъявляться при изменении номера кадра
        interval=interval,  # Временной интервал между кадрами
        blit=blit,          # Использовать буферизацию
                            # для устранения мерцания
        repeat=repeat
)
plt.show()
```

■ Цвет как средство отображения

Цвет - это один из самых распространённых способов отображения информации. Во многом презентабельность и читаемость рисунка зависит от подобранных цветов. В научной графике, как правило, уделяется достаточно мало внимания подборке сочетания цветов, важна однозначность идентификации по цвету.

Matplotlib предоставляет широкие возможности для работы с цветом. Существует множество предустановленных цветов с соответствующими аббревиатурами:

- Красный - 'red', 'r', (1.0, 0.0, 0.0);
- Оранженвый - 'orange';
- Жёлтый - 'yellow', 'y', (0.75, 0.75, 0);
- Зелёный - 'green', 'g', (0.0, 0.5, 0.0);
- Голубой - 'cyan', 'c', (0.0, 0.75, 0.75);
- Синий - 'blue', 'b', (0.0, 0.0, 1.0);
- Фиолетовый - 'magenta', 'm', (0.75, 0, 0.75);
- Чёрный - 'black', 'k', (0.0, 0.0, 0.0);
- Белый -'white', 'w', (1.0, 1.0, 1.0).

Цветовую гамму можно разнообразить за счёт различной степени прозрачности, которая обеспечивается соответствующим значением параметра alpha. Но для задания определённого цвета требуется пользовательская настройка. Matplotlib может отображать цвета, заданные в различных форматах: как в формате RGB, так и в HEX. Существуют различные таблицы цветов, в

которых представленным оттенкам соответствует код в RGB или HEX формате. Ими удобно пользоваться при составлении небольших пользовательских цветовых палитр.

■ Способы задания цветов. RGB и HEX

Список способов задания цвета в Matplotlib:

- С помощью однобуквенной строки (например, 'g').
- С помощью словесного описания цвета (например, 'goldenrod').
- С помощью словесного описания цвета из таблицы xkcd (например, 'xkcd:moss green').
- С помощью указания компонент цвета в формате #RRGGBB (например, '#31D115').
- С помощью указания компонент цвета в формате #RRGGBBTT (например, '#31D1155C'). Последняя пара символов в строке определяет степень прозрачности заданного цвета (transparency).
- С помощью указания компонент цвета в формате #RGB (например, '#AF5').
- С помощью указания компонент цвета в виде кортежа или списка трех чисел в диапазоне 0.0 - 1.0 (например, (0.5, 0.2, 0.3)).
- С помощью указания компонент цвета и альфа-канала в виде кортежа или списка четырех чисел в диапазоне 0.0 - 1.0 (например, (0.5, 0.2, 0.3, 0.8)).

Для задания серого цвета можно применить строку с числом с плавающей точкой в диапазоне [0.0 - 1.0] (например, '0.3').

Цвет в формате RGB представляет собой триплет или кортеж, состоящий из трёх целых значений от 0 до 255 для красного, зелёного и синего цветов. Различные сочетания этих трёх значений позволяет получить огромное количество цветов и оттенков.

В matplotlib цвета rgb формата задаются в относительных единицах, в диапазоне от [0, 1]. Если необходимый цвет задан в диапазоне [0, 255], то значение нужно поделить на 255.

Цвет, представленный в формате HEX, передаётся в виде шестнадцатеричной html строки (символ # перед шестизначным значением обязателен).

```
# ========================================================================

import matplotlib.pyplot as plt
import numpy as np
# ========================================================================
N = 10
x = np.arange(1, N+1, 1)
y = 20.*np.random.rand(N)
# ========================================================================
fig = plt.figure()     # matplotlib.pyplot ================================
ax = fig.add_subplot(111)   # добавление области рисования ax
# Цвет в формате RGB: триплет или кортеж, состоящий из трёх ЦЕЛЫХ значений
# от 0 до 255 для красного, зелёного и синего цветов
x_rgb = np.array([204, 255, 51]) / 725.
ax.bar(x, y, color=x_rgb, alpha=0.75, align='center')
x_hex = '#660099' # символ '#' перед строкой со значением цвета
ax.plot(x, y, color=x_hex)     # фиолетовый график
ax.set_xticks(x)                          # установка делений на оси OX
ax.set_xlim(np.min(x)-1, np.max(x)+1)    # ограничение области изменения
                                          # по оси OX
ax.grid(True)
plt.show()
# ========================================================================
```

Для создания множества областей для рисования удобно не просто добавлять их на рисунок последовательно, по одному, а всего одной строчкой разбить форму на несколько областей рисования.

Это делается методом plt.subplots(), и при его вызове надо указать количество строк и столбцов создаваемой таблицы областей, каждая из ячеек которой и есть объект-экземпляр subplots.

Метод возвращает объект типа figure и массив из созданных subplots. Их можно перебрать в цикле, но лучше применить перебор из списка fig.axes, куда автоматически добавляются все области рисования текущего рисунка fig.

В следующем примере создаются 9 пар прямоугольников, которые отображаются в 9 областях рисования. "нижний" прямоугольник в паре всегда чёрный. "верхний" прямоугольник рисуется поверх "нижнего" чёрного прямоугольника.

В каждой паре верхний прямоугольник обладает различной степенью прозрачности, которая определяется последней парой символов в шестнадцатеричной html строке определения цвета в формате #RRGGBBTT.

Строки определения цвета "верхних" прямоугольников в формате #RRGGBBTT различаются последними двумя символами, которые определяют прозрачность цвета.

```
colors = [
        '#33DD1100',    # полностью прозрачный цвет (невидимый)
        '#33DD1111',
        '#33DD1133',
        '#33DD1155',
        '#33DD1177',
        '#33DD1199',
        '#33DD11AA',
        '#33DD11CC',
        '#33DD11FF'     # полностью непрозрачный цвет
       ]
```

Таким образом, верхний прямоугольник может быть как абсолютно прозрачным и нижний прямоугольник будет полностью просвечивать через абсолютно прозрачный верхний прямоугольник, так и абсолютно непрозрачным и нижний прямоугольник будет полностью скрываться абсолютно непрозрачным верхним прямоугольником.

```
"""
Для создания множества областей для рисования удобно не просто добавлять
их на рисунок последовательно, по одному, а всего одной строчкой разбить
форму на несколько областей рисования.
Это делается методом plt.subplots(),
и при его вызове надо указать количество строк и столбцов создаваемой
таблицы областей, каждая из ячеек которой и есть объект-экземпляр
subplots.
Метод возвращает объект типа figure и массив из созданных subplots.
Их можно перебрать в цикле, но лучше применить перебор из списка
fig.axes, куда автоматически добавляются все области рисования
```

```
 текущего рисунка fig.
"""

import numpy as np
import matplotlib.lines
import matplotlib.patches
import matplotlib.path
import matplotlib.pyplot as plt
# функция для отрисовки прямоугольников. Отображает прямоугольники
# различной степени прозрачности
def drawRect(ax, color, xlim, ylim):
    plt.xlim(xlim)
    plt.ylim(ylim)
    ax.grid(True)
    # Создаётся "нижний" прямоугольник. Он всегда чёрный.
    rect_back = matplotlib.patches.Rectangle((-1.0, -1.0), 2, 2, color='k')
    ax.add_patch(rect_back)
    # Создаётся "верхний" прямоугольник, который рисуется поверх "нижнего"
    # прямоугольника. Он обладает различной степенью прозрачности, которая
    # определяется последней парой символов в шестнадцатеричной html
    # строке определения цвета в формате HEX
    rect_front = matplotlib.patches.Rectangle((-1.5, -1.5), 3, 3,
color=color)
    ax.add_patch(rect_front)
# ============================================================================
# список кодов цветов. Различаются последними двумя числами, которые =====
# определяют прозрачность цвета. Верхний прямоугольник может быть
# абсолютно прозрачным и нижний прямоугольник просвечивает через абсолютно
# прозрачный верхний прямоугольник. Верхний прямоугольник может быть
# абсолютно непрозрачным и нижний прямоугольник полностью скрывается
# абсолютно непрозрачным верхним прямоугольником. ========================
colors = [
           '#33DD1100',     # полностью прозрачный цвет (невидимый)
           '#33DD1111',
           '#33DD1133',
           '#33DD1155',
           '#33DD1177',
           '#33DD1199',
           '#33DD11AA',
           '#33DD11CC',
           '#33DD11FF'      # полностью непрозрачный цвет
         ]
# ============================================================================
fig, subplots = plt.subplots(nrows=3, ncols=3, sharex=True, sharey=True)
i = 0
for ax in fig.axes:
    drawRect(ax, colors[i], (-2,2), (-2,2))
    ax.text(0.0, 0.0, str(i), color='k')
    i += 1
plt.show()
# ============================================================================
"""
       Способы задания цвета в Matplotlib:

 С помощью однобуквенной строки (например, 'g').
 С помощью словесного описания цвета (например, 'goldenrod').
 С помощью словесного описания цвета из таблицы xkcd
   (например, 'xkcd:moss green').
 С помощью указания компонент цвета в формате #RRGGBB
   (например, '#31D115').
 С помощью указания компонент цвета в формате #RRGGBBTT
   (например, '#31D1155C'). Последняя пара символов в строке определяет
   степень прозрачности заданного цвета (transparency).
 С помощью указания компонент цвета в формате #RGB (например, '#AF5').
 С помощью указания компонент цвета в виде кортежа или списка трех чисел
   в диапазоне 0.0 - 1.0 (например, (0.5, 0.2, 0.3)).
```

```
 С помощью указания компонент цвета и альфа-канала в виде кортежа или
   списка четырех чисел в диапазоне 0.0 - 1.0
   (например, (0.5, 0.2, 0.3, 0.8)).

 Для задания серого цвета можно использовать строку с числом с плавающей
 точкой в диапазоне 0.0 - 1.0 (например, '0.3').

"""
# ============================================================================
```

■ Цветовая палитра colormap

Последовательность или набор цветов образует цветовую палитру colormap. Чаще всего она используется при отрисовке трёхмерных данных. Но и автоматический подбор цветов при добавлении каждого нового экземпляра plot также осуществляется из цветовой палитры по умолчанию.

Для получения текущей цветовой палитры можно воспользоваться методом plt.get_cmap('название палитры'). Список всех предустановленных палитр можно получить с помощью метода plt.cm.datad. В настройках matplotlibrc можно также изменить цветовую палитру с помощью параметра image.cmap. В интерактивном режиме её можно поменять через plt.rcParams['image.cmap']='имя_палитры' или через plt.set_cmap('имя_палитры'). При этом plt.set_cmap(...) позволяет изменить палитру текущего рисунка уже после вызова графических команд.

```
# ============================================================================
import matplotlib.pyplot as plt
import numpy as np
dat = np.random.random(200).reshape(20, 10)  # создаётся матрица значений
# Создаётся список цветовых палитр из словаря
maps = [m for m in plt.cm.datad]
# или так ====================================================================
# maps = []
#
# for i, m in enumerate(plt.cm.datad):
#     maps.append(m)
#     print('%d - %s' % (i, m))
# ============================================================================
print(u'Предустановленные цветовые палитры:', maps)
# ============================================================================
# четыре картинки на одной фигуре: 2 строки, 2 столбца ======================
fig, axes = plt.subplots(
                         nrows=2,
                         ncols=2,
                         sharex=True,
                         sharey=True
                        )
# ============================================================================
cmaplist = plt.cm.datad
# ============================================================================
for ax in fig.axes:
    random_cmap = np.random.choice(maps)
    cf = ax.contourf(dat, cmap=plt.get_cmap(random_cmap))
    ax.set_title('%s colormap' % random_cmap)
    fig.colorbar(cf, ax=ax)
# ============================================================================
plt.suptitle(u'Различные цветовые палитры')  # единый заголовок рисунка
plt.show()
# ============================================================================
```

■ Плавная цветовая палитра

Плавная цветовая палитра представляет собой результат линейной интерполяции между последовательностью цветов, составляющих основу палитры.

Различные методы отображения трёхмерных данных (разные графические команды) по умолчанию используют разные типы цветовых палитр.

Так методы plt.imshow() и plt.pcolor() будут сопровождаться плавной цветовой палитрой:

```
# ======================================================================
import matplotlib.pyplot as plt
import numpy as np
dat = np.random.random(200).reshape(20, 10)  # создаём матрицу значений
# Список цветовых палитр из словаря ====================================
# maps = [m for m in plt.cm.datad]
# или так
maps = []
for m in plt.cm.datad:
    maps.append(m)
# ======================================================================
fig, axes = plt.subplots(
                         nrows=2,
                         ncols=1,
                         sharex=True
                        )
# Про рисунки с несколькими областями рисования ...
cmaplist = plt.cm.datad
for i, ax in enumerate(fig.axes):
    random_cmap = np.random.choice(maps)
    if (i == 0):
        cf = ax.pcolor(dat, cmap=plt.get_cmap(random_cmap))
    else:
        cf = ax.imshow(dat, cmap=plt.get_cmap(random_cmap))
    ax.set_title('%s colormap' % random_cmap)
    fig.colorbar(cf, ax=ax)
plt.suptitle(u'Различные цветовые палитры')
plt.show()
# ======================================================================
```

Для создания пользовательской плавной цветовой палитры нужно определиться с её основными цветами. И если было выбрано следующее сочетание цветов: розовый, синий, зелёный, оранжевый и красный, то в представлении RGB это сочетание означает список следующих значений:

- розовый - rgb(150, 0, 0),
- синий - rgb(0, 0, 255),
- зелёный - rgb(0, 255, 0),
- оранжевый - rgb(255, 150, 0),
- красный - rgb(255, 0, 0).

Matplotlib работает с rgb в относительных единицах, поэтому значения rgb триплетов нужно поделить на 255.

После этого нужно создать "словарь палитры" cdict. Это словарь для каждого из оттенков RGB ('red', 'blue', 'green'), где каждому оттенку приписан список или кортеж (x, y0, y1):

x - определяет положение в палитре в диапазоне [0,1]. Проще представить будущую палитру в виде шкалы от 0 до 1;

y0 - значение цвета в относительных единицах ([0, 255] -> [0, 1]) с одной стороны (слева) от положения x;

y1 - значение цвета ([0, 255] -> [0, 1]) с другой стороны (справа) от положения x.

Между заданными позициями x метод LinearSegmentedColormap линейно интерполирует значения цветов между (x_i - y1_i) и (x_i+1 - y0_i). Обычно y0 и y1 совпадают, но, используя разные значения, можно добиваться более сложных эффектов, например разрывов в палитре.

```
# ======================================================================
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
dat = np.random.random(200).reshape(20,10)   # создаётся матрица значений
xx = np.array([0.5, 0.0, 0.1])*255    # Создаётся список цветовых палитр
print('{0}'.format(xx))
# ======================================================================
# Вариант 1 Ромашка (бело-жёлтый) ======================================
cdict1 = {'red':   ((0.0, 1.0, 1.0),
                    (1.0, 1.0, 1.0)),
          'green': ((0.0, 1.0, 1.0),
                    (1.0, 1.0, 1.0)),
          'blue':  ((0.0, 1.0, 1.0),
                    (1.0, 0.0, 0.0))
         }
cmap1 = mpl.colors.LinearSegmentedColormap('cmap1', cdict1)
# ======================================================================
# Вариант 2 Светофор(красный-жёлтый-зелёный) ===========================
cdict2 = {'red':   ((0.0, 0.0, 0.0),
                    (0.5, 1.0, 1.0),
                    (1.0, 1.0, 1.0)),
          'green': ((0.0, 1.0, 1.0),
                    (0.5, 1.0, 1.0),
                    (1.0, 0.0, 0.0)),
          'blue':  ((0.0, 0.0, 0.0),
                    (0.5, 0.0, 0.0),
                    (1.0, 0.0, 0.0))
         }
cmap2 = mpl.colors.LinearSegmentedColormap('cmap2', cdict2)
# ======================================================================
fig, axes = plt.subplots(
                         nrows=2,
                         ncols=1,
                         sharex=True
                        )
cmaplist = plt.cm.datad
cmaps = [cmap1, cmap2]
for i, ax in enumerate(fig.axes):
    cf = ax.imshow(dat, cmap=cmaps[i])
    fig.colorbar(cf, ax=ax)
plt.suptitle(u'Создание цветовых палитр')
plt.show()
# ======================================================================
```

Позиции цветов x на палитре могут быть неравномерными. Тогда на какие-то цвета будут приходиться большие участки цветовой палитры.

```
# ======================================================================
import matplotlib as mpl
```

```
import matplotlib.pyplot as plt
import numpy as np
# ============================================================================
# розовый [255, 204, 255] -> [1., 0.8, 1.]
# фиолетовый [153, 0, 255] -> [0.6, 0., 1.]
# синий [0, 0, 255] -> [0., 0., 1.]
# красный [255, 0, 0] -> [1., 0., 0.]
# ============================================================================
dat = np.random.random(200).reshape(20, 10)  # создание матрицы значений
cdict1 = {'red':   ((0.0, 1.0, 1.0),   # red in RGB of pink, розовый
                    (0.2, 0.6, 0.6),   # red in RGB of blue синий
                    (0.8, 0.0, 0.0),   # red in RGB of violet фиолетовый
                    (1.0, 1.0, 1.0)),  # red in RGB of red красный
          'green':  ((0.0, 0.8, 0.8),   # green in RGB of pink, розовый
                    (0.2, 0.0, 0.0),   # green in RGB of blue синий
                    (0.8, 0.0, 0.0),   # green in RGB of violet фиолетовый
                    (1.0, 0.0, 0.0)),  # green in RGB of red красный
          'blue':   ((0.0, 1.0, 1.0),   # blue in RGB of pink, розовый
                    (0.2, 1.0, 1.0),   # blue in RGB of blue синий
                    (0.8, 1.0, 1.0),   # blue in RGB of violet фиолетовый
                    (1.0, 0.0, 0.0))   # blue in RGB of red красный
          }
cmap1 = mpl.colors.LinearSegmentedColormap('cmap1', cdict1)
plt.register_cmap(cmap=cmap1)
# ============================================================================
fig = plt.figure()
ax = fig.add_subplot(111)
cmaplist = plt.cm.datad
cf = ax.pcolor(dat, cmap=cmap1)
cbar = fig.colorbar(cf, ax=ax)
plt.suptitle(u'Розово-сине-фиолетово-красная палитра')
plt.show()
# ============================================================================
```

Далее приводится код и пример применения функции, которая преобразует заданный список цветов RGB в относительные единицы и может создавать как равномерную, так и неравномерную цветовую палитры.

```
def make_cmap(colors, position=None, bit=False, debug=False):
    """
    Пример функции, которая преобразует заданный список цветов RGB
    в относительные единицы и создаёт как равномерную,
    так и не равномерную цветовую палитры.
    make_cmap takes a list of tuples which contain RGB values. The RGB
    values may either be in 8-bit [0 to 255] (in which bit must be set to
    True when called) or arithmetic [0 to 1] (default). make_cmap returns
    a cmap with equally spaced colors.
    Arrange your tuples so that the first color is the lowest value for the
    colorbar and the last is the highest.
    position contains values from 0 to 1 to dictate the location of each
    color.
    """

    # импорт модулей возможен из любого места кода ======================
    import sys
    import matplotlib as mpl
    import numpy as np
    bit_rgb = np.linspace(0, 1, 256)
    if position == None:
        position = np.linspace(0, 1, len(colors))
    else:
```

```
            if len(position) != len(colors):
                sys.exit("position length must be the same as colors")
            elif position[0] != 0 or position[-1] != 1:
                sys.exit("position must start with 0 and end with 1")
        if (debug):
            print ('position:', position)
            print ('len colors', len(colors))
        if bit:
            for i in range(len(colors)):
                colors[i] = (bit_rgb[colors[i][0]],
                             bit_rgb[colors[i][1]],
                             bit_rgb[colors[i][2]])
        if (debug):
            print ('colors', colors)
        cdict = {'red': [], 'green': [], 'blue': []}
        for pos, color in zip(position, colors):
            cdict['red'].append((pos, color[0], color[0]))
            cdict['green'].append((pos, color[1], color[1]))
            cdict['blue'].append((pos, color[2], color[2]))
        if (debug):
            print ('red', cdict['red'])
            print ('green', cdict['green'])
            print ('blue', cdict['blue'])
        cmap = mpl.colors.LinearSegmentedColormap('test_colormap', cdict, 256)
        return cmap
### An example of how to use make_cmap
# пример использования функции make_cmap
import matplotlib.pyplot as plt
import numpy as np
import sys
fig = plt.figure()
ax = fig.add_subplot(311)
### Create a list of RGB tuples
# 'r', 'y', 'w', 'g', 'b'
colors = [(255, 0, 0), (255, 255, 0), (255, 255, 255), (0, 157, 0), (0, 0,
255)]
# This example uses the 8-bit RGB Call the function make_cmap
# which returns the colormap
my_cmap = make_cmap(colors, bit=True)
### Use your colormap
plt.pcolor(np.random.rand(25, 50), cmap=my_cmap)
plt.colorbar()
ax = fig.add_subplot(312)
colors = [(1, 1, 1), (0.5, 0, 0)]  # This example uses the arithmetic RGB
### If you are only going to use your colormap once you can take out a step.
# Если планируется использовать цветовую карту только один раз,
# можно сделать шаг.
plt.pcolor(np.random.rand(25, 50), cmap=make_cmap(colors))
plt.colorbar()
ax = fig.add_subplot(313)
colors = [(0.4, 0.2, 0.0), (1, 1, 1), (0, 0.3, 0.4)]
### Create an array or list of positions from 0 to 1.
# создаётся список значений от 0 до 1
position = [0, 0.3, 1]
plt.pcolor(np.random.rand(25, 50), cmap=make_cmap(colors, position=position))
plt.colorbar()
plt.show()
```

■ Дискретная цветовая палитра

При отображении трёхмерных поверхностей в виде плоских карт и 3D графиков плавная цветовая палитра хорошо смотрится. Однако в научной графике часто требуется не столько красота, сколько

простота и однозначность рисунка. Поэтому там, где критично однозначное определение оттенков цвета и значений, следует использовать дискретные палитры.

Дискретная палитра представляет собой набор цветов с чёткой границей между соседними оттенками. Обычно этот набор берётся из одной из непрерывных шкал, но можно задать свою последовательность цветов либо в RGB, либо в HEX форматах.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
dat = np.random.random(200).reshape(20, 10) # создаётся матрица значений
N = 10
# Список цветов в HEX формате
cpool = [ '#bd2309', '#bbb12d', '#1480fa', '#14fa2f', '#000000',
          '#faf214', '#2edfea', '#ea2ec4', '#ea2e40', '#cdcdcd',
          '#577a4d', '#2e46c0', '#f59422', '#219774', '#8086d9' ]
# Создание дискретных цветовых палитр (colormap)
cmap1 = mpl.colors.ListedColormap(
    ['r', 'orange', 'y', 'g', 'c', 'b', 'violet'], 'indexed'
                                  ) # 7 цветов
cmap2 = mpl.colors.ListedColormap(
    cpool[5:5+N], 'indexed'
                                  ) # 10 цветов
fig, axes= plt.subplots(nrows=2, ncols=1, sharex=True)
cmaps = [cmap1, cmap2]
# ============================================================================
for i, ax in enumerate(fig.axes):
    cf = ax.pcolor(dat, cmap=cmaps[i])
    fig.colorbar(cf, ax=ax)
# ============================================================================
fig.suptitle(u'Создание дискретных цветовых палитр')
plt.show()
```

При задании дискретной палитры необходимо обращать внимание на согласованность границ делений цветовой шкалы и границ цветов, иначе анализ рисунка будет серьёзно затруднён. На плавной цветовой шкале это не так заметно, как на дискретной.

Для задания положения делений в методе fig.colorbar() или plt.colorbar() необходимо указать параметр ticks. Число положений делений должно быть на 1 больше числа цветов в палитре (делений на отрезке на одну больше, чем промежутков между ними)!

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
dat = np.random.random(200).reshape(20, 10)  # создаётся матрица значений
N = 10
# Список цветов в HEX формате
cpool = [ '#bd2309', '#bbb12d', '#1480fa', '#14fa2f', '#000000',
          '#faf214', '#2edfea', '#ea2ec4', '#ea2e40', '#cdcdcd',
          '#577a4d', '#2e46c0', '#f59422', '#219774', '#8086d9' ]
# Создание дискретных цветовых палитр (colormap)
cmap1 = mpl.colors.ListedColormap(
        ['r', 'orange', 'y', 'g', 'c', 'b', 'violet'], 'indexed'
                                  )  # 7 цветов
cmap2 = mpl.colors.ListedColormap(
        cpool[5:5 + N], 'indexed'
                                  )  # 10 цветов
fig, axes = plt.subplots(nrows=2, ncols=1, sharex=True)
cmaps = [cmap1, cmap2]
```

```
# ============================================================================
for i, ax in enumerate(fig.axes):
    cf = ax.pcolor(dat, cmap=cmaps[i])
    if (i == 0):
        fig.colorbar(cf, ax=ax, ticks=np.linspace(0, 1, 7 + 1))
    else:
        fig.colorbar(cf, ax=ax)
# ============================================================================
plt.suptitle(u'Создание дискретных цветовых палитр')
plt.show()
```

■ Координатные оси

Есть координатные оси (контейнер Axis). Есть деления (ticks) на координатных осях. Деления неотделимы от координатной оси на которой они находятся. При этом свойства самих делений (цвет, длина, толщина и др.), их подписей (кегль, поворот, цвет шрифта, шрифт и др.), а также связанные с ними линии вспомогательной сетки grid, хранятся в отдельном хранилище-контейнере (контейнер Ticks), а не в контейнере Axis. Деления и оси тесно связаны и одни отдельно от других не имеют смысла. Тем более, что для удобства пользователей разработчики сделали множество методов для работы с делениями из контейнеров более высокого уровня (Axes, Axis).

■ Дополнительная координатная ось

Иногда требуется нанести на один рисунок две величины, которые имеют общие единицы измерения по одной оси, но разные по другой. Это могут быть временные ряды сильно отличающихся по масштабу величин. Или ряды величин, имеющих разные единицы измерения. В таких случаях удобно рисовать дополнительную координатную ось (обычно - ординат).

Такую возможность обеспечивает метод ax.twinx() для оси OX и метод ax.twiny() для оси OY. В результате создаётся ещё одна область рисования, совпадающая по размерам с исходной. На каждой области рисуются соответствующие значения, при этом с помощью пользовательской настройки подписей осей ординат можно добиться, чтобы вспомогательные сетки grid обеих областей совпадали.

```
from matplotlib import rcParams
import matplotlib.pyplot as plt
import numpy as np
# ============================================================================
def fun_0(arg0, arg1):
    x = np.arange(-2*arg0, 2*arg0, arg1)
    y = np.sin(x) * np.cos(x)
    f = np.sin(x) + np.cos(x)
    return x, y, f
# ============================================================================
# ============================================================================
def do_it():
    # ========================================================================
    fig = plt.figure()
    # ========================================================================
    x, y, f = fun_0(np.pi, 0.2)
    # ========================================================================
    ax1 = fig.add_subplot(111)  # Явно задаётся область рисования
    ax1.plot(x, f,  color='green')
    ax1.plot(-0.0, -1.0, label = u'Сумма cos и sin', color='green')
    ax1.set_ylabel(u'Функция 1', color='green')
    ax1.grid(True, color='green')
    ax1.legend(loc=2)
    ax1.set_title(u'Несколько разномасштабных переменных')
                                          # Легенда для всего рисунка fig
    y = y * 333
```

```
    ax2 = ax1.twinx()                       # Создаётся вторая шкала ax2
    ax2.plot(x, y, color='red')
    ax2.plot(0.0, 200.0, color='red')
    ax2.set_ylabel(u'Функция 2', color='red')
    ax2.grid(True, color='red')
# ======================================================================
if __name__ == '__main__':
    do_it()
    plt.show()
```

■ Легенда дополнительной оси

Если график с дополнительной координатной осью, выполнен в чёрно-белом варианте, понять какой график относится к левой шкале, а какой к правой невозможно. В таком случае для идентификации данных используют легенду. Имеются особенности при работе с легендой такого графика. Одна легенда отобразилась для одной координатной оси из двух. Вторая легенда отобразились для второй координатной оси и перезатёрла предыдущую. Для неё можно было указать другое место расположения. Чтобы каждая подпись была в своём месте, нужно воспользоваться методом ax.legend(), а не plt.legend().

Расположение легенды задаётся параметром loc метода legend(), который принимает значения в виде цифр (1-9), начиная от верхнего левого угла и заканчивая нижним правым, и в виде условных обозначений. Значение loc='best' автоматически выберет место на рисунке, где легенда меньше всего будет "портить" график. Также можно убрать рамку вокруг легенды для визуального уменьшения места (frameon=False), занимаемого легендой.

```
from matplotlib import rcParams
import matplotlib.pyplot as plt
import numpy as np
# ======================================================================
def fun_0(arg0, arg1):
    x = np.arange(-2*arg0, 2*arg0, arg1)
    y = np.sin(x) * np.cos(x)
    f = np.sin(x) + np.cos(x)
    return x, y, f
# ======================================================================
# ======================================================================
def do_it():
    # ==================================================================
    fig = plt.figure()
    # ==================================================================
    x, y, f = fun_0(np.pi, 0.2)
    # ==================================================================
    ax1 = fig.add_subplot(111)  # Явно задаётся область рисования
    ax1.plot(x, f, color='green')
    ax1.plot(-5.0, 1.5, label=u'Сумма cos и sin', color='green')
    ax1.set_ylabel(u'Функция 1', color='green')
    ax1.grid(True, color='green')
    ax1.legend(loc=1)
    y = y * 333
    ax2 = ax1.twinx()                       # Создаётся вторая шкала ax2
    ax2.plot(x, y*3, color='red')
    ax2.plot(5.0, 400.0, label=u'Произведение cos и sin', color='red')
    ax2.set_ylabel(u'Функция 2', color='red')
    ax2.grid(True, color='red')
    ax2.legend(loc=2)
    plt.title(u'Несколько разномасштабных переменных')
                                            # Легенда для всего рисунка fig
# ======================================================================
if __name__ == '__main__':
    do_it()
```

```
    plt.show()
```

■ Единая легенда

В случае необходимости можно объединить обе легенды.

```
from matplotlib import rcParams
import matplotlib.pyplot as plt
import numpy as np
# ============================================================================
def fun_0(arg0, arg1):
    x = np.arange(-2*arg0, 2*arg0, arg1)
    y = np.sin(x) * np.cos(x)
    f = np.sin(x) + np.cos(x)
    return x, y, f
# ============================================================================
# ============================================================================
def do_it():
    # ========================================================================
    fig = plt.figure()
    # ========================================================================
    x, y, f = fun_0(np.pi, 0.2)
    # ========================================================================
    ax1 = fig.add_subplot(111)  # Явно задаётся область рисования
    ax1.plot(x, f, color='green')
    line1 = ax1.plot(0, 0, label=u'Сумма cos и sin', color='green')
    ax1.set_ylabel(u'Функция 1', color='green')
    ax1.grid(True, color='green')
    y = y * 333
    ax2 = ax1.twinx()                       # Создаётся вторая шкала ax2
    ax2.plot(x, y*3, color='red')
    line2 = ax2.plot(0, 0, label=u'Произведение cos и sin', color='red')
    ax2.set_ylabel(u'Функция 2', color='red')
    ax2.grid(True, color='red')
    # общая единая легенда ===================================================
    legends = line1 + line2
    labels = [l.get_label() for l in legends]
    ax1.legend(legends, labels, loc=3, frameon=False)
    # ========================================================================
    plt.title(u'Несколько разномасштабных переменных')
                                            # Легенда для всего рисунка fig
# ============================================================================
if __name__ == '__main__':
    do_it()
    plt.show()
```

■ Логарифмические координатные оси

Если ось ординат имеет логарифмическую шкалу, то график функции, которая равна экспоненте от аргумента, будет выглядеть как прямая линия.

Поэтому иногда на координатных осях бывает очень удобно использовать не стандартную равномерную шкалу, а логарифмическую.

Сделать шкалу координатной оси логарифмической можно с помощью методов

plt.xscale('log')/plt.yscale('log')

или для областей рисования -

ax.set_xscale('log')/ax.set_yscale('log').

Методы plt.loglog() и ax.loglog() позволяют выразить обе координатные оси в логарифмических шкалах.

```
import numpyimport matplotlib.pyplot as plt
import numpy as np
def set_data(value):
    x = np.arange(value)
    y = np.exp(x)
    return x, y
def do_it_221(fig, x, y):
    # 1 x и y=x
    ax = fig.add_subplot(221)
    ax.plot(x, x)
    ax.grid(True)
    ax.set_xlabel('x', fontsize=14)
    ax.set_ylabel('y = x', fontsize=14)
def do_it_222(fig, x, y):
    # 2 x и y=exp(x)
    ax = fig.add_subplot(222)
    ax.plot(x, y)
    ax.grid(True)
    ax.set_xlabel('x', fontsize=14)
    ax.set_ylabel('y = exp(x)', fontsize=14)
def do_it_223(fig, x, y):
    # 3 x и y=exp(x). OY с log шкалой
    ax = fig.add_subplot(223)
    ax.set_yscale('log')    # log здесь - натуральный логарифм!
    # для работы с типом axis -> ax.set_yscale('log')
    ax.plot(x, y)
    ax.grid(True)
    ax.set_xlabel('x', fontsize=14)
    ax.set_ylabel('y = exp(x)', fontsize=14)
    ax.set_title(u'OY log шкала', loc='center')
def do_it_224(fig, x, y):
    # 4  x и y=x. OX с log шкалой
    ax = fig.add_subplot(224)
    ax.set_xscale('log')    # log здесь - натуральный логарифм!
    ax.plot(x, x)
    ax.grid(True)
    ax.set_xlabel('x', fontsize=14)
    ax.set_ylabel('y = x', fontsize=14)
    ax.set_title(u'OX log шкала', loc='center')
if __name__ == '__main__':
    fig = plt.figure()
    x,y = set_data(20)
    do_it_221(fig, x, y)
    do_it_222(fig, x, y)
    do_it_223(fig, x, y)
    do_it_224(fig, x, y)
    # Автоматическое форматирование риснука
    plt.tight_layout()
    plt.show()

matplotlib.pyplot as plt
import numpy as np

def set_data(value):
    x = np.arange(value)
    y = np.exp(x)
    return x, y

def do_it_221(fig, x, y):
    # Обычные шкалы
```

```
    ax = fig.add_subplot(211)
    ax.plot(x, y)
    ax.set_xlabel('x', fontsize=14)
    ax.set_ylabel('y = exp(x)', fontsize=14)
    ax.set_title(u'Обычные шкалы')
    ax.grid(True)

def do_it_212(fig, x, y):
    # Log шкалы
    ax = fig.add_subplot(212)
    ax.loglog()
    ax.plot(x, y)
    ax.set_xlabel('x', fontsize=14)
    ax.set_ylabel('y = exp(x)', fontsize=14)
    ax.set_title(u'OY log и OX log шкалы')
    ax.grid(True)

if __name__ == '__main__':
    fig = plt.figure()
    x, y = set_data(50)

    do_it_221(fig, x, y)
    do_it_212(fig, x, y)

    # Автоматическое форматирование рисунка
    plt.tight_layout()
    plt.show()
```

■ Поток (и процесс)

В этом разделе рассматриваются потоки и процессы, их применение, особенно в условиях многопоточности (и мультипроцессности).

■ Понятия и определения

Процесс - это исполняемый экземпляр какой-либо программы. Процесс включает следующие элементы:

- образ машинного кода;
- область памяти, в которую включается исполняемый код, данные процесса (входные и выходные данные);
- стек вызовов и куча (области памяти для хранения динамически создаваемых данных);
- дескрипторы операционной системы (например, файловые дескрипторы);
- информация о состоянии процесса.

Процесс имеет собственное ядро и выделенную ему память, которые НЕ используется совместно с другими процессами. Процесс может клонировать себя, создавая в одном ядре процессора несколько экземпляров этого процесса. В современных операционных системы каждый процесс имеет прямой доступ только к своим собственным ресурсам. В целях стабильности и безопасности, доступ к ресурсам других процессов возможен через межпроцессное взаимодействие (например, посредством файлов, при помощи именованных и неименованных каналов и многих других средств).

В multiprocessing (мультипроцессной) среде процесс - представляет собой также независимую последовательность выполнения вычислений.

Сам процесс может быть разделен на потоки.

Поток (поток выполнения, thread) является наименьшей единицей обработки, исполнение которой может быть назначено ядром операционной системы. Потоки существуют внутри одного процесса и имеют доступ к ресурсам этого процесса. Каждый поток обладет собственным набором регистров и собственным стеком вызова, но доступ к ним также имеют и другие потоки. Это независимая последовательность выполнения каких-то вычислений. В рамках одного ядра процессора поток thread может делить память, а также его процессорное время со всеми другими потоками, которые создаются выполняемой программой в рамках ресурсов, выделенных программе ядру процессора. Таким образом, потоки могут использовать одну и ту же выделенную память.

Когда несколько потоков выполняются одновременно, нельзя предусмотреть порядок, в котором потоки будут обращаться к общим данным. Результат доступа к совместно используемым данным зависит от алгоритма планирования, который решает, какой поток и когда запускать. Если такого алгоритма нет, то возможный результат вычислений может быть не такими как он ожидался.

Например, если объявлена общая переменная

```
a = 2
```

И два потока, thread_one и thread_two, которые выполняют следующие операции:

```
a = 2

# функция 1 потока
def thread_one():
```

```
    global a
    a = a + 2

# функция 2 потока
def thread_two():
    global a
    a = a * 3
```

Если поток thread_one получит доступ к общей переменной

`a`

первым и thread_two вторым, то результат будет 12:

2 + 2 = 4;

4 * 3 = 12.

Если сначала запустится thread_two, а затем thread_one, то будет получен другой результат:

2 * 3 = 6;

6 + 2 = 8.

Таким образом, порядок выполнения операций потоками имеет принципиальное значение. Без алгоритмов планирования доступа потоков к общим данным такие ошибки очень трудно найти и произвести отладку. Кроме того, они, как правило, происходят случайным образом, вызывая беспорядочное и непредсказуемое поведение.

Ещё более худший вариант развития событий, который может произойти без встроенной в Python блокировки потоков GIL может произойти, например, если оба потока начнут читать глобальную переменную

`a`

одновременно. При этом оба потока увидят, что

`a = 2`

, а дальше, в зависимости от того какой поток произведет вычисления последним, значение переменной

`a`

будет равно 4 или 6.

Потоки имеют несколько применений. Первое - ускорение (?) работы программы. Возможно, создание илюзии ускорения, которая достигается за счет параллельного выполнения независимых друг от друга вычислений.

Второе - независимое исполнение операций. Например, если есть приложение с графическим интерфейсом, в котором весь код выполняется в одном потоке. При выполнении какой-нибудь долгой операции (например, копирования файла большого размера) интерфейс приложения перестанет отвечать на действия (запросы) пользователя до тех пор, пока не завершится долгий процесс копирования. В таком случае в один поток помещается работа графического интерфейса, в другой - остальные вычисления. В этом случае поток интерфейса позволит проводить другие операции даже во время выполнения в его потоке операции копирования.

Программы на языке программирования python по умолчанию имеют один основной поток. Этому способствует реализация интерпретатора GIL (Global Interpreter Lock), которая имеет встроенный механизм, который обеспечивает выполнение одного потока в любой момент времени. GIL облегчает реализацию интерпретатора (такую программу проще написать), защищая объекты выполняемой программы от одновременного доступа из нескольких потоков. Поэтому создание нескольких потоков не приведет к их одновременному исполнению на разных ядрах процессора. Однако в выполняющей среде python можно создать больше потоков и позволить переключаться между ними. При этом некоторые модули, как стандартные, так и сторонние, были созданы для освобождения GIL при выполнении тяжелых вычислительных операций (например, сжатия или хеширования). К тому же, GIL всегда свободен при выполнении операций ввода-вывода.

■ Описание разных подходов к параллельным вычислениям в Python

Определяется функция, которая будет использоваться для описания различных вариантов вычислений. В следующих примерах будет использоватся одна и та же функция, называемая master() (ядро потока):

```
def master(n):
  for x in range(1, n):
    for y in range(1, n):
      res = x**y
```

Функция master() представляет собой вложенный цикл, который выполняет возведение в степень. При выполнении математических вычислений можно увидеть загрузку процессора близкую к 100%.

Функция master() будет запускаться по-разному, тем самым демонстрируя различия между обычной однопоточной программой Python, многопоточностью и мультипроцессорностью.

■ Однопоточный режим работы

Каждая программа Python имеет по крайней мере один основной поток. Ниже представлен пример кода для запуска функции master() в одном основном потоке одного ядра процессора, который производит все операции последовательно и будет служить эталоном с точки зрения скорости выполнения:

```
import time

def master(n, i):
    for x in range(1, n):
        for y in range(1, n):
            result = x**y
            print(f"\nЦикл № {i}. Результат {result}\n")

# последовательный запуск функции master
```

```
def sequential(n):
    for i in range(n):
        master(10, i)
    print(f"{n} циклов вычислений закончены")

if __name__ == "__main__":
    start = time.time()
    sequential(50)
    end = time.time()
    print(f"\nОбщее время работы: {end — start}")
```

■ Пример использования потоков с применением библиотеки threading

В следующем примере будет использоваться несколько потоков для выполнения функции master(). Работа с потоками осуществляется при помощи стандартной библиотеки threading. Будет произведено 400 циклов вычислений. Для этого вычисления разделяются на 4 блока по 10 потоков, в каждом из которых будет запущено по 10 циклов. Метод .join() вызывается от имени потока. Он позволяет указать одному выполняемому потоку, что необходимо дождаться завершения другого потока (других потоков). При использовании этого метода поток, от имени которого происходт вызов метода .join() остановится и будет ждать завершения (завершение — это НЕ остановка) других потоков. Метод join() может принимать один аргумент - таймаут в секундах. Если таймаут задан, join() бликирует работу потока на указанное время. Если по истечении времени ожидаемый поток не будет завершен, join() все равно разблокирует работу потока, вызвашего его. Проверить, исполняется ли поток можно с помощью метода is_alive().

```
from threading import Thread
import time
import numpy as np

def master(i, n):
    for x in range(1, n):
        for y in range(1, n):
            result = x**y
            print(f"\nЦикл № {i}. Результат {result}\n")

# запуск функции master на нескольких потоках
def sequential(nthSet,nth ,calc, master):

    for i in range(calc):
        print(f"Запускается поток № {nth} в блоке {nthSet} операция {i}")
        master(i, 10)

    print(f"\n{calc} циклов вычислений в блоке {nthSet} потока {nth}
закончены.")

def threaded(nthSet, nth, calc):
    # nthSet - количество блоков
    # nth - количество потоков в блоке
    # calc - количество операций на поток

    threads = [[], [], [], []]

    #  вычисления делятся на 4 блока,
    #  в блоке 10 потоков
    for thElem in range(0, nthSet):
        for nthElem in range(0, nth):
```

```
                threads[thElem].append(Thread(target=sequential,
args=(thElem,nthElem ,calc, master, )))

    # каждый поток должен быть запущен
    for thElem in range(0, nthSet):
        for nthElem in range(nth):
            print(f'\n--- блок {thElem}, поток {nthElem}: start ! ---')
            threads[thElem][nthElem].start()

    # Ожидание завершения работы для всех потоков
    for thElem in range(0, nthSet):
        for nthElem in range(0, nth):
            threads[thElem][nthElem].join()

if __name__ == "__main__":
    start = time.time()
    #  вычисления разделяются на 4 блока потоков
    #  в каждом из которых по 10 потоков,
    #  которые делают по 20 циклов
    threaded(nthSet=4, nth = 10, calc=20)
    end = time.time()
    print("Общее время работы: ", end — start)
```

Однопоточный режим работы оказывается быстрее, потому что один поток не имеет накладных расходов на создание потоков (в данном случае создается 4 потока) и переключение между ними. Если бы у Python не было GIL, то вычисления функции master() происходили бы быстрее, а общее время выполнения программы стремилось ко времени выполнения однопоточной программы. Причина, по которой многопоточный режим в данном примере не будет работать быстрее однопоточного заключается в вычислениях, связанных с процессором и заключаются в GIL.

Если бы функция master() имел много блокирующих операций, таких как сетевые вызовы или операции с файловой системой, то применение многопоточного режима работы было бы оправдано и (возможно) дало увеличение скорости.

Это утверждение можно проверить смоделировав операции ввода-вывода при помощи функции time.sleep().

■ threading: детали применения

В библиотеке threading представлен класс Thread для создания потока выполнения.

Конструктор класса Thread имеет следующие аргументы:

- group должно быть None; зарезервировано для будующих реализаций Python 3;
- target является исполняемым объектом (по умолчанию равен None, ничего не исполняется);
- name обозначет имя потока (по умолчанию имя генерируется автоматически);
- args - кортеж аргументов для исполняемого объекта;
- kwargs - словарь именованных аргументов для исполняемого объекта;
- daemon равное True обозначет служебный поток (служебные потоки завершаются принудительно при завершении процесса); по умолчанию False.

После того, как объект создан, поток запускается путем вызова метода start(). Задание исполняемого кода в отдельном потоке возможно двумя способами:

- передача исполняемого объекта (функции) в конструктор класса;
- переопределение функции run() в классе-наследнике.

```
import threading
import sys

def thread_job(number):
    print(f'Hello {format(number)}')
    sys.stdout.flush()
#  thread_job на нескольких потоках

def run_threads(count):
    threads = [
        threading.Thread(target=thread_job, args=(i,))
        for i in range(0, count)
    ]
# каждый поток из списка потоков должен быть запущен
    for thread in threads:
        thread.start()
# ожидание исполнения ВСЕХ потоков
    for thread in threads:
        thread.join()

run_threads(5)
print('finish')
```

Выполнение программы в python кончается, когда завершены все неслужебные потоки.

И ещё один пример корректно работающего кода (метод .join здесь не вызывается):

```
from threading import Thread
import time

def thread_job(number):
    time.sleep(2)  # поток "усыпляется" на 2 сек
    print(f'\nHello {number}', flush=True)

def run_threads(count):
    threads = [
    Thread(target=thread_job, args=(i, ))
    for i in range(0, count)
    ]
    # каждый поток должен быть запущен
    for thread in threads:
        thread.start()

run_threads(5)
print('finish')
```

В этом примере строка "finish" печатается раньше строки "Hello 0", т.к. главный поток теперь не ждет завершения работы других потоков. Метод join() используется для блокирования исполнения родительского потока до тех пор, пока созданный поток не завершится. Это требуется в случаях, когда для работы потока-родителя важен результат работы потока-потомка. В этом случае вычисление итогового значения выполняется в главном потоке, но это возможно только после завершения вычислений в побочных потоках. В таком случае главный поток нужно просто приостановить до тех пор, пока не завершатся все побочные потоки.

■ multiprocessing

Библиотека multiprocessing позволяет организовать параллелизм вычислений за счет создания подпроцессов. Так как каждый подпроцесс (процесс) выполняется независимо от других, этот метод параллелизма позволяет избежать проблем с GIL. Предоставляемый библиотекой API библиотеки схож с тем, что есть в threading, хотя есть уникальные вещи. Создание процесса происходит путём создания объекта класса Process. Аргументы конструктора аналогичны аргументам, объявленным в конструкторе Thread. В том числе аргумент daemon позволяет создавать служебные процессы. Эти процессы завершаются вместе с родительским процессом и НЕ могут порождать собственные подпроцессы.

Пример работы с библиотекой:

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('tst',))
    p.start()
    p.join()
```

Каждый процесс имеет свой ID и в этом можно убедиться, запустив код:

```
from multiprocessing import Process
import os


def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Как и всегда, конструкция __name__ == '__main__' нужна для того, чтобы модуль можно было безопасно подключать в другие модули и при этом без ведома программистов не создавались бы новые процессы.

■ Взаимодействие между процессами

multiprocessing предоставляет ТРИ вида межпроцессного обмена данными: очереди, каналы данных (pipe), файлы.

Очереди (класс Queue) аналогичны структуре данных "очередь".

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # выводит "[42, None, 'hello']"
    p.join()
```

Класс Pipe (канал) отвечает за канал обмена данными (по умолчанию, двунаправленный), представленный двумя объектами класса Connection, концами канала. С одним концом канала работает родительский процесс, а с другим концом - подпроцесс.

```
"""Класс Pipe (канал) отвечает за канал обмена данными (по умолчанию,
двунаправленный), представленный двумя объектами класса Connection,
 концами канала. С одним концом канала работает родительский процесс,
с другим концом - подпроцесс. """

from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
# подпроцесс готов посылать по каналу child_conn
    p = Process(target=f, args=(child_conn,))
    p.start() # запуск подпроцесса
# родительский процесс по каналу parent_conn получает procTxt
    procTxt = parent_conn.recv()
    print(procTxt)   # выводит "[42, None, 'hello']"
    p.join()
```

Еще один вид обмена данными может быть достигнут путем записи/чтения обычных файлов. Чтобы исключить одновременную работу двух процессов с одним файлом, в библиотеке есть классы аналогичные threading.

```
from multiprocessing import Process, Lock

def fun(lck, i):
    lck.acquire()
    try:
        print('hello world', i)
    finally:
        lck.release()

if __name__ == '__main__':
    lock = Lock()
    for num in range(10):
        Process(target=fun, args=(lock, num)).start()
```

■ Класс Pool и multiprocessing

Класс Pool представляет механизм распараллеливания выполняемых функций, распределения входных данных по процессам и т.д.

Наиболее важные методы класса: Pool.apply, Pool.map, Pool.apply_async, Pool.map_async.

Пример работы класса Pool:

```
from multiprocessing import Pool

def cube(x):
    return x**3

if __name__ == "__main__":
    pool = Pool(processes=4)  # создаётся пул из 4 процессов
    # в apply можно передать несколько аргументов
    results = [pool.apply(cube, args=(x,)) for x in range(1,7)]
    # числа раскидываются от 1 до 7 по 4 процессам
    print(results)

    pool = Pool(processes=4)
    # то же самое, но с map. разбивает итерируемый объект (range(1,7))
    # на chunks и раскидывает аргументы по процессам
    results = pool.map(cube, range(1, 7))
    print(results)
```

map, apply - блокирующие вызовы. Главная программа будет заблокирована, пока процесс не выполнит работу.

map_async, apply_async - неблокирующие. При их вызове, они сразу возвращают управление в главную программу (возвращают ApplyResult как результат). Метод get() объекта ApplyResult блокирует основной поток, пока функция не будет выполнена.

```
pool = mp.Pool(processes=4)
results = [pool.apply_async(cube, args=(x,)) for x in range(1,7)]
output = [p.get() for p in results]
print(output)
```

■ Применение многопоточности: взгляд на звёзды

От простой модели из двух целевых функций к объектно ориентированному приложению

■ Первый этап разработки приложения

Телескопы пытаются рассмотреть звёзды целевой функцией starSearching.

```
import random
from collections import defaultdict
from threading import Thread


#                           звёзды и телескопы
STARS = (None,
         'жёлтыйКарлик',
         'красныйГигант',
```

```
        'двойнаяЗвезда',
        'новаяЗвезда', )

SCOPES = ('rScope0', 'rScope1', 'rScope2')

#           Обработка результатов поиска звёзд. Работают 2 телескопа
def starSearching(scopename, attempts):
    catch = defaultdict(int)    # счётчик результатов словарь (key, value)
                                # телескопа scopename для учёта найденных звёзд
                                # (какая звезда, сколько таких звёзд)

    for attempt in range(attempts):
        print(f'\n{scopename} {attempt} попытка ... идёт поиск', flush=True)

# таким образом отрабатывается пауза (время работы для телескопа scopename)
        _ = 5 ** (random.randint(25, 75) * 10000)
# ============================================================================

        star = random.choice(STARS)     # телескоп отработал положенный
                                        # интервал времени. И что то нашёл?

        # разбор результатов поиска:
        if star is None:
            print(f'{scopename} {attempt} пустая попытка', flush=True)
        else:
            print(f'{scopename} {attempt} объект найден', flush=True)
            catch[star] += 1  # увеличен счётчик результатов телескопа
                              # scopename

    print(f'\nна телескопе {scopename} найдено:', flush=True)
    for star, count in catch.items():
        print(f'   {star}, {count}     ', flush=True)
    print()

def doIt():
    # 0-й телескоп (rScope0) - в поток и запуск из потока.
    # kwargs - аргументы функции target (целевой функции starSearching)
    thread = Thread(target=starSearching, kwargs=dict(scopename=SCOPES[0],
attempts=10))
    thread.start()

    #  1-й телескоп. Прямой запуск и присоединение к потоку
    starSearching(scopename=SCOPES[1], attempts=10)

    thread.join() # ожидание окончания работы потоков

if __name__ == '__main__':
    doIt()
```

■ Второй этап разработки приложения

Здесь не объявлены классы.

```
import random
from collections import defaultdict
from threading import Thread
```

```
#             звёзды и телескопы
STARS = (None,
        'жёлтыйКарлик',
        'красныйГигант',
        'двойнаяЗвезда',
        'новаяЗвезда', )

SCOPES = ('rScope0', 'rScope1', 'rScope2')

def starSearching(scopename, attempts, catch):
    #             телескоп
    #                         количество попыток
    #                                   счётчик звёзд
    #                                  (значение для возврата из потока)


    for attempt in range(attempts):
        print(f'{scopename} {attempt} попытка ... идёт поиск', flush=True)

 # таким образом отрабатывается пауза (время работы для телескопа scopename)
        _ = 5 ** (random.randint(25, 75) * 10000)
        #
=========================================================================

        star = random.choice(STARS)      # телескоп отработал положенный
                                         # интервал времени. И что то нашёл?

        # разбор результатов поиска:
        if star is None:
            print(f'\n{scopename} {attempt} пустая попытка\n', flush=True)
        else:
            print(f'\n{scopename} {attempt} объект найден\n', flush=True)
            catch[star] += 1  # увеличен счётчик результатов
                              # телескопа scopename


def doIt(scopes, starList):

    # 0-й телескоп - в поток и запуск из потока.
    #                                    kwargs - аргументы функции target
    rScope0_catch = defaultdict(int)     # счётчик результатов словарь
                                         # (key, value)
                                         # телескопа scopename для учёта
                                         # найденных звёзд
                                         # (какая звезда, сколько таких звёзд)
    starList.append(rScope0_catch)
    # это запуск потока
    #               starSearching
    #                                  аргументы (scopename, attempts, catch)
    thread = Thread(target=starSearching, kwargs=dict(scopename=scopes[0],
                                                      attempts=10,
                                                   catch=starList[0]))

    thread.start()

    #  1-й телескоп. Это прямой запуск  и присоединение к потоку
    rScope1_catch = defaultdict(int)  # счётчик результатов
                                      # словарь (key, value)
```

```
                                        # телескопа scopename для учёта
                                        # найденных звёзд
                                        # (какая звезда, сколько таких звёзд)
    starList.append(rScope1_catch)
    starSearching(scopename=scopes[1], attempts=10, catch = starList[1])

    thread.join() # ожидание окончания работы потоков


if __name__ == '__main__':
    lst = []
    doIt(scopes = SCOPES, starList = lst)

    # итоги работы телескопов
    # nScope = 0
    # for name, catch in ((SCOPES[0], lst[0]), (SCOPES[1], lst[1])):
    #     print(f'\nна телескопе {SCOPES[nScope]} найдено:', flush=True)
    #     for star, count in catch.items():
    #         print(f'   {star}, {count}     ', flush=True)
    #     nScope += 1
    #     print()

    nScope = 0
    for nScope in range(0, 2):
        catch = lst[nScope]
        print(f'\nна телескопе {SCOPES[nScope]} найдено:', flush=True)
        for star, count in catch.items():
            print(f'   {star}, {count}     ', flush=True)
        nScope += 1
        print()
```

■ Третий этап разработки приложения

```
import random
from collections import defaultdict
from threading import Thread

#                           звёзды и телескопы
STARS = (None,
        'жёлтыйКарлик',
        'красныйГигант',
        'двойнаяЗвезда',
        'новаяЗвезда', )
SCOPES = ('rScope0', 'rScope1', 'rScope2')

#=============================================================================
class rScope():

    def __init__(self, scopename, attempts):
        self.scopename = scopename
        self.attempts = attempts
        self.stars = defaultdict(int)  # счётчик результатов словарь
                                       # (key, value)
                                       # телескопа scopename для учёта
                                       # найденных звёзд
                                       # (какая звезда, сколько таких звёзд)

    def run(self):
```

```
        for attempt in range(0, self.attempts):
            print(f'\n{self.scopename} {attempt} попытка . идёт поиск',
                                                                    flush=True)

# таким образом отрабатывается пауза (время работы для телескопа
#                                                                 self.scopename)
            _ = 5 ** (random.randint(25, 75) * 10000)
#=========================================================================

            star = random.choice(STARS)  # телескоп отработал положенный
                                         # интервал времени. И что он нашёл?

            # разбор результатов поиска:
            if star is None:
                print(f'\n{self.scopename} {attempt} объект НЕ найден\n',
                                                                    flush=True)
            else:
                print(f'\n{self.scopename} {attempt} объект найден\n',
                                                                    flush=True)
                self.stars[star] += 1  # увеличен счётчик результатов
                                       # телескопа self.scopename

def doIt(lstScopes, nScopes):

    for sc in range(0, nScopes):
        # формируется список телескопов. У каждого по 10 попыток
        # для звёздного поиска
        lstScopes.append(rScope(SCOPES[sc], attempts=10))

    # это подготовка к запуску потока
    #               run - целевая функция
    #
    # thread = Thread(target=lstScopes[0].run)
    # thread.start()
    #
    # # это непосредственный запуск целевой функции
    # lstScopes[1].run()
    # thread.join()

    threads = []
    # это подготовка к запуску 3-х потоков
    #                          run - целевая функция
    threads.append(Thread(target=lstScopes[0].run))
    threads.append(Thread(target=lstScopes[1].run))
    threads.append(Thread(target=lstScopes[2].run))

    # threads[0].start()
    # threads[1].start()
    # threads[2].start()
    for t in range(0, len(threads)):
        threads[t].start()

    # threads[0].join()
    # threads[1].join()
    # threads[2].join()
    for t in range(0, len(threads)):
        threads[t].join()
```

```
if __name__ == '__main__':
    lstScopes = []
    doIt(lstScopes, nScopes = 3)

    # итоги работы телескопов
    for rScope_catch in lstScopes:
        print(f'\nна телескопе {rScope_catch.scopename} найдено:',
                                                                flush=True)
        for star in rScope_catch.stars:
            print(f'    {star}, {rScope_catch.stars[star]}  ')
        print()
```

■ Четвёртый этап разработки приложения

В приложении учитывается реакция на возможные нештатные ситуации.

```
import random
from collections import defaultdict
from threading import Thread

#                           звёзды и телескопы
STARS = (None,
        'жёлтыйКарлик',
        'красныйГигант',
        'двойнаяЗвезда',
        'новаяЗвезда', )
SCOPES = ('rScope0', 'rScope1', 'rScope2')

#     Отработка ошибок в потоке (вероятные поломки телескопов)
#=============================================================================

class rScope():

    def __init__(self, name, attempts):
        self.thread = Thread(target=self.run)  # поток объявляется элементом
                                               # объекта
                                               # метод run - целевая функция
        self.dice = None
        self.scopename = name
        self.attempts = attempts
        self.stars = defaultdict(int)  # счётчик результатов
                                       # словарь (key, value)
                                       # телескопа scopename для учёта
                                       # найденных звёзд
                                       # (какая звезда, сколько таких звёзд)

# поиск (поток) выделен в отдельный метод.
#                         И в нём могут возникать ошибки (поломки телескопов)
    def _starSearching(self):
        for attempt in range(0, self.attempts):
            print(f'\n{self.scopename} {attempt} попытка ... идёт поиск')

# таким образом отрабатывается пауза
#                                  (время работы для телескопа self.scopename)
            _ = 5 ** (random.randint(25, 75) * 10000)

     # возможная поломка телескопа self.scopename - возбуждается ошибка
```

```
            self.dice = random.randint(0, 10)
            if self.dice == 0:
                raise ValueError(f'{self.scopename} попытка {attempt} ...')

#=============================================================================

            star = random.choice(STARS)  # телескоп отработал положенный
                                         # интервал времени. И что он нашёл?

            # разбор результатов поиска:
            if star is None:
                print(f'\n{self.scopename} {attempt} объект НЕ найден\n',
                                                                    flush=True)
            else:
                print(f'\n{self.scopename} {attempt} объект найден\n',
                                                                    flush=True)
                self.stars[star] += 1  # увеличен счётчик результатов
                                       # телескопа self.scopename

    # перехват ошибки, которая имеет вид исключения, осуществляется в потоке
    def run(self):
        try:
            self._starSearching()
        except Exception as exc:
            print(f'\nНЕШТАТНАЯ СИТУАЦИЯ: {exc}')

def doIt(lstScopes, nScopes):
    # формируется список телескопов. У каждого по 10 попыток
    for sc in range(0, nScopes):
        lstScopes.append(rScope(SCOPES[sc], attempts=10))

    # запуск потоков
    for sc in range(0, len(lstScopes)):
        lstScopes[sc].thread.start()

    # контроль потоков
    for sc in range(0, len(lstScopes)):
        lstScopes[sc].thread.join()

if __name__ == '__main__':
    lstScopes = []
    doIt(lstScopes, nScopes=3)

    # итоги работы телескопов
    for rScope_catch in lstScopes:
        print(f'\nна телескопе {rScope_catch.scopename} найдено:',
flush=True)
        for star in rScope_catch.stars:
            print(f'    {star}, {rScope_catch.stars[star]}  ')
        print()
```

■ Пятый этап разработки приложения

Неудавшийся результат поиска при попытке (а их всего 10, и они могут возникать при поломке телескопа или в результате неточного прицеливания) — тоже результат, который учитывается в данной версии.

```
import random
import threading
from collections import defaultdict
from threading import Thread
from multiprocessing import Queue

#                           звёзды и телескопы
STARS = ('None', 'жёлтыйКарлик', 'красныйГигант', 'двойнаяЗвезда',
'новаяЗвезда', )
SCOPES = ('rScope0', 'rScope1', 'rScope2', 'rScope3', 'rScope4')

nScopes = 5

#                   Отработка ошибок в потоке
#                   Очереди для обмена данными
#=============================================================================
class rScope():

    def __init__(self, name, attempts):
        # поток объявляется элементом объекта
        # метод scopeRun - целевая функция

        self.thread = Thread(target=self.scopeRun)
        self.attempts = attempts
        self.dice = None
        self.scopename = name

        self.stars = {STARS[0]: 0,
                      STARS[1]: 0,
                      STARS[2]: 0,
                      STARS[3]: 0,
                      STARS[4]: 0}

        # прочие варианты объявления словаря
        # self.stars = dict(zip([STARS[0],
        #                        STARS[1],
        #                        STARS[2],
        #                        STARS[3],
        #                        STARS[4]],
        #                       [0, 0, 0, 0, 0]))

        # self.stars = dict()
        # self.stars[STARS[0]] = 0
        # self.stars[STARS[1]] = 0
        # self.stars[STARS[2]] = 0
        # self.stars[STARS[3]] = 0
        # self.stars[STARS[4]] = 0
                                    # счётчик результатов словарь (key, value)
                                # телескопа scopename для учёта найденных звёзд
                                        # (какая звезда, сколько таких звёзд)

# перехват ошибки, которая имеет вид исключения, осуществляется в потоке
    def scopeRun(self):
        try:
            return self._starSearching()
        except Exception as exc:
            print(f'\nНЕШТАТНАЯ СИТУАЦИЯ: {exc}')
            return 'ERROR'
```

```
# поиск (поток) выделен в отдельный метод. И в нём могут возникать ошибки
# (поломки телескопов)
    def _starSearching(self):
        if self.attempts > 0:
            print(f'\n{self.scopename} {self.attempts} попытка ... идёт
поиск')

# таким образом отрабатывается пауза
# (время работы для телескопа self.scopename)
            _ = 5 ** (random.randint(25, 75) * 10)

# возможная поломка телескопа self.scopename - возбуждается ошибка ========
            self.dice = random.randint(0, 10)
            if self.dice == 0:
                raise ValueError(f'{self.scopename} попытка
{self.attempts} ...')

#============================================================================

            star = random.choice(STARS)  # телескоп отработал положенный
                                         # интервал времени. И что он нашёл?

            # разбор результатов поиска:
            if star == 'None':
                print(f'\n{self.scopename} {self.attempts} объект НЕ
найден\n', flush=True)
            elif star == 'ERROR':
                print(f'\n{self.scopename} {self.attempts} ERROR\n',
flush=True)
            else:
                print(f'\n{self.scopename} {self.attempts} {self.stars[star]}
объект найден\n', flush=True)
                self.stars[star] += 1  # увеличен счётчик результатов
                                       # телескопа self.scopename
        return star

class rRoll(threading.Thread):

    def __init__(self, lscScopes):
        super().__init__()
        self.scopes = lscScopes
        self.nStars = 0
        self.receiver = Queue()
        self.stars = defaultdict(int)  # счётчик результатов словарь
                                       # (key, value)
                                       # телескопа scopename для учёта
                                       # найденных звёзд
                                       # (какая звезда, сколько таких звёзд)

    def startScopes(self):
        # старт телескопов из списка телескопов
        print(f'  Телескопы - в работу\n', flush=True)
        for nsc in range(0, len(self.scopes)):
            print(f'{self.scopes[nsc].scopename} стартует...')
            self.scopes[nsc].thread.start()

    def goScope(self):
```

```
        nsc = 0

        while self.scopes[nsc].attempts > 0:

            star = self.scopes[nsc].scopeRun()

            if star == 'None':
                print(f'>>> {self.scopes[nsc].scopename}
{self.scopes[nsc].attempts} пусто ', flush=True)
                self.scopes[nsc].stars[star] += 1
            elif star == 'ERROR':
                pass
            else:
                print(f'>>> {self.scopes[nsc].scopename}
{self.scopes[nsc].attempts} {self.scopes[nsc].stars[star]}',

flush=True)
                self.enroll()


            self.scopes[nsc].attempts -= 1
# попытками телескопов управляет объект rRoll


            for n in range(0, len(self.scopes)):
                self.scopes[n].thread.join()    # проверка потока

            nsc += 1
            if nsc == len(self.scopes):
                nsc = 0

            print(f'---***---', flush=True)


    def enroll(self):
        self.nStars += 1

if __name__ == '__main__':
    # формируется список телескопов. У каждого по 10 попыток потока.
    lstScopes = []
    for sc in range(0, nScopes):
        lstScopes.append(rScope(SCOPES[sc], attempts=10))

    roll = rRoll(lstScopes)
    roll.startScopes()
    roll.goScope()


    # итоги работы телескопов
    print(f'всего звёзд {roll.nStars}', flush=True)
    for rscope in roll.scopes:
        print(f'\nтелескоп {rscope.scopename} нашёл:', flush=True)
        for star in rscope.stars:
            print(f'  {rscope.scopename}: {star}, {rscope.stars[star]}  ')
        print()
```

Общее представление об асинхронном вводе-выводе

Асинхронный ввод-вывод с модулем asyncio также даёт возможность в одной программе работать над несколькими вычислениями одновременно. Асинхронный ввод-вывод не является ни потоковым (threading), ни многопроцессорным (multiprocessing). Это однопоточная однопроцессная парадигма не относится к параллельным вычислениям. Такого поведения в Python можно добиться несколькими способами:

- используя многопоточность threading, позволяя нескольким потокам работать по очереди.
- используя несколько ядер процессора multiprocessing. Делать сразу несколько вычислений, используя несколько ядер процессора. Это и называется параллелизмом.
- используя асинхронный ввод-вывод с модулем asyncio. При этом, запуская какую то задачу, вместо ожидания ответа от сетевого подключения или от операций чтения/записи, можно продолжать делать другие вычисления.

Следует использовать модули threading или asyncio для программ, связанных с вводом-выводом, чтобы повысить производительность.

Модуль multiprocessing используется для решения проблем, связанных с операциями ЦП. Этот модуль использует весь потенциал всех ядер процессора.

Оглавление

Учебное издание

Марченко Антон Леонардович

Python

Большая книга примеров

Электронное издание сетевого распространения

Художественное оформление *Ю. Н. Симоненко*
Текст публикуется в авторской редакции

Макет утвержден 25.05.2023. Формат 60×90/8. Усл. печ. л. 44,0.
Изд. № 12433.

Издательство Московского университета. 119991, Москва, ГСП-1,
Ленинские горы, д. 1, стр. 15 (ул. Академика Хохлова, 11).
Тел.: (495) 939-32-91; e-mail: secretary@msupress.com
Отдел реализации. Тел.: (495) 939-33-23; e-mail: zakaz@msupress.com
Сайт Издательства МГУ: http://msupress.com

ISBN 978-5-19-011853-7

9 785190 118537