Python: основы и применения

Конспект по курсу на stepik с дополнениями.

created by araz abdyev

tg @aabdyev

☐ Организуй квиз когда закончишь конспект по темам из этого конспекта

☐ Обязательно выложи свой конспект в гитхаб хотя бы на время, когда будешь проходить собес в яшку или еще куда на разраба

- Можно указать это пет-проектом типа: сделал свой полезный конспект и квиз по нему

Предисловие

Комментарии к программному коду написаны на английском языке. Так, на момент написания этого конспекта я “приучаю” себя читать и писать комментарии на английском :)

Содержание

Базовые принципы языка

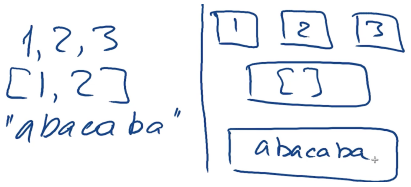
Модель данных: объекты

📎 **Данные** - то, что лежит в оперативной памяти на время исполнения нашей программы (числа, списки, функции, типы или классы, и др.).

📎 **Объект** - это абстракция, способ представления данных.

> ### Как работают объекты в питоне ?

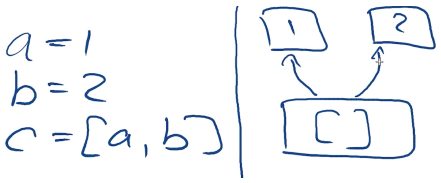
Оперативная память - склад.
Объекты - коробки для данных.



Хранение данных в памяти в “коробках”

Не важно какие данные мы бы хотели хранить в памяти, сначала мы бы поместили их в коробку и только затем на склад. Объект - контейнер в памяти, содержащий данные.

Все данные в языке представлены не только объектами. Вернее будет сказать, что данные представлены в виде объектов и отношений между объектами.



Данные - объекты и отношения

> **Итог:**
> Объекты - абстракция для хранения данных.
> Данные - объекты и отношения между объектами.

Между собой объекты различаются тремя атрибутами:

1. Идентификатор
2. Тип
3. Значение

> 📎 **Идентификатор** - то, что в любой момент времени позволяет отличить один объект от другого.
> Гарантируется, что никакие два объекта в один момент времени не обладают одинаковым идентификатором.

Идентификатор не меняется с момента создания объекта.

Когда же создаются объекты ?

Если интерпретатор встречает в коде **число**,

```
a = 8
```

то для него это явный знак того, что нужно создать объект.

Если же интерпретатор встретит **список,**

```
arr = [1, 2, 3]
```

то сначала он создаст по объекту для каждого элемента списка, а затем уже создаст объект и для самого списка.

Если интерпретатор встретит в коде **строку,**

```
s = 'aboba'
```

то он создаст для нее объект.

Если интерпретатор встретит в коде **кортеж,**

```
t = tuple(1, 2, 3)
```

то он создаст и для него объект

> **Итог:**
> Из слов выше становится понятно, что, когда мы меняем элемент в списке (mutable type), мы спокойно делаем это без создания нового объекта, в отличие от числа, строки или кортежа (immutable types)

Оператор присваивания

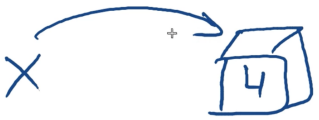
```
variable_name = something
```

something всегда будет каким-то объектом.

Оператор присваивание делает следующее: он запоминает за именем переменной из левой части, идентификатор объекта правой части

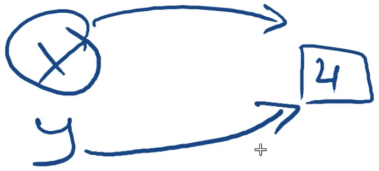
```
x = 4
```

- Создается объект для числа 4
- Устанавливается соответствие переменной *x* и идентификатора объекта 4, т.е. *x* теперь *ссылается* на объект, который содержит число 4.



- Идентификаторы между двумя одинаковыми объектами у неизменяемых типов одинаковые, у изменяемых - разные.

```
y = x
```



- Интерпретатор “вспоминает” идентификатор объекта *x* (равный идентификатору объекта 4)
- Устанавливается соответствие переменной *y* идентификатора объекта правой части

Здесь и далее вместо оборота *“соответствие переменной какому-либо идентификатору”* мы будем использовать термин *“ссылается”*

```
x = 4
y = x
```

- *x* ссылается на объект 4
- *y* ссылается на тот же объект, что и *x*

Функция **id()**

- получить идентификатор объекта.

```
x = [1, 2, 3]
print(id(x))  # 4318278848
print(id([1, 2, 3]))  # 4318278840
```

Идентификаторы отличаются, так как это два объекта, которые были созданы в разное время.

Оператор is

- проверяет ссылаются ли две переменные на один объект в памяти, результат True/False.

```
x = [1, 2, 3]
y = x

y is x          # True
y is [1, 2, 3]  # False
x is [1, 2, 3]  # False

[1, 2, 3] is [1, 2, 3]    # False
['1', '2'] is ['1', '2']  # False
```

В результате выполнения программы [1, 2, 3] будет создан пять раз.

Интересный момент

```
'a' * 20 is 'a' * 20  # True
'a' * 2 000 000 000 is 'a' * 2 000 000 000  # False
```

Все зависит от количества занимаемой памяти. До какого-то размера питон кэширует, потом все, различные участки.

Проверка поведения

```
variable_a = 1
variable_b = 1
variable_c = [1]
variable_d = [1]
variable_e = 10*10*10
variable_f = 10*10*10
variable_g = 10*10
variable_h = 10*10
variable_i = variable_a * (-1)
variable_j = -1
variable_k = variable_a * (-10)
variable_l = -10

print("a <-> b is {}".format(variable_a is variable_b))
print("c <-> d is {}".format(variable_c is variable_d))
print("e <-> f is {}".format(variable_e is variable_f))
print("g <-> h is {}".format(variable_g is variable_h))
print("i <-> j is {}".format(variable_i is variable_j))
print("k <-> l is {}".format(variable_k is variable_l))

'''
Output:
a <-> b is True
c <-> d is False
e <-> f is True
g <-> h is True
i <-> j is True
k <-> l is False
'''
```

📎 **Тип** - определяет возможные операции с объектом и принимаемые им возможные значения.

Тип объекта, как и его идентификатор, не меняется в течение жизни объекта.

Функция type()

- позволяет узнать тип объекта.

```
x = [1, 2, 3]
type(x)  # <class 'list'>
type(4)  # <class 'int'>

# особый пример
type(type(1))  # <class 'type'>
```

Таким образом, тип тоже является объектом. Не стоит пугаться этого, скоро станет очевидно, что в питоне все и чуть больше есть объекты! :)

Заполним табличку неизменяемых и изменяемых типов

Immutable	Mutable
• int, float, complex	• list
• bool	• dict
• tuple	• set

- str
- frozenset

▼ bool

Когда мы запустили наш интерпретатор, в памяти создаются два объекта True и False. И, в дальнейшем, все результаты логических выражений ссылаются на эти объекты

▼ string

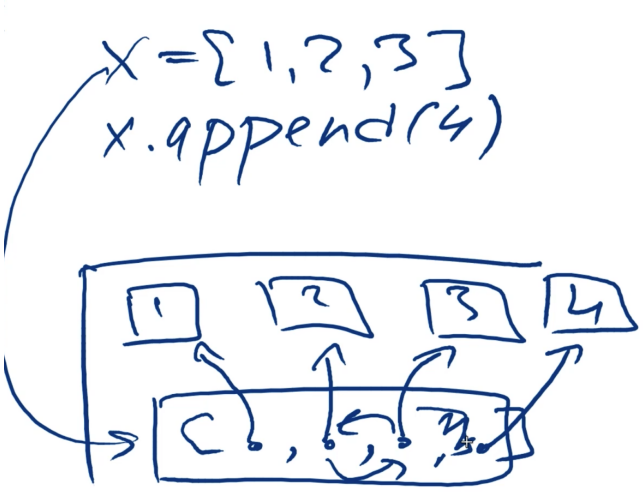
Неизменяемые последовательности символов. В отличие от списка и кортежа, мы храним не ссылки на объекты, а просто символы в юникоде. Стандартная кодировка строк — UTF-8 (U+hhhh).

▼ list, tuple

Для этого нужно разобраться с тем, что же является значением списка

```
x = [1, 2, 3]
```

Значение списка - упорядоченное множество ссылок на объекты, поэтому мы легко можем поменять значение списка, просто изменив ссылку на объект.



С кортежем (tuple) такая тема не прокатит. Кортеж есть неизменяемая последовательность ссылок

Теперь снова вернемся к созданию объектов.

Ранее мы говорили, что если интерпретатор встречает в коде, например, число, то он создаст для него новый объект. Это не всегда верно для **неизменяемых** типов.

Interning

Полезная ссылка

Микрооптимизация. Эта техника заключается в том, что питон оптимизирует используемую память и улучшает производительность кода, переиспользуя неизменяемые объекты, такие как строки, числа и экземпляры пользовательских типов.

Неизменяемые типы

Рассмотрим числа и переменные логического типа

```
1
True
False
```

Если интерпретатор встречает в программе **True** или **False**, то ему не нужно создавать новый объект для этого, он просто будет ссылаться на заранее созданные объекты.

Рассмотрим еще один момент.

```
a = 1
b = 1
c = 1
```

Нет ничего ужасного в том, чтобы эти переменные ссылались на один и тот же объект в памяти. Ведь до тех пор, пока наш объект неизменяем, нам ничего не грозит. Этот принцип показывает техника interning.

```
x = 888
```

Что мы можем гарантировать для этой строчки кода?

Если объекта 888 в памяти еще не было, то в памяти появится объект 888, иначе мы просто будем ссылаться на созданный ранее объект.

Эти утверждения работают для всех ***неизменяемых*** типов

Изменяемые типы

Для них все работает честнее. Рассмотрим это на примере списков

```
x = [ ]
y = [ ]

# id(x) != id(y)
```

Гарантируется, что когда интерпретатор встретит в коде список, то он создат для него объект. Эти утверждения работают для всех ***изменяемых*** типов.

Задачка

Реализуйте программу, которая будет вычислять количество различных объектов в списке.
Два объекта a и b считаются различными, если a is b равно False.

Вашей программе доступна переменная с названием objects, которая ссылается на список, содержащий не более 100 объектов. Выведите количество различных объектов в этом списке.

Решение

```
print(len(set(list(map(id, objects)))))
```

Интересный момент

В питоне в *set* объекты 1 и True будут одинаковыми, хотя 1 is not True. Это связано с тем, что с давних времен устоялась следующая традиция

```
False == 0
```

```
True == 1
```

А внутри проверка объектов у *set* устроена следующим образом:

1) проверка на hash()

2) a == b

Ну а раз True и 1 имеют на самом деле одно значение (True == 1), то и, стало быть, хэши у них будут одинаковые.

- **Хэш берется от** большинства **неизменяемых** объектов в питоне.
- **Изменяемые** контейнеры типа листа или словаря **не хэшируются**.

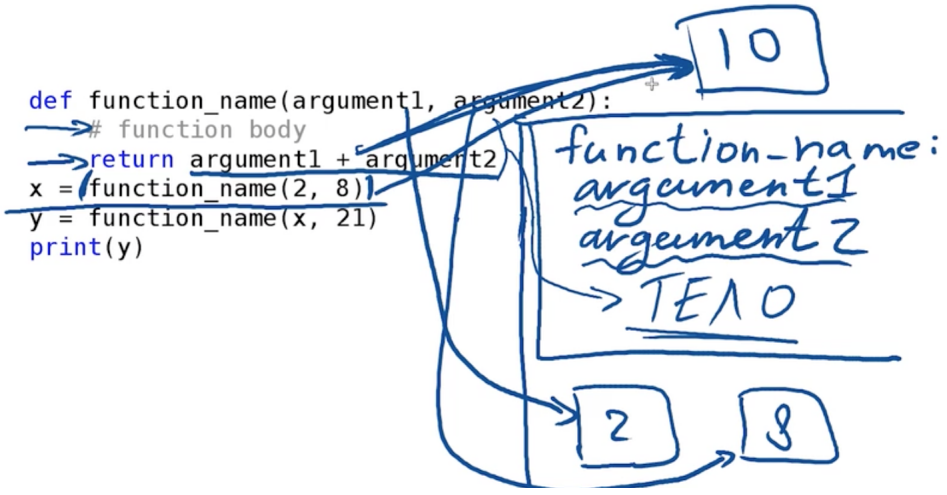
Функции и стек вызовов

Кейсы использования функций:

- *Переиспользование кода*
- *Разбиение программы на компоненты (структурирование кода)*
- *Сокрытие деталей реализации*

Определение функции выполняется целиком, а не построчно. После этого **для функции создастся объект** в памяти.

Можно взять type() от функции или посмотреть ее id(). Функции в питоне такие же объекты, как и числа или строки.

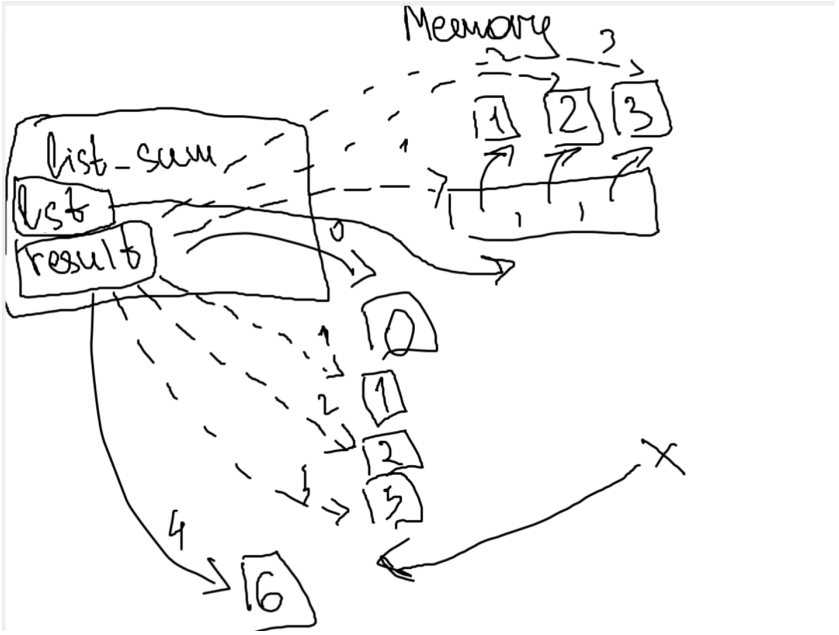


Оператору **return** интерпретатор говорит, подставь в правую часть выражения `x = function_name(2, 8)` ссылку на объект со значением 10 из оперативной памяти. Тогда нет ничего удивительного в том, что пер-я *x* будет равняться 10.

Работа функции из призмы оперативной памяти

```
def list_sum(lst):
    result = 0
    for elem in lst:
        result += elem
```

```
    return result


x = list_sum([1, 2, 3])
print(x)
```



Создание объектов по мере выполнения программы

Сначала создастся объект для функции **list_sum()**. Он будет включать в себя объекты **lst** и **result**. Далее, в памяти создадутся объекты 1, 2, 3 и лист для хранения их ссылок.

После этого **result** будет ссылаться на разные объекты в памяти. Интересно, что, когда **result** будет равен, например, 1, то эта единица может быть как объектом, созданным для списка, так и абсолютно не причастным другим объектом со значением 1. В конце создастся объект для функции **print()**.

> *Поэтому, выполняя даже маленькие на первый взгляд функции, в памяти может образоваться куча объектов.*

Стек вызовов

Функция **module** создает для нас объекты, ставит функции в выполнение и проделывает кучу других операций.

В стеке вызовов есть следующий момент: каждая функция, которая там находится, выполняется. Но лишь одна из них выполняется реально - та, что на вершине стека. Другие же функции ждут, пока выполнится их верхний сосед и до них не дойдет очередь.

Передача параметров в функцию

Различают два типа **параметров** функций - ***позиционные*** и ***именные***.

Позиционные - (non-keyword) - для которых важна позиция.

Именные - (keyword) - для которых нужно указывать название параметра и только потом передавать значение. Именные всегда идут после позиционных.

```
def printab(a, b):
    print(a)
    print(b)


# non-keyword way of calling a function
printab(1, 2)

# keyword way
print(b = 2, a = 1)

# keyword args only after non-keyword
print(1, b = 2)

lst = [10, 20]
printab(*lst)  # = printab(lst[0], lst[1])

args = {'a': 10, 'b':20}
print(**args)  # printab(a=args[key1],
               #         b=args[key2])
```

Интересный момент

> Нужно быть аккуратным с тем, чтобы передавать в параметры функции в качестве значения по умолчанию мутабельный объект.

Следовало бы упомянуть один важный момент: значение по умолчанию вычисляется только один раз - при первом вызове функции. При всех последующих вызовах будет повторно использован тот объект, который получили при первом вычислении значения по умолчанию.

В случае с неизменяемыми значениями (числа, строки) это поведение ни на что не влияет, а если в значении по умолчанию используется изменяемый тип - то все изменения этого аргумента будут сохраняться между вызовами.

В качестве примера используем новый список как значение по умолчанию для аргумента b:

```
def f(a, b=[]):
    b.append(a)


f(1)  # b = [1]
f(4)  # b = [1, 4]
f(100, b=[])  # b = [100]
f(7)  # b = [1, 4, 7]
```

Передача в функцию неопределенного числа аргументов

Позиционные аргументы

Допустим, нам нужно найти минимум двух числе. Для этого можно написать простую функцию.

А что, если нам будет необходимо найти минимум трех чисел? Четырех? Пяти?…

Для этого в питоне существует конструкция ***args,** которая позволяет нам передавать функции неопр. количество аргументов.

```
def find_min(*args):
    print(f'min value equals {min(args)}')
    # args is a tuple of arguments


find_min(10, 20, 30, 40, 50, -1, 100)
```

Именованные аргументы

*Возьми все именованные (которые не участвовали в инициализации пер-х a и b) параметры и засунь их в словарь **kwargs.***

```
def printab(a, b, **kwargs):
    print(a, b)
    for key in kwargs:
        print(key, kwargs[key])


printab(1, 2, t=3, f=4)
# 1 2
# t 3
# f 4
```

Как формально правильно выглядит синтаксическое определение функции?

```
def function_name(positional_args,
                  positional_args_with_default,
                  *pos_args_name,
                  keyword_only_args,
                  **kw_args_name):
    pass
```

1. Позиционные аргументы без значений по умолчанию.
2. Позиционные аргументы со значениями по умолчанию.
3. Позиционные аргументы, указанные как `*args`.
4. Именованные аргументы без значений по умолчанию.
5. Именованные аргументы со значением по умолчанию
6. Именованные аргументы, указанные как `**kwargs`.

Пространства имен и области видимости

> ***Пространство имен (namespace)** - словарь, содержащий имена и ссылки на объекты.*

```
def foo(a, b, c):
    print('function foo() local namespace: ', locals())
    print('global namespace: ', globals())
    print(f'locals is globals?\n-- {locals() is globals()}')
    return locals()  # to see what local variables this function has


amogus = 1000
foo(10, 'a', True)

print('globals is locals out of any function?\n--', globals() is locals()
```

Как понять, когда определяется namespace и область видимости?

> ***Область видимости (scope)*** - определяет, в каком *пространстве имен* (по сути в каком словаре) интерпретатор будет искать определение конкретного имени, а также и то в каком порядке этот поиск будет производиться.

Все просто: в месте, где выполняется присваивание, определяется и пространство имен; вместе с определением пространства имен, python определяет и область видимости.

Порядок поиска имен определяется правилом **LEGB**.

L - Local

E - Enclosed (замкнутость, выше локальнойго, но еще не глобальный скоуп)

G - Global

B - Built-ins

```
name = 'Alen'

def foo():
    name = 'Araz'

    def foo_2():
        print(name)
        print(locals())


    foo_2()

    print(name)
    print(locals())

foo()

print(name)
print(globals())
```

name в **foo_2()** функция **print()** будет искать на уровне **Local**; там она ничего не найдет, поэтому распечатает значение, найденное уже на уровне **Enclosed**, определенное в **foo()**.

Для чего нужны namespace и scope?

Это защитные механизмы, которые нужны, чтобы избегать конфликтов имен.

Интересный момент

Что будет исполняться первее? **print()** или **foo()**?

```
def foo():
    print('*knock-knock*\n-Who is there?')
    return ''


print('-Hello, motherfucker!', foo())
```

▼ Ответ

foo()

Все потому как **function_1(function_2(object))** работает как сложная функция в математике. Сначала мы должны проинициализировать аргумент для **function_1()**, но т.к. аргумент сам есть функция, то для начала мы должны проинициализировать это значение.

- *Соответственно на стеке вызовов сверху будут более внутренние функции, а снизу - более внешние.*

Интересный момент

Циклы и условия не создают локальные неймспейсы.

```
for i in range(5):
    x = i * i

print(x)  # 16
```

Итог

```
def a():
    print(x)
def b():
    x = 1
    a()
b() # NameError
```

Функция **b()** вызывает **a()**, зоны видимости функций не включены друг в друга. Т. е. интерпретатор не найдя имя 'x' в зоне видимости **a()** перейдёт в глобальную зону видимости, потом в зону видимости builtins - и не обнаружит 'x'. В результате: **NameError**

```
def b():
    def a():
        print(x)
    x = 1
    a()
b()  # 1
```

Функция **b()** также вызывает **a()**, но теперь **a()** определена внутри **b()**. Т. е. интерпретатор не найдя имя 'x' в зоне видимости **a()** перейдёт в зону видимости **b()** и найдет 'x' . Результат: **1**

Ну и упражнение для закрепления темы

> Какие числа будут выведены на экран в результате выполнения данного кода?

```
x, y = 1, 2

def foo():
    global y
    if y == 2:
        x = 2
        y = 1

foo()
print(x)
if y == 1:
    x = 3
print(x)
```

▼ Ответ

1, 3

Введение в ООП

Доп. материал

Данные в питоне представлены в виде **объектов и отношений между ними**. У любого объекта есть тип.

Но не всегда базовых типов нам хватает для решения задач, поэтому приходится определять пользовательские типы (они же классы).

Классы позволяют описать поведение объектов данного класса, создавать экземпляры.

> ***Классы в Python*** *- механизм и синтаксис для описания пользовательских типов данных.*

```
class MyClass:
    a = 10

    def func(self):
        print('Hello')
```

Важно! В отличие от функций, тело класса исполняется в момент его определения.

Так же как и для функций, для имен класса создается свой неймспейс.

======== (далее конспект по материалам из 6 уроков)

Классы и инстанцирование (создание экземпляра)

Скажем, у нас есть апликуха для нашей компании. И мы хотим представить наших работников в питоне.

Это хороший способ применить понятие класса, так как каждый сотрудник имеет свои атрибуты (имя, возраст, позиция) и методы (действия, что они могут выполнять).

Отличие класса от экземпляра в том, что класс - шаблон для создания экземпляров. Каждый работник будет экземпляром этого класса.

Интересный момент

```
class Employee:
    pass


# инстанцирование - создание экземпляра класса
emp_1 = Employee()
emp_2 = Employee()

print(emp_1)
print(emp_2)
# out: different addresses

emp_1.email = 'Corey.Schaffer@company.com'
emp_1.pay   = 50000

emp_2.email = 'Kylie.Smokey@company.com'
emp_2.pay   = 65000

print(emp_1.email, emp_2.email)
# out: emails

emp_3 = Employee()
print(emp_3.email)  # cause an error
```

Мы можем задать пустой класс, затем создавать от него экземпляры, такие, что каждый **может** иметь уникальные атрибуты.

emp_1 и **emp_2 имеют атрибут email**, тогда как для **emp_3 он отсутствует**; мы его не задавали в *определении* класса.

Метод `__init__()`

```
class Employee:
    def __init__(self, first, last, pay):  # initialize
        self.first = first
        self.last  = last
        self.pay   = pay
```

Параметр **self** передается в методы класса по умолчанию и по договоренности.

Теперь вместо строки

```
emp_1.first = 'Corey'
```

будет строка

```
self.first = first  # automatically initialize an instance with a string containing in argument first
```

Затем можно будет пользоваться ***конструктором*** при создании экземпляра класса **Employee()**.

```
emp_1 = Employee('Araz', 'Abdyev', 3000000)  # init method initialize attributes automatically
# self method will get emp_1
```

Добавим **методов** в наш класс. Пусть мы хотим выводить полное имя каждого сотрудника, для этого удобно использовать метод, чтобы не повторяться.

```
class Employee:
    def __init__(self, first, last, pay):  # initialize
        self.first = first
        self.last  = last
        self.pay   = pay


    def fullname(self):
        return f'{self.first} {self.last}'


# print(emp_1.fullname())
```

Интересный момент (про self)

Если мы попробуем убрать параметр self из метода, то мы получим ошибку.

```
  def fullname():
        return None


print(emp_1.fullname())
# TypeError: fullname() takes 0 positional arguments but 1 was given
```

Как так? Мы же не передвали ничего.

А все потому, что методу класса автоматически передался параметр **self**, когда мы вызвали метод от экземпляра **emp_1**.

Мы можем также **вызывать методы класса от его имени**.

```
class Employee:
    def __init__(self, first, last, pay):  # initialize
        self.first = first
        self.last  = last
        self.pay   = pay


    def fullname(self):
        return f'{self.first} {self.last}'


print(emp_1.fullname())
print(Employee.fullname(emp_1))  # another way to call class methods
```

Только тут мы должны явно передать в метод экземпляр. Это вместо использования ключего слова **self**.

Переменные класса и экземпляра

Те переменные, значения которых мы хотим расшарить на все созданные экземпляры.

Допустим, мы хотим увеличивать зарплату наших сотрудников каждый год на какой-то процент. Чтобы не указывать это вручную, создадим для этого переменную класса.

```
class Employee:
    raise_amount = 1.04  # 4% up

    def __init__(self, first, last, pay):
        self.first = first
        self.last  = last
        self.pay   = pay

    def fullname(self):
        return f'{self.first} {self.last}

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)  # or Employee.raise_amount
```

! Обращаться к переменным класса нужно через экземпляр или через название класса.

```
emp_1 = Employee('Araz', 'Abdyev', 100000)

print(emp_1.raise_amount)
print(Employee.raise_amount)
```

Оба **print**'а выведут одно и то же, но вот дело в том, что в экземпляре emp_1 этой пер-й нет. Она наследуется от класса **Employee**.

Это можно проверить, посмотрев на неймспейсы.

```
print(emp_1.__dict__)
print('-'*30)
print(Emplyee.__dict__)
```

Но, нужно быть осторожным.

```
Employee.raise_amount = 1.05  # задать значение пер-й класса

emp_1.raise_amount = 1.06     # задать значение пер-й экземпляра (пер-я появится в его неймспейсе)
```

Поскольку, при использовании таких пер-х от **self** или от имени **класса**, результат может получиться разный.

```
def apply_raise(self):
        self.pay = int(self.pay * Employee.raise_amount)  # увеличение на 5% для emp_1

def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)     # увеличение на 6% для emp_1
```

Методы класса и статические методы

> ***Метод класса*** *- такой метод, перед которым устанавливается декоратор* ***@classmethod*** *и который принимает в качестве первого аргумента параметр* ***cls****.*

Обычные методы в классе автоматически принимают экземпляр в качестве параметра и по договоренности его называют **self.**

Как это можно изменить, чтобы метод вместо экземпляра автоматически принимал в себя **класс** как первый аргумент? Для этого и нужны **методы класса**.

Добавим новый метод. Чтобы сделать его методом класса, нужно написать декоратор.

```
@classmethod  # making regular method written to a class as a classmethod
def set_raise_amt(cls, amount):
    # cls - also convention as a "self"
    cls.raise_amt = amount


Employee.set_raise_amt(1.05)
```

Использование ***метода класса от экземпляра*** будет работать точно так же, только вот обычно так не делают.

Методы класса можно использовать как **альтернативные конструкторы**.

```
emp_str_1 = 'John-Doe-70000'

first, last, pay = emp_str_1.split('-')

new_emp = Employee(first, last, pay)
```

Напишем конструктор для случая, когда мы получаем строки с тире и нужно из этого сделать нового сотрудника.

```
@classmethod
def from_string(cls, emp_str):
    first, last, pay = emp_str.split('-')
    return cls(first, last, pay)

emp_str_1 = 'John-Doe-70000'

new_emp = Employee.from_string(emp_str_1)

print(new_emp.email)  # 'John.Do@email.com'
```

Статические методы

Выше были рассмотрены:

- обычные методы в классе, которые принимают по умолчанию параметр **self**
- методы класса, которые принимают по умолчанию параметр **cls**

> ***Статический метод*** *- это обычный метод, написанный в классе просто потому, как он имеет логическую связь с другими элементами класса. По умолчанию не принимает в себя* ***cls*** *или* ***self****. Перед объявлением нужно использовать декоратор* ***@staticmethod****.*

```
@staticmethod
def is_workday(day):
    if day.weekday() in (5, 6):
        return False
    return True


import datetime

my_date = datetime.date(2016, 7, 10)

print(Employee.is_workday(my_date))  # False because it's Sunday
```

Когда использовать статические методы?

- тогда, когда не нужно иметь доступ к экземпляру или классу.

Наследование - создание подкласса

> ***Наследование*** *- механизм, который позволяет пользоваться атрибутами и методами класса, от которого мы наследуемся; использовать (или дописывать) функциональность родительского класса* ***без его изменения.***

```
class Employee:

    raise_amt = 1.04

    def __init__(self, first, last, pay):
        self.first = first
        self.last  = last
        self.email = first + '.' + last + '@email.com'
        self.pay   = pay


    def fullname(self):
        return f'{self.first} {self.last}'


    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)


class Developer(Employee):
    pass


dev_1 = Employee('Araz', 'Abdyev', 100000)
print(dev_1.email)

dev_2 = Developer('Alen', 'Karapetyan', 99000)
print(dev_2.email)
```

1. При создании экземпляра класса **Developer** python заглянет в описание этого класса и будет искать там метод **__init__.**
2. Когда он его не найдет, то дальше он пойдет по “цепочке наследований”, пока не найдет нужное ему. Это называется **MRO** ***(Method Resolution Order)*****.**

Интересный момент (про MRO)

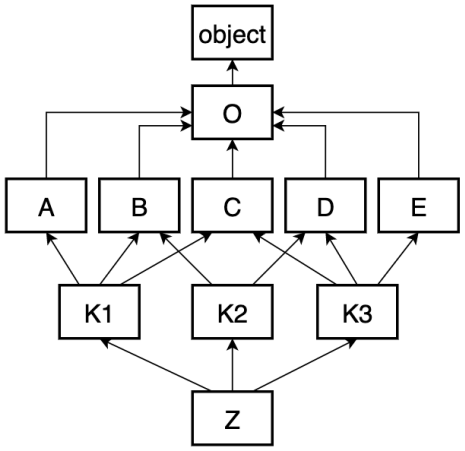
> ***MRO*** *- порядок разрешения методов в Python. Но это относится не только к методам, но и атрибутам класса, так как методы - частный случай понятия “атрибут”.*

И так, вследствие множественного наследования в языке пайтон, существует так называемый “порядок поиска атрибутов”.

Например, как будет происходить поиск в этом случае?

```
class O: ...
class A(O): ...
class B(O): ...
class C(O): ...
class D(O): ...
class E(O): ...
class K1(A, B, C): ...
class K2(B, D): ...
class K3(C, D, E): ...
class Z(K1, K2, K3): ...
```

Понятно, что поиск будет проводиться по следующей схеме:



Только вот каков порядок?)

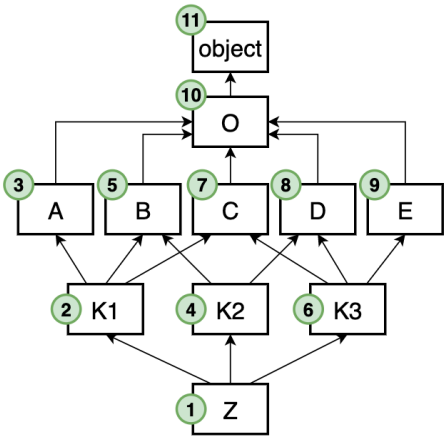
Узнаем это, запустив метод **mro()**

```
class O: ...
class A(O): ...
class B(O): ...
class C(O): ...
class D(O): ...
class E(O): ...
class K1(A, B, C): ...
class K2(B, D): ...
class K3(C, D, E): ...
class Z(K1, K2, K3): ...


print(Z.mro())
# [<class '__main__.Z'>, <class '__main__.K1'>, <class '__main__.A'>, <class '__main__.K2'>, <class '__main__.B'>, <class '__main__.K3

# сделаем понагляднее вывод, печатая только имена классов со стрелочками:
def print_mro(T):
    print(*[c.__name__ for c in T.mro()], sep=' -> ')

print_mro(Z)
# Z -> K1 -> A -> K2 -> B -> K3 -> C -> D -> E -> O -> object
```

Почему так?

В 1996 году был предложен алгоритм C3-линеаризации, принятый впоследствии в версии Python 2.3.

> C3-линеаризация — процесс превращения графа множественного наследования в плоский список.



Метод help(cls)

Позволяет узнать всю информацию о классе, в т.ч. его MRO.

```
help(Developer)


"""
Help on class Developer in module __main__:

class Developer(Employee)
 |  Developer(first, last, pay)
 |
 |  Method resolution order:
 |      Developer
 |      Employee
 |      builtins.object
 |
 |  Methods inherited from Employee:
 |
 |  __init__(self, first, last, pay)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  apply_raise(self)
 |
 |  fullname(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Employee:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from Employee:
 |
 |  raise_amt = 1.04
"""
```

Допустим теперь мы хотим у разработчиков знать еще и их язык программирования.

Как это сделать чисто без копирования уже готового **__init__**?

```
class Employee:

    raise_amt = 1.04

    def __init__(self, first, last, pay):
        self.first = first
        self.last  = last
        self.email = first + '.' + last + '@email.com'
        self.pay   = pay


    def fullname(self):
        return f'{self.first} {self.last}'


    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)


class Developer(Employee):
    def __init__(self, first, last, pay, language):
            super().__init__(first, last, pay)

            #   good for multiple intheritance
            # Employee.__init__(self, first, last, pay)

            self.language = language
```

Метод super()

Позволяет получить доступ к атрибутам родительского класса.

Интересный момент (передача изменяемых объектов в качестве умолчательного параметра)

В питоне нежелательно использовать в качестве параметра по умолчанию изменяемые объекты, потому что они создаются только один раз при определении функции. Это означает, что если функция изменяет этот объект, то он будет изменен для всех последующих вызовов этой функции.

Например, рассмотрим следующий код:

```
def append_to_list(item, my_list=[]):
    my_list.append(item)
    return my_list


print(append_to_list(1))
print(append_to_list(2))
print(append_to_list(3))
```

Ожидаемый результат:

[1]
[2]
[3]

Однако фактический результат будет таким:

[1]
[1, 2]
[1, 2, 3]

Это происходит потому, что список **my_list** создается только один раз при определении функции, и он используется для всех последующих вызовов. Поэтому при каждом вызове функции элементы добавляются в один и тот же список.

Чтобы избежать этой проблемы, в качестве параметра по умолчанию лучше использовать неизменяемые объекты, такие как None, True или False. Если нужно передать изменяемый объект в качестве параметра по умолчанию, то его можно создавать внутри функции. Например:

```
def append_to_list(item, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(item)
    return my_list


print(append_to_list(1))
print(append_to_list(2))
print(append_to_list(3))

# [1]
# [2]
# [3]
```

Функция isinstance()

Позволяет узнать, является ли объект экземпляром класса.

```
print(isinstance(mgr_1, Manager))  # True
print(isinstance(-100, int))  # True
```

Функция issubclass()

Позволяет узнать, является ли первый класс подклассом второго.

```
print(issubclass(Developer, Employee))  # True
```

Специальные (магические/дандер) методы

- позволяют эмулировать **built-in поведение** языка питон, а также как мы имплементируем **перегрузку операторов**

```
# overloading operators
print(1 + 2)
print('1' + '2')
```

📎 Статья про дандер методы на хабре

Сейчас, когда мы пытаемся вывести экземпляр методом **print()**, мы получаем какой-то страшный вывод. Сделаем что-нибудь более юзерфрендли из этого.

Интересный момент (str() vs repr())

📎 **str()** возвращает **user-friendly** описание объекта

repr() возвращает **developer-friendly** представление объекта

> *Если **str()** нужен для красивого представления объекта пользователю, то для чего нужен **repr()**?*

Например, мы вернули из базы данных какое-то значение и распечатали его:

‘2023-10-21 19:03’

и что же это? Может быть такое, что это объект типа date.datetime. А может и нет, значит нужно конвертнуть его из строкового в нужный тип.

```
def __repr__(self):
    return f'Employee({self.first}, {self.last}, {self.pay})'


def __str__(self):
    return f'{self.fullname()} - {self.email}'


print(emp_1)  # calls emp_1.__str__()
print(repr(emp_1)  # calls emp_1.__repr__()
```

Теперь принты будут работать нормально.

> Таким образом, в любом более-менее сложном классе сначала мы определим эти три метода:
> **__init__**
> **__repr__**
> **__str__**

```
print(1 + 2)              # 3
print(int.__add__(1, 2))  # 3
```

Допустим, мы хотим использовать оператор ‘+’ вместе с нашими employees. Складывая одного с другим, мы на самом деле складывали бы их зарплаты.

```
class Employee:

    def __init__(self, first, last, pay):
        self.first = first
        self.last  = last
        self.pay   = pay

    def __repr__(self):
        return f'Employee({self.first}, {self.last}, {self.pay})'

    def __str__(self):
        return f'{self.fullname()} - {self.pay}'

    def __add__(self, other):
        return self.pay + other.pay

    def fullname(self):
        return f'{self.first} {self.last}'

    @staticmethod
    def sum(*args):
        """Add salaries of list of employees"""
        s = 0

        for emp in args:
```

```
            s += emp.pay

        return s


emp_1 = Employee('Araz', 'Abdyev', 100000)
emp_2 = Employee('Alen', 'Karapetyan', 55000)
print(emp_1)

print(repr(emp_1))


print(emp_1 + emp_2)
print(Employee.sum(emp_1, emp_2))
```