# БИБЛИОТЕКИ ЯЗЫКА PYTHON ДЛЯ КОМПЬЮТЕРНЫХ ВЫЧИСЛЕНИЙ И МОДЕЛИРОВАНИЯ

Функциональное и объектно-ориентированное программирование в **Python**


**Лобанов Алексей Владимирович**
Главный специалист отдела разработки и внедрения АИС,
Ассистент кафедры информационных компьютерных технологий РХТУ им. Д.И. Менделеева

# ТЕМЫ

- Функции
- Аргументы функции
- **Lambda** функции
- Генераторы
- Декораторы

- Основные термины и понятия ООП
- Классы
- Наследование классов
- Магические методы и их переопределение
- Жизненный цикл объекта
- Сокрытие данных в Python
- Свойства, методы класса и статические методы.

# ПОЛЕЗНЫЕ РЕСУРСЫ

- Функции официальная документация
- Функции
- Генераторы vs функции
- Декораторы

- Официальная документация по классам
- Основы ООП Python
- Магические методы

# ФУНКЦИИ

Функция в python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции def.

Функции используются для организации определенного функционала в одну конструкцию, что облегчает его многоразовое использование и уменьшает количество работы необходимое для внесения модификаций.

**Псевдо-код**

```
1 def <имя функции>(<аргументы>):
2     <тело функции>
```

**Реальный пример**

```
1 def func():
2     pass
```

Ключевое слово pass используется для указания отсутствия действий

# ФУНКЦИИ (КЛАССИФИКАЦИЯ)

Функции

```
def func1():
    print(5)
```

Процедуры
Ничего не возвращают

```
def func2():
    return 5
```

Функции
Возвращают значение

# ФУНКЦИИ (КЛАССИФИКАЦИЯ)

Функции

```
def func1():
    for i in range(10):
        print(i)
```

Без аргументов

```
def func2(a, b):
    return a + b
```

С аргументами

# ФУНКЦИИ(ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ)

Для того чтобы вернуть какое-либо значение из функции используется ключевое слово return

Любая функция возвращает только одно значение, и если оно не указано явно, функция возвращает None

## Исходный код

```
1 def func1():
2     pass
3 a = func1()
4print(a)
5
6 def func2():
7     return None
8 b = func2()
9 print(b)
```

## Результат

```
None
None
```

None - это отдельный тип данных который наиболее часто используется для обозначения отсутствия данных

# ФУНКЦИИ(МНОЖЕСТВО RETURN)

## МНОЖЕСТВО RETURN

Ключевое слово return функционирует как break для цикла, после певрого встреченного return'а функция завершает свое выполнение

### Исходный код

```
1 def func():
2     return 1
3     return 2
4 a = func()
5 print(a)
```

### Результат

```
1
```

# АРГУМЕНТЫ ФУНКЦИИ

Порядковые явные

Именованные явные

Аргументы

Порядковые неявные

Именованные неявные

# АРГУМЕНТЫ ФУНКЦИИ

Порядковые явные

```
1  def func(a, b):
2      return (a * b) + b
3
4  res1 = func(5, 2)
5  # Можно явно указывать какому
6  # аргументу присваевается значение
7  res2 = func(5, b=2)
8  res3 = func(b=2, a=5)
9  print(res1)
10 print(res1 == res2 == res3)
```

Наиболее простой тип аргументов, перечисляются через запятую (все указываются при вызове)

Именованные явные

```
1  def func(a, b, c=3):
2      return (a * b) + c
3
4  res1 = func(2, 4)
5  print(res1)
6  res2 = func(2, 4, 5)
7  print(res2)
8  res3 = func(2, c=5, b=4)
9  print(res3)
```

При определении функции им присваивается значение которое будет использоваться, если при вызове функции оно не будет перезаписано явно

# АРГУМЕНТЫ ФУНКЦИИ

Порядковые неявные

```
1 def func(a, b, *c):
2     print(a)
3     print(b)
4     print(c)
5
6 func(1, 2, 3, 4, 5)
```

Порядковые неявные аргументы принимают все значения которые переданы в функцию без имени и не описаны явно при определении функции

Именованные неявные

```
1 def func(a, b, c=3, **d):
2     print(a)
3     print(b)
4     print(c)
5     print(d)
6
7 func(1, b=2, c=3, d=4, e=5)
```

Именованные неявные аргументы принимают все значения которые переданы в функцию по имени и не описаны явно при определении функции

# АРГУМЕНТЫ ФУНКЦИИ

```
1 def func(a, b, *c, d=4, e=5, **f):
2     print(a)
3     print(b)
4     print(c)
5     print(d)
6     print(e)
7     print(f)
8
9 func(1, 2, 3, 4, e=6, f=7, g=8)
```

Наиболее частно порядковые неявные аргументы называют *args, а именованные неявные - **kwargs

# LAMBDA ФУНКЦИИ

Анонимные функции (lambda функции) используются когда нецелесообразно реализовывать целую функцию. Использование анонимных функций сильно затрудняет отладку кода, поэтому их стоит использовать крайне редко в определенных случаях. Например, **lambda** функции используются при передаче в качестве аргумента в другую функцию.

Пример **lambda** функций

```
1 f = lambda a, b: a * b
2 res1 = f(5, 2)
3 print(res)
4 res2 = (lambda a, b: a + b)(5, 2)
5 print(res2)
```

```
10
7
```

# ГЕНЕРАТОРЫ

Генератор в Python — это функция с уникальными возможностями. Она позволяет приостановить или продолжить работу. Генератор возвращает итератор, по которому можно проходить пошагово, получая доступ к одному значению с каждой итерацией. Генератор создается по принципу обычной функции. Отличие заключается в том, что вместо return используется инструкция yield. Она уведомляет интерпретатор Python о том, что это генератор, и возвращает итератор.

Общий вид генератора

```
def gen_func(args):
    ...
    while [cond]:
        ...
        yield [value]
```

# ГЕНЕРАТОРЫ

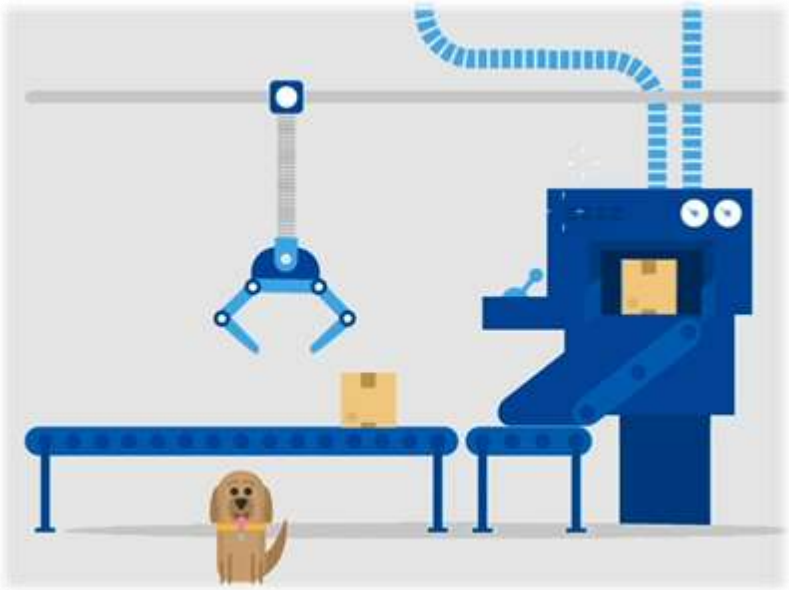
Return всегда является последней инструкцией при вызове функции, в то время как yield временно приостанавливает исполнение, сохраняет состояние и затем может продолжить работу позже. “Перебирать” генератор можно при помощи **next** или **for.**

Применение **next**

```
def generator1(end=5):
    n = 0
    while n <= end:
        yield n ** 2
        n += 1

g = generator1(5)

print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

Применение **for**

```
def generator1(end=5):
    n = 0
    while n <= end:
        yield n ** 2
        n += 1

for g in generator1(5):
    print(g)
```

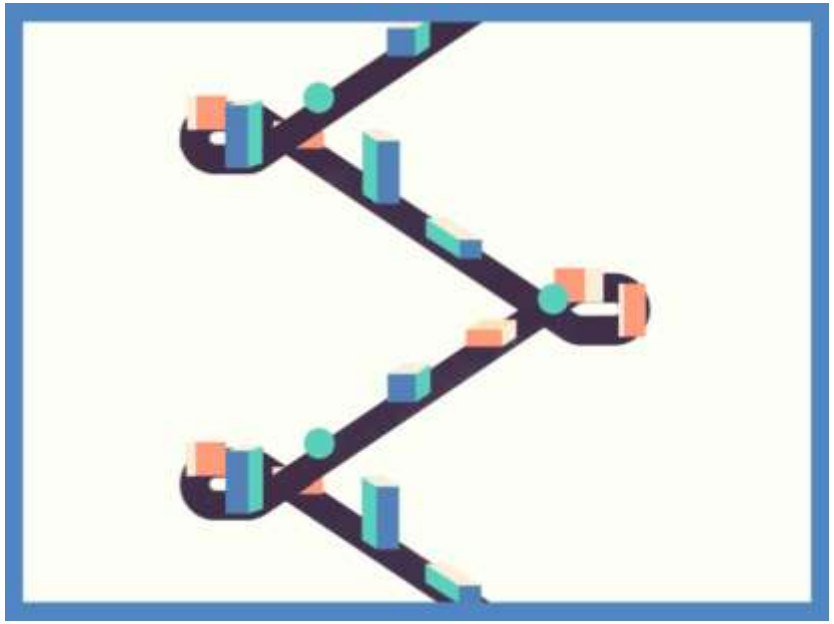


# ГЕНЕРАТОРЫ

Python позволяет писать выражения генератора для создания анонимных функций генератора. Процесс напоминает создание лямбда-функций для создания анонимных функций. Синтаксис похож на используемый для создания списков с помощью цикла for. Однако там применяются квадратные скобки, а здесь — круглые.

Анонимный генератор

```
alist = [4, 16, 64, 256]

out = (a**(1/2) for a in alist)

print(next(out))
print(next(out))
print(next(out))
print(next(out))
print(next(out))
```



# ДЕКОРАТОРЫ

Декораторы предназначены для модификации функций с помощью других функций.

Декораторы — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код.

Декоратор

```
def my_decorator(func):
    def wrap():
        print("++++++")
        func()
        print("++++++")
    return wrap

@my_decorator
def my_func():
    print("Hello!")
```

Декорируемая функция

# ЧТО ТАКОЕ ООП?

**Объектно-ориентированное программирование (ООП)** — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

**Класс** — тип, описывающий устройство объектов. Или комплексный тип данных, состоящий из набора полей и методов

**Объект** — это экземпляр класса. Класс можно сравнить с чертежом, по которому создаются объекты.

**Поле** - переменная являющаяся частью класса

**Метод** - функция являющаяся частью класса

Python соответствует принципам объектно-ориентированного программирования. Кроме того, стоит отметить, что всё в **Python** является объектами: строки, списки, числа и тд.

# КЛАССЫ (СОЗДАНИЕ)

Классы оформляются с помощью ключевого слова class и в виде блока с отступом, содержащего атрибуты класса (которые являются переменными) и методы класса (которые являются функциями).

Общий вид класса

```
class <имя_класса>:
    <атрибуты_класса>
    <методы_класса>
```

Класс собака

```
class Dog:
    breed = "Доберман"

    def bark(self):
        print("Гав-гав")
```

# КЛАССЫ (ОБЪЕКТЫ)

Для того, чтобы создать объект класса, используются функции, называемые конструкторами. Они по названию совпадают с названием класса. Иногда конструкторам нужно передать некие параметры (об этом дальше), иногда это не требуется. У объектов можно изменять и получать поля, а так же вызывать методы класса.

Конструктор без параметров

```
my_dog = Dog()
print(my_dog.breed)
my_dog.bark()
```

Конструктор с параметрами

```
my_dog = Dog("Гончая")
print(my_dog.breed)
my_dog.bark()
```

# КЛАССЫ (SELF)

**self -** это ни в коем случае не зарезервированное слово. Это просто название переменной. В методах класса первый параметр функции **по соглашению** именуют **self**, и это ссылка на сам объект этого класса.

```
class Dog:
    breed = "Доберман"

    def bark(self):
        print("Гав-гав")

    def write_breed(self):
        print(self.breed)
```

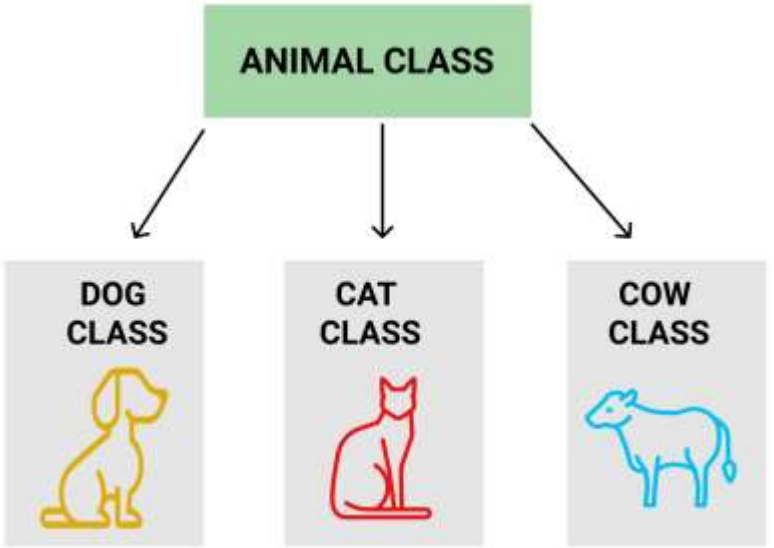
```
my_dog = Dog()
my_dog.breed = "Гончая"
my_dog.write_breed()
```

```
Гончая

Process finished with exit code 0
```

# НАСЛЕДОВАНИЕ КЛАССОВ

Наследование классов — очень мощная возможность в объектно-ориентированном программировании. Оно позволяет создавать производные классы (классы наследники), взяв за основу все методы и элементы базового класса (класса родителя). Таким образом экономится масса времени на написание и отладку кода новой программы.

```python
class Animal:
    def __init__(self, name, color):
        self.name = name
        self.color = color

class Cat(Animal):
    def purr(self):
        print("Purr...")

class Dog(Animal):
    def bark(self):
        print("Woof!")

fido = Dog("Fido", "brown")
print(fido.color)
fido.bark()
```



# МАГИЧЕСКИЕ МЕТОДЫ

Магические методы, если говорить простыми словами, это такие функции класса, которые переопределяют какое-либо базовое поведения класса. Такое поведение, как создание объекта этого класса, сложение объектов класса, строковое представление объектов и тд. Магические методы начинаются с __ и ими же заканчиваются.

# МАГИЧЕСКИЕ МЕТОДЫ (__INIT__)

__init__ - Этот метод используется для определения/инициализации экземпляра. Именно в методе __init__ обычно задаются начальные атрибуты создаваемого экземпляра.

```
class Dog:
    breed = "Доберман"

    def __init__(self, breed=None):
        if breed is not None:
            self.breed = breed

    def bark(self):
        print("Гав-гав")

    def write_breed(self):
        print(self.breed)
```

# МАГИЧЕСКИЕ МЕТОДЫ (__STR__)

Метод __str__ может вернуть более описательные данные экземпляра. Следует отметить, что этот метод используется функцией print() для отображения информации экземпляра

```
class Dog:

    def __init__(self, breed=None, age=5):
        self.breed = breed
        self.age = age

    def __str__(self):
        return "{}, {} лет".format(self.breed, self.age)

    def bark(self):
        print("Гав-гав")
```

# МАГИЧЕСКИЕ МЕТОДЫ (ЛОГИЧЕСКИЕ МЕТОДЫ)

`__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, `>=` .

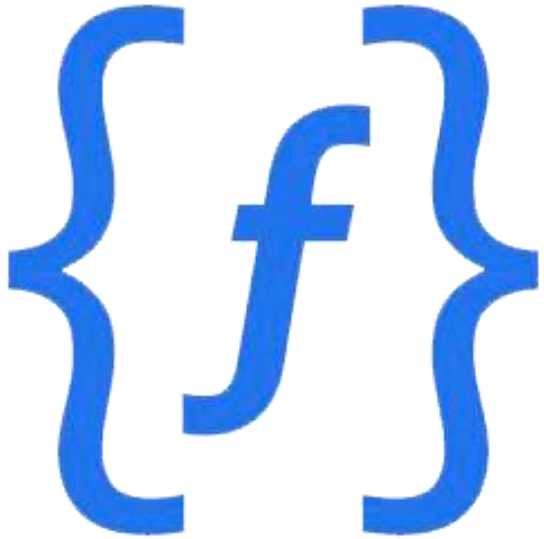
```
class Dog:

    def __init__(self, breed=None, age=5):
        self.breed = breed
        self.age = age

    def __eq__(self, other):
        return self.age == other.age

    def __gt__(self, other):
        return self.age > other.age

    def __ge__(self, other):
        return self.age >= other.age
```

```
my_dog1 = Dog("Доберман", 5)
my_dog2 = Dog(age=5)
print(my_dog2 >= my_dog1)
```

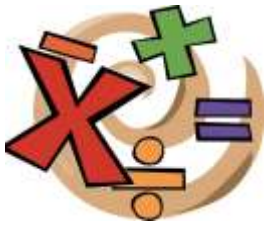
# МАГИЧЕСКИЕ МЕТОДЫ (__CALL__)

Позволяет любому экземпляру вашего класса быть вызванным как-будто он функция. __call__ принимает произвольное число аргументов; то есть, вы можете определить __call__ так же как любую другую функцию, принимающую столько аргументов, сколько вам нужно.

```
class Test:

    def __init__(self, n = 1):
        self.n = n

    def __call__(self, a, b):
        return self.n * (a + b)


t = Test(5)

print(t(3, 2))
```



# МАГИЧЕСКИЕ МЕТОДЫ (МАТЕМАТИЧЕСКИЕ МЕТОДЫ)



- `__add__(self, other)`
  Сложение.
- `__sub__(self, other)`
  Вычитание.
- `__mul__(self, other)`
  Умножение.
- `__div__(self, other)`
  Деление, оператор `/` .

```python
class Rect:

    def __init__(self, a, b):

        self.a = a
        self.b = b

    def __str__(self):

        return f"a={self.a} b={self.b}"

    def __add__(self, other):
        return Rect(self.a + other.a, self.b + other.b)

r1 = Rect(1, 2)
r2 = Rect(2, 3)

r3 = r1 + r2

print(r3)
# a=3 b=5
```

# ЖИЗНЕННЫЙ ЦИКЛ

__init__

Создание → Использование → Уничтожение

__del__

# СОКРЫТИЕ ДАННЫХ

**Сокрытие данных -** понятие, суть которого в том, что детали реализации класса должны быть скрыты, и чистый стандартный интерфейс должен быть представлен тем, кто будет использовать класс. В других языках программирования это достигается с использованием частных методов и атрибутов, которые закрывают доступ извне к определенным методам и атрибутам класса.

Идеология Python несколько иначе. В сообществе Python часто звучит фраза «мы все взрослые и по своему согласию здесь», что означает, что не следует устанавливать свои ограничения на доступ к отдельным частям класса. Так как все равно невозможно обеспечить строгую частность метода или атрибута.

# СОКРЫТИЕ ДАННЫХ

**Условно частные** методы и атрибуты оформляются с единым подчеркиванием в начале имени. Это частные методы, которые не должны взаимодействовать со внешней частью программы. Но часто это правило условно; внешняя часть программы может получить к ним доступ.
Реальная особенность этих методов лишь в том, что from module_name import * не будет импортировать переменные, которые начинаются с единого подчеркивания.

**Строго частные** методы и атрибуты оформляются с двойным подчеркиванием в начале имени. Таким образом их имена искажаются, и внешняя часть программы не может получить к ним доступ. Но это делается не для того, чтобы обеспечить их частность, а чтобы избежать ошибок, если где-либо в коде есть подклассы, которые имеют методы или атрибуты с такими же именами.

# СОКРЫТИЕ ДАННЫХ

Пример частных атрибутов

```
class MyClass:

    _secret_field = 5
    __super_secret_field = 15

mc = MyClass()

print(mc._secret_field)
# 5
mc._secret_field = 10
print(mc._secret_field)
# 10

print(mc._MyClass__super_secret_field)
# 15
mc._MyClass__super_secret_field = 20
print(mc._MyClass__super_secret_field)
# 20

print(mc.__super_secret_field)
# AttributeError
```

# СВОЙСТВА

Свойства позволяют нам создавать вычисляемые поля класса и ограничивать доступ к их изменению извне. Для этого используется декоратор property, при вызове мы обращаемся к свойству по имени функции, но не используем скобки (как у атрибутов).

```
class Rect:
    def __init__(self, h, w):
        self.h = h
        self.w = w

    @property
    def s(self):
        return self.h * self.w
```

# МЕТОДЫ КЛАССА

Методы класса несколько отличаются от методов объекта класса: они вызываются классом, который передается параметру cls метода. Чаще всего это используется в фабричных методах: создается экземпляр класса, при этом используются иные параметры, чем те, которые обычно передаются в конструктор класса. Методы класса оформляются с декоратором classmethod.

```
class Rect:
    def __init__(self, h, w):
        self.h = h
        self.w = w

    @classmethod
    def square(cls, h):
        return cls(h, h)

    @property
    def s(self):
        return self.h * self.w
```

# МЕТОДЫ КЛАССА

Статические методы похожи на методы класса с тем отличием, что они не берут никаких дополнительных аргументов; они аналогичны обычным функциям класса. Они оформляются с декоратором staticmethod.

```
class Rect:
    def __init__(self, h, w):
        self.h = h
        self.w = w

    @staticmethod
    def print_msg(name):
        print("Я прямоугольник по имени {}".format(name))

Rect.print_msg("Ваня")
```

# ЗАДАНИЕ ДЛЯ САМОКОНТРОЛЯ

**Задание 1**

- Напишите функцию, которая принимает строку из одного или нескольких слов и возвращает ту же строку, но с перевернутыми всеми словами из пяти или более букв. **Hey fellow warriors -> Hey wollef sroirraw**
- Создать генератор **N** первых чисел Фибоначчи.
- Создать декоратор, для вывода аргументов и результата выполнения функции с двумя переменными.

**Задание 2**

Используя объектно-ориентированный подход создать программу на языке Python, позволяющую определить температуру смеси по данным двух смешивающихся потоков. Исходя из того, что температуру можно вычислить по формуле:

$$t_m = \frac{m_1 c_{p1} t_1 + m_2 c_{p2} t_2}{m_1 c_{p1} + m_2 c_{p2}}$$

Где m – масса, $c_p$ – удельная теплоёмкость, t – температура.

# СПАСИБО ЗА ВНИМАНИЕ!