Сегодня мы поговорим об объектно-ориентированном программировании и о его применении в python.

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

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

Python соответствует принципам объектно-ориентированного программирования. В python всё является объектами - и строки, и списки, и словари, и всё остальное.

Но возможности ООП в python этим не ограничены. Программист может написать свой тип данных (класс), определить в нём свои методы.

Это не является обязательным - мы можем пользоваться только встроенными объектами. Однако ООП полезно при долгосрочной разработке программы несколькими людьми, так как упрощает понимание кода.

Приступим теперь собственно к написанию своих классов на python. Попробуем определить собственный класс:

```
>>>
```

```
>>> # Пример простейшего класса, который ничего не делает

... class A:

...     pass
```

Теперь мы можем создать несколько экземпляров этого класса:

```
>>>
```

```
>>> a = A()

>>> b = A()

>>> a.arg = 1  # у экземпляра a появился атрибут arg, равный 1

>>> b.arg = 2  # а у экземпляра b - атрибут arg, равный 2

>>> print(a.arg)

1

>>> print(b.arg)
```

```
2

>>> c = A()

>>> print(c.arg)  # а у этого экземпляра нет arg

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AttributeError: 'A' object has no attribute 'arg'
```

Классу возможно задать собственные методы:

```
>>>
```

```
>>> class A:

...     def g(self): # self - обязательный аргумент, содержащий в
себе экземпляр

...                  # класса, передающийся при вызове метода,

...                  # поэтому этот аргумент должен присутствовать

...                  # во всех методах класса.

...         return 'hello world'

...

>>> a = A()

>>> a.g()

'hello world'
```

И напоследок еще один пример:

```
>>>
```

```
>>> class B:

...     arg = 'Python' # Все экземпляры этого класса будут иметь
атрибут arg,

...                    # равный "Python"

...                    # Но впоследствии мы его можем изменить

...     def g(self):

...         return self.arg

...

>>> b = B()

>>> b.g()

'Python'

>>> B.g(b)

'Python'

>>> b.arg = 'spam'

>>> b.g()

'spam'
```

Инкапсуляция

Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса.

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.

Одиночное подчеркивание в начале имени атрибута говорит о том, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени.

```
class A:

    def _private(self):

        print("Это приватный метод!")



>>> a = A()

>>> a._private()

Это приватный метод!
```

Двойное подчеркивание в начале имени атрибута даёт большую защиту: атрибут становится недоступным по этому имени.

```
>>>
```

```
>>> class B:

...     def __private(self):

...         print("Это приватный метод!")

...

>>> b = B()

>>> b.__private()

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AttributeError: 'B' object has no attribute '__private'
```

Однако полностью это не защищает, так как атрибут всё равно остаётся доступным под именем _ИмяКласса__ИмяАтрибута:

```
>>>
```

```
>>> b._B__private()

Это приватный метод!
```

Наследование

Наследование подразумевает то, что дочерний класс содержит все атрибуты родительского класса, при этом некоторые из них могут быть переопределены или добавлены в дочернем. Например, мы можем создать свой класс, похожий на словарь:

>>>

```
>>> class Mydict(dict):

...     def get(self, key, default = 0):

...         return dict.get(self, key, default)

...

>>> a = dict(a=1, b=2)

>>> b = Mydict(a=1, b=2)
```

Класс Mydict ведёт себя точно так же, как и словарь, за исключением того, что метод get по умолчанию возвращает не None, а 0.

>>>

```
>>> b['c'] = 4

>>> print(b)

{'a': 1, 'c': 4, 'b': 2}

>>> print(a.get('v'))

None

>>> print(b.get('v'))
```

```
0
```

Полиморфизм

Полиморфизм - разное поведение одного и того же метода в разных классах. Например, мы можем сложить два числа, и можем сложить две строки. При этом получим разный результат, так как числа и строки являются разными классами.

```
>>>
```

```
>>> 1 + 1

2

>>> "1" + "1"

'11'
```

Перегрузка операторов — один из способов реализации полиморфизма, когда мы можем задать свою реализацию какого-либо метода в своём классе.

Например, у нас есть два класса:

```
class A:

    def go(self):

        print('Go, A!')



class B(A):

    def go(self, name):

        print('Go, {}!'.format(name))
```

В данном примере класс B наследует класс A, но переопределяет метод go, поэтому он имеет мало общего с аналогичным методом класса A.

Однако в python имеются методы, которые, как правило, не вызываются напрямую, а вызываются встроенными функциями или операторами.

Например, метод **__init__** перегружает конструктор класса. Конструктор - создание экземпляра класса.

>>>

```
>>> class A:

...     def __init__(self, name):

...         self.name = name

...

>>> a = A('Vasya')

>>> print(a.name)

Vasya
```

Собственно, далее пойдёт список таких "магических" методов.

__new__(cls[, ...]) — управляет созданием экземпляра. В качестве обязательного аргумента принимает класс (не путать с экземпляром). Должен возвращать экземпляр класса для его последующей его передачи методу __init__.

__init__(self[, ...]) - как уже было сказано выше, конструктор.

__del__(self) - вызывается при удалении объекта сборщиком мусора.

__repr__(self) - вызывается встроенной функцией repr; возвращает "сырые" данные, использующиеся для внутреннего представления в python.

__str__(self) - вызывается функциями str, print и format. Возвращает строковое представление объекта.

__bytes__(self) - вызывается функцией bytes при преобразовании к байтам.

__format__(self, format_spec) - используется функцией format (а также методом format у строк).

__lt__(self, other) - x < y вызывает x.__lt__(y).

__le__(self, other) - x ≤ y вызывает x.__le__(y).

__eq__(self, other) - x == y вызывает x.__eq__(y).

__ne__(self, other) - x != y вызывает x.__ne__(y)

__gt__(self, other) - x > y вызывает x.__gt__(y).

__ge__(self, other) - x ≥ y вызывает x.__ge__(y).

__hash__(self) - получение хэш-суммы объекта, например, для добавления в словарь.

__bool__(self) - вызывается при проверке истинности. Если этот метод не определён, вызывается метод __len__ (объекты, имеющие ненулевую длину, считаются истинными).

__getattr__(self, name) - вызывается, когда атрибут экземпляра класса не найден в обычных местах (например, у экземпляра нет метода с таким названием).

__setattr__(self, name, value) - назначение атрибута.

__delattr__(self, name) - удаление атрибута (del obj.name).

__call__(self[, args...]) - вызов экземпляра класса как функции.

__len__(self) - длина объекта.

__getitem__(self, key) - доступ по индексу (или ключу).

__setitem__(self, key, value) - назначение элемента по индексу.

__delitem__(self, key) - удаление элемента по индексу.

__iter__(self) - возвращает итератор для контейнера.

__reversed__(self) - итератор из элементов, следующих в обратном порядке.

__contains__(self, item) - проверка на принадлежность элемента контейнеру (item in self).

Перегрузка арифметических операторов

__add__(self, other) - сложение. x + y вызывает x.__add__(y).

__sub__(self, other) - вычитание (x - y).

__mul__(self, other) - умножение (x * y).

__truediv__(self, other) - деление (x / y).

__floordiv__(self, other) - целочисленное деление (x // y).

__mod__(self, other) - остаток от деления (x % y).

__divmod__(self, other) - частное и остаток (divmod(x, y)).

__pow__(self, other[, modulo]) - возведение в степень (x ** y, pow(x, y[, modulo])).

__lshift__(self, other) - битовый сдвиг влево (x << y).

__rshift__(self, other) - битовый сдвиг вправо (x >> y).

__and__(self, other) - битовое И (x & y).

__xor__(self, other) - битовое ИСКЛЮЧАЮЩЕЕ ИЛИ (x ^ y).

__or__(self, other) - битовое ИЛИ (x | y).

Пойдём дальше.

__radd__(self, other),

__rsub__(self, other),

__rmul__(self, other),

__rtruediv__(self, other),

__rfloordiv__(self, other),

__rmod__(self, other),

__rdivmod__(self, other),

__rpow__(self, other),

__rlshift__(self, other),

__rrshift__(self, other),

__rand__(self, other),

__rxor__(self, other),

__ror__(self, other) - делают то же самое, что и арифметические операторы, перечисленные выше, но для аргументов, находящихся справа, и только в случае, если для левого операнда не определён соответствующий метод.

Например, операция x + y будет сначала пытаться вызвать x.__add__(y), и только в том случае, если это не получилось, будет пытаться вызвать y.__radd__(x). Аналогично для остальных методов.

Идём дальше.

__iadd__(self, other) - +=.

__isub__(self, other) - -=.

__imul__(self, other) - *=.

__itruediv__(self, other) - /=.

__ifloordiv__(self, other) - //=.

__imod__(self, other) - %=.

__ipow__(self, other[, modulo]) - **=.

__ilshift__(self, other) - <<=.

__irshift__(self, other) - >>=.

__iand__(self, other) - &=.

__ixor__(self, other) - ^=.

__ior__(self, other) - |=.

__neg__(self) - унарный -.

__pos__(self) - унарный +.

__abs__(self) - модуль (abs()).

__invert__(self) - инверсия (~).

__complex__(self) - приведение к complex.

__int__(self) - приведение к int.

__float__(self) - приведение к float.

__round__(self[, n]) - округление.

__enter__(self), **__exit__**(self, exc_type, exc_value, traceback) - реализация менеджеров контекста.

Рассмотрим некоторые из этих методов на примере двухмерного вектора, для которого переопределим некоторые методы:

```
import math



class Vector2D:

    def __init__(self, x, y):

        self.x = x

        self.y = y



    def __repr__(self):

        return 'Vector2D({}, {})'.format(self.x, self.y)



    def __str__(self):

        return '({}, {})'.format(self.x, self.y)



    def __add__(self, other):

        return Vector2D(self.x + other.x, self.y + other.y)



    def __iadd__(self, other):

        self.x += other.x

        self.y += other.y

        return self
```

```
    def __sub__(self, other):
        return Vector2D(self.x - other.x, self.y - other.y)

    def __isub__(self, other):
        self.x -= other.x
        self.y -= other.y
        return self

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return self.x != 0 or self.y != 0

    def __neg__(self):
        return Vector2D(-self.x, -self.y)

>>> x = Vector2D(3, 4)
>>> x
```

```
Vector2D(3, 4)

>>> print(x)

(3, 4)

>>> abs(x)

5.0

>>> y = Vector2D(5, 6)

>>> y

Vector2D(5, 6)

>>> x + y

Vector2D(8, 10)

>>> x - y

Vector2D(-2, -2)

>>> -x

Vector2D(-3, -4)

>>> x += y

>>> x

Vector2D(8, 10)

>>> bool(x)

True

>>> z = Vector2D(0, 0)
```

```
>>> bool(z)

False

>>> -z

Vector2D(0, 0)
```

В заключение хочу сказать, что перегрузка специальных методов - вещь хорошая, но не стоит ей слишком злоупотреблять. Перегружайте их только тогда, когда вы уверены в том, что это поможет пониманию программного кода.

Декораторы в Python и примеры их практического использования.

Итак, что же это такое? Для того, чтобы понять, как работают декораторы, в первую очередь следует вспомнить, что функции в python являются объектами, соответственно, их можно возвращать из другой функции или передавать в качестве аргумента. Также следует помнить, что функция в python может быть определена и внутри другой функции.

Вспомнив это, можно смело переходить к декораторам. **Декораторы** — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код.

Создадим свой декоратор "вручную":

```
>>>
```

```
>>> def my_shiny_new_decorator(function_to_decorate):

...     # Внутри себя декоратор определяет функцию-"обёртку". Она будет обёрнута вокруг декорируемой,

...     # получая возможность исполнять произвольный код до и после неё.

...     def the_wrapper_around_the_original_function():

...         print("Я - код, который отработает до вызова функции")

...         function_to_decorate() # Сама функция

...         print("А я - код, срабатывающий после")

...     # Вернём эту функцию
```

```
...     return the_wrapper_around_the_original_function

...

>>> # Представим теперь, что у нас есть функция, которую мы не планируем больше трогать.

>>> def stand_alone_function():

...     print("Я простая одинокая функция, ты ведь не посмеешь меня изменять?")

...

>>> stand_alone_function()

Я простая одинокая функция, ты ведь не посмеешь меня изменять?

>>> # Однако, чтобы изменить её поведение, мы можем декорировать её, то есть просто передать декоратору,

>>> # который обернет исходную функцию в любой код, который нам потребуется, и вернёт новую,

>>> # готовую к использованию функцию:

>>> stand_alone_function_decorated = my_shiny_new_decorator(stand_alone_function)

>>> stand_alone_function_decorated()

Я - код, который отработает до вызова функции

Я простая одинокая функция, ты ведь не посмеешь меня изменять?

А я - код, срабатывающий после
```

Наверное, теперь мы бы хотели, чтобы каждый раз, во время вызова stand_alone_function, вместо неё вызывалась stand_alone_function_decorated. Для этого просто перезапишем stand_alone_function:

```
>>>
```

```
>>> stand_alone_function =
my_shiny_new_decorator(stand_alone_function)

>>> stand_alone_function()

Я - код, который отработает до вызова функции

Я простая одинокая функция, ты ведь не посмеешь меня изменять?

А я - код, срабатывающий после
```

Собственно, это и есть декораторы. Вот так можно было записать предыдущий пример, используя синтаксис декораторов:

```
>>>
```

```
>>> @my_shiny_new_decorator

... def another_stand_alone_function():

...     print("Оставь меня в покое")

...

>>> another_stand_alone_function()

Я - код, который отработает до вызова функции

Оставь меня в покое

А я - код, срабатывающий после
```

То есть, декораторы в python — это просто синтаксический сахар для конструкций вида:

```
another_stand_alone_function =
my_shiny_new_decorator(another_stand_alone_function)
```

При этом, естественно, можно использовать несколько декораторов для одной функции, например так:

```
>>>
```

```
>>> def bread(func):
...     def wrapper():
...         print()
...         func()
...         print("<\______/>")
...     return wrapper
...
>>> def ingredients(func):
...     def wrapper():
...         print("#помидоры#")
...         func()
...         print("~салат~")
...     return wrapper
...
>>> def sandwich(food="--ветчина--"):
...     print(food)
...
>>> sandwich()
--ветчина--
>>> sandwich = bread(ingredients(sandwich))
```

```
>>> sandwich()



#помидоры#

--ветчина--

~салат~

<\______/>
```

И используя синтаксис декораторов:

```
>>>
```

```
>>> @bread

... @ingredients

... def sandwich(food="--ветчина--"):

...     print(food)

...

>>> sandwich()



#помидоры#

--ветчина--

~салат~

<\______/>
```

Также нужно помнить о том, что важен порядок декорирования. Сравните с предыдущим примером:

```
>>>
```

```
>>> @ingredients
... @bread
... def sandwich(food="--ветчина--"):
...     print(food)
...
>>> sandwich()
#помидоры#



--ветчина--

<\_____/>

~салат~
```

Передача декоратором аргументов в функцию

Однако, все декораторы, которые мы рассматривали, не имели одного очень важного функционала — передачи аргументов декорируемой функции. Собственно, это тоже несложно сделать.

```
>>>
```

```
>>> def a_decorator_passing_arguments(function_to_decorate):
...     def a_wrapper_accepting_arguments(arg1, arg2):
...         print("Смотри, что я получил:", arg1, arg2)
...         function_to_decorate(arg1, arg2)
...     return a_wrapper_accepting_arguments
```

```
...

>>> # Теперь, когда мы вызываем функцию, которую возвращает
декоратор, мы вызываем её "обёртку",

>>> # передаём ей аргументы и уже в свою очередь она передаёт их
декорируемой функции

>>> @a_decorator_passing_arguments

... def print_full_name(first_name, last_name):

...     print("Меня зовут", first_name, last_name)

...

>>> print_full_name("Vasya", "Pupkin")

Смотри, что я получил: Vasya Pupkin

Меня зовут Vasya Pupkin
```

Декорирование методов

Один из важных фактов, которые следует понимать, заключается в том, что функции и методы в Python — это практически одно и то же, за исключением того, что методы всегда ожидают первым параметром ссылку на сам объект (self). Это значит, что мы можем создавать декораторы для методов точно так же, как и для функций, просто не забывая про self.

```
>>>
```

```
>>> def method_friendly_decorator(method_to_decorate):

...     def wrapper(self, lie):

...         lie -= 3

...         return method_to_decorate(self, lie)

...     return wrapper
```

```
...

>>> class Lucy:

...     def __init__(self):

...         self.age = 32

...     @method_friendly_decorator

...     def sayYourAge(self, lie):

...         print("Мне {} лет, а ты бы сколько
дал?".format(self.age + lie))

...

>>> l = Lucy()

>>> l.sayYourAge(-3)

Мне 26 лет, а ты бы сколько дал?
```

Конечно, если мы создаём максимально общий декоратор и хотим, чтобы его можно было применить к любой функции или методу, то можно воспользоваться распаковкой аргументов:

```
>>>
```

```
>>> def
a_decorator_passing_arbitrary_arguments(function_to_decorate):

...     # Данная "обёртка" принимает любые аргументы

...     def a_wrapper_accepting_arbitrary_arguments(*args,
**kwargs):

...         print("Передали ли мне что-нибудь?:")

...         print(args)

...         print(kwargs)
```

```
...         function_to_decorate(*args, **kwargs)
...     return a_wrapper_accepting_arbitrary_arguments
...
>>> @a_decorator_passing_arbitrary_arguments
... def function_with_no_argument():
...     print("Python is cool, no argument here.")
...
>>> function_with_no_argument()
Передали ли мне что-нибудь?:
()
{}
Python is cool, no argument here.
>>> @a_decorator_passing_arbitrary_arguments
... def function_with_arguments(a, b, c):
...     print(a, b, c)
...
>>> function_with_arguments(1, 2, 3)
Передали ли мне что-нибудь?:
(1, 2, 3)
{}
```

```
1 2 3

>>> @a_decorator_passing_arbitrary_arguments

... def function_with_named_arguments(a, b, c, platypus="Почему
нет?"):

...     print("Любят ли {}, {} и {} утконосов? {}".format(a, b, c,
platypus))

...

>>> function_with_named_arguments("Билл", "Линус", "Стив",
platypus="Определенно!")

Передали ли мне что-нибудь?:

('Билл', 'Линус', 'Стив')

{'platypus': 'Определенно!'}

Любят ли Билл, Линус и Стив утконосов? Определенно!

>>> class Mary(object):

...     def __init__(self):

...         self.age = 31

...     @a_decorator_passing_arbitrary_arguments

...     def sayYourAge(self, lie=-3): # Теперь мы можем указать
значение по умолчанию

...         print("Мне {} лет, а ты бы сколько
дал?".format(self.age + lie))

...

>>> m = Mary()
```

```
>>> m.sayYourAge()

Передали ли мне что-нибудь?:

(<__main__.Mary object at 0x7f6373017780>,)

{}

Мне 28 лет, а ты бы сколько дал?
```

Декораторы с аргументами

А теперь попробуем написать декоратор, принимающий аргументы:

>>>

```
>>> def decorator_maker():

...     print("Я создаю декораторы! Я буду вызван только раз: когда
ты попросишь меня создать декоратор.")

...     def my_decorator(func):

...         print("Я - декоратор! Я буду вызван только раз: в
момент декорирования функции.")

...         def wrapped():

...             print ("Я - обёртка вокруг декорируемой функции.\n"

...                    "Я буду вызвана каждый раз, когда ты
вызываешь декорируемую функцию.\n"

...                    "Я возвращаю результат работы декорируемой
функции.")

...             return func()

...         print("Я возвращаю обёрнутую функцию.")

...         return wrapped
```

```
...     print("Я возвращаю декоратор.")

...     return my_decorator

...

>>> # Давайте теперь создадим декоратор. Это всего лишь ещё один
вызов функции

>>> new_decorator = decorator_maker()

Я создаю декораторы! Я буду вызван только раз: когда ты попросишь
меня создать декоратор.

Я возвращаю декоратор.

>>>

>>> # Теперь декорируем функцию

>>> def decorated_function():

...     print("Я - декорируемая функция.")

...

>>> decorated_function = new_decorator(decorated_function)

Я - декоратор! Я буду вызван только раз: в момент декорирования
функции.

Я возвращаю обёрнутую функцию.

>>> # Теперь наконец вызовем функцию:

>>> decorated_function()

Я - обёртка вокруг декорируемой функции.

Я буду вызвана каждый раз, когда ты вызываешь декорируемую функцию.
```

```
Я возвращаю результат работы декорируемой функции.


Я - декорируемая функция.
```

Теперь перепишем данный код с помощью декораторов:

```
>>>
```

```
>>> @decorator_maker()

... def decorated_function():

...     print("Я - декорируемая функция.")

...

Я создаю декораторы! Я буду вызван только раз: когда ты попросишь
меня создать декоратор.

Я возвращаю декоратор.

Я - декоратор! Я буду вызван только раз: в момент декорирования
функции.

Я возвращаю обёрнутую функцию.

>>> decorated_function()

Я - обёртка вокруг декорируемой функции.

Я буду вызвана каждый раз когда ты вызываешь декорируемую функцию.

Я возвращаю результат работы декорируемой функции.

Я - декорируемая функция.
```

Вернёмся к аргументам декораторов, ведь, если мы используем функцию, чтобы создавать декораторы "на лету", мы можем передавать ей любые аргументы, верно?

```
>>>
```

```
>>> def decorator_maker_with_arguments(decorator_arg1,
decorator_arg2):

...     print("Я создаю декораторы! И я получил следующие
аргументы:",

...           decorator_arg1, decorator_arg2)

...     def my_decorator(func):

...         print("Я - декоратор. И ты всё же смог передать мне эти
аргументы:",

...               decorator_arg1, decorator_arg2)

...         # Не перепутайте аргументы декораторов с аргументами
функций!

...         def wrapped(function_arg1, function_arg2):

...             print ("Я - обёртка вокруг декорируемой функции.\n"

...                    "И я имею доступ ко всем аргументам\n"

...                    "\t- и декоратора: {0} {1}\n"

...                    "\t- и функции: {2} {3}\n"

...                    "Теперь я могу передать нужные аргументы
дальше"

...                    .format(decorator_arg1, decorator_arg2,

...                            function_arg1, function_arg2))

...             return func(function_arg1, function_arg2)

...         return wrapped

...     return my_decorator
```

```
...

>>> @decorator_maker_with_arguments("Леонард", "Шелдон")

... def decorated_function_with_arguments(function_arg1,
function_arg2):

...     print ("Я - декорируемая функция и я знаю только о своих
аргументах: {0}"

...            " {1}".format(function_arg1, function_arg2))

...

Я создаю декораторы! И я получил следующие аргументы: Леонард
Шелдон

Я - декоратор. И ты всё же смог передать мне эти аргументы: Леонард
Шелдон

>>> decorated_function_with_arguments("Раджеш", "Говард")

Я - обёртка вокруг декорируемой функции.

И я имею доступ ко всем аргументам

    - и декоратора: Леонард Шелдон

    - и функции: Раджеш Говард

Теперь я могу передать нужные аргументы дальше

Я - декорируемая функция и я знаю только о своих аргументах: Раджеш
Говард
```

Таким образом, мы можем передавать декоратору любые аргументы, как обычной функции. Мы можем использовать и распаковку через *args и **kwargs в случае необходимости.

Некоторые особенности работы с декораторами

- Декораторы несколько замедляют вызов функции, не забывайте об этом.

- Вы не можете "раздекорировать" функцию. Безусловно, существуют трюки, позволяющие создать декоратор, который можно отсоединить от функции, но это плохая практика. Правильнее будет запомнить, что если функция декорирована — это не отменить.
- Декораторы оборачивают функции, что 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.

  Забавным фактом является то, что functools.wraps тоже является декоратором.

  >>>

```
>>> def foo():
...     print("foo")
...
>>> print(foo.__name__)
foo
>>> # Однако, декораторы мешают нормальному ходу дел:
... def bar(func):
...     def wrapper():
...         print("bar")
...         return func()
...     return wrapper
...
>>> @bar
... def foo():
...     print("foo")
...
```

```
>>> print(foo.__name__)

wrapper

>>> import functools  # "functools" может нам с этим помочь

>>> def bar(func):

...     # Объявляем "wrapper" оборачивающим "func"

...     # и запускаем магию:

...     @functools.wraps(func)

...     def wrapper():

...         print("bar")

...         return func()

...     return wrapper

...

>>> @bar

... def foo():

...     print("foo")

...

>>> print(foo.__name__)

foo
```

Примеры использования декораторов

Декораторы могут быть использованы для расширения возможностей функций из сторонних библиотек (код которых мы не можем изменять), или для упрощения отладки (мы не хотим изменять код, который ещё не устоялся).

Также полезно использовать декораторы для расширения различных функций одним и тем же кодом, без повторного его переписывания каждый раз, например:

```
>>>
```

```
>>> def benchmark(func):
...     """
...     Декоратор, выводящий время, которое заняло
...     выполнение декорируемой функции.
...     """
...     import time
...     def wrapper(*args, **kwargs):
...         t = time.clock()
...         res = func(*args, **kwargs)
...         print(func.__name__, time.clock() - t)
...         return res
...     return wrapper
...
>>> def logging(func):
...     """
...     Декоратор, логирующий работу кода.
...     (хорошо, он просто выводит вызовы, но тут могло быть и
логирование!)
...     """
```

```
...     def wrapper(*args, **kwargs):
...         res = func(*args, **kwargs)
...         print(func.__name__, args, kwargs)
...         return res
...     return wrapper
...
>>> def counter(func):
...     """
...     Декоратор, считающий и выводящий количество вызовов
...     декорируемой функции.
...     """
...     def wrapper(*args, **kwargs):
...         wrapper.count += 1
...         res = func(*args, **kwargs)
...         print("{0} была вызвана: {1}x".format(func.__name__,
wrapper.count))
...         return res
...     wrapper.count = 0
...     return wrapper
...
>>> @benchmark
```

```
... @logging

... @counter

... def reverse_string(string):

...     return ''.join(reversed(string))

...

>>> print(reverse_string("А роза упала на лапу Азора"))

reverse_string была вызвана: 1х

wrapper ('А роза упала на лапу Азора',) {}

wrapper 0.00011799999999997923

арозА упал ан алапу азор А

>>> print(reverse_string("A man, a plan, a canoe, pasta, heros,
rajahs, a coloratura,"

... "maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a
tag,"

... "a banana bag again (or a camel), a crepe, pins, Spam, a rut, a
Rolo, cash,"

... "a jar, sore hats, a peon, a canal: Panama!"))

reverse_string была вызвана: 2х

wrapper ('A man, a plan, a canoe, pasta, heros, rajahs, a
coloratura,maps, snipe, ...',) {}

wrapper 0.00017800000000001148

!amanaP :lanac a ,noep a ,stah eros ,raj a,hsac ,oloR a ,tur a
,mapS ,snip ,eperc a , .
```