ООП-2

Python

ООП. Иерархия типов

Python

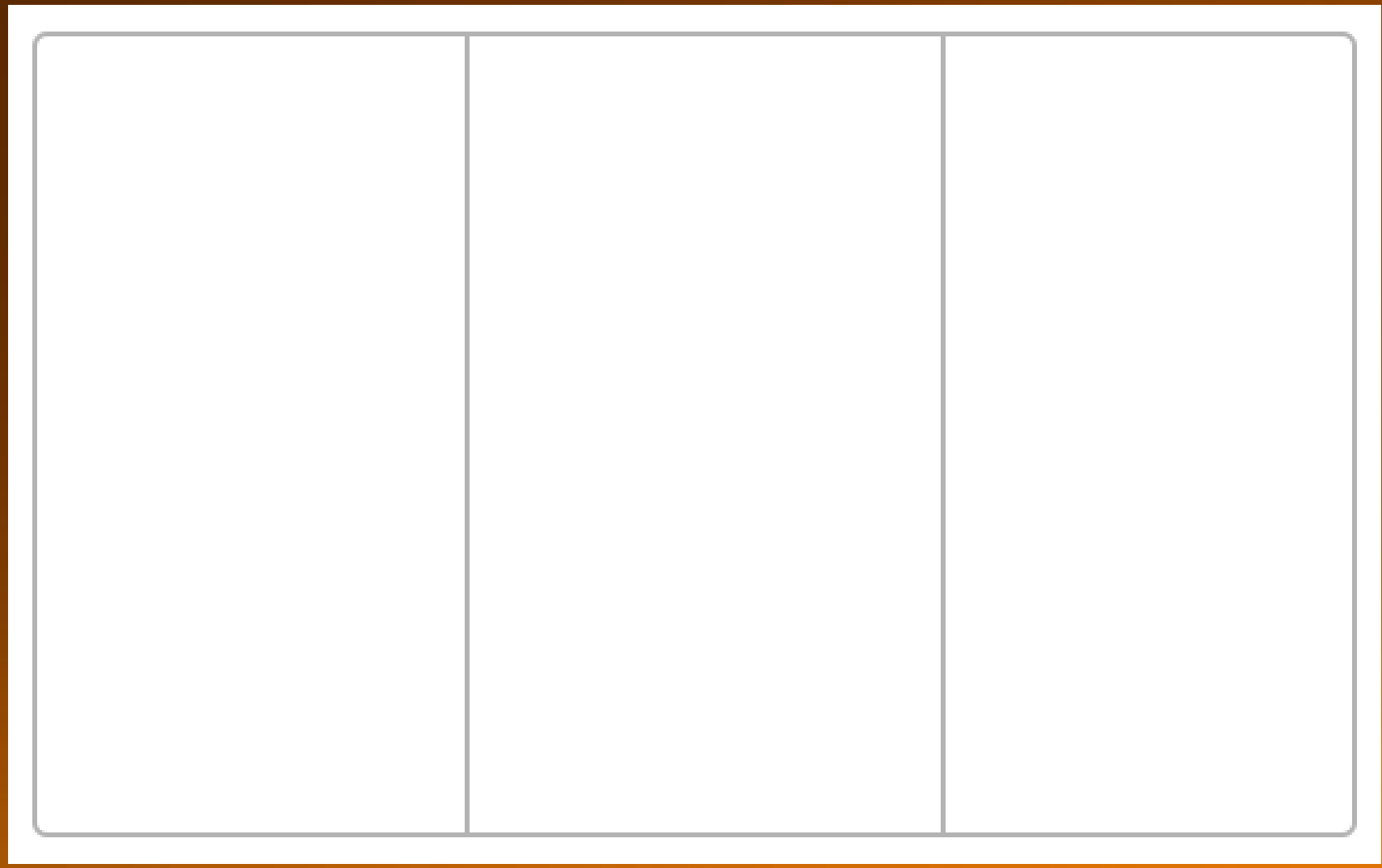
Пример про int...

```
 >>> two = 2 #1
 >>> type(two)
<type 'int'> #2
 >>> type(type(two))
<type 'type'> #3
 >>> type(two).__bases__
(<type 'object'>,) #4
 >>> dir(two) #5
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
 '__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
 '__floordiv__', '__format__', '__getattribute__', '__getnewargs__',
 '__hash__', '__hex__', '__index__', '__init__', '__int__', '__invert__',
 '__long__', '__lshift__', '__mod__', '__mul__', '__neg__', '__new__',
 '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
 '__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
 '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
 '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
 '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
 'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

Первое правило Python ООП

- ~~Никому не говорить про Python ООП~~
- Все объект!
- «Объекты» хранят ссылку на свой класс __class__

Начало

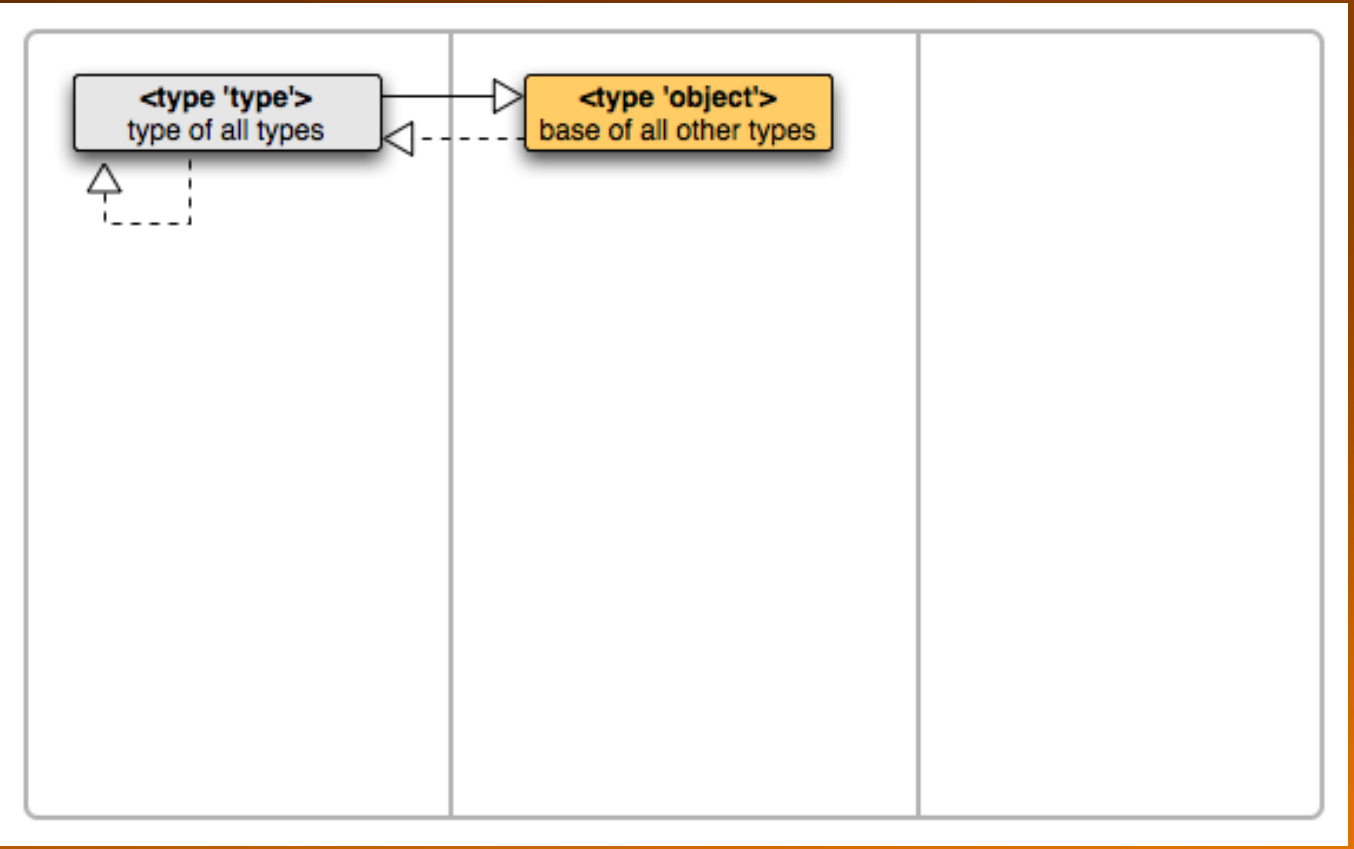


Курицы и яйца...

```
 >>> object
<type 'object'>
 >>> type
<type 'type'>
 >>> type(object)
<type 'type'>
 >>> object.__class__
<type 'type'>
 >>> object.__bases__
()
 >>> type.__class__
<type 'type'>
 >>> type.__bases__
(<type 'object'>,)
```

Object & type

Объекты object и type являются базовыми в языке Python.
Исходя из предыдущего эксперимента, можно построить диаграмму отношений между ними.



Объекты-типы

Продолжим эксперименты:

```
>>> isinstance(object, object)
True
>>> isinstance(type, object)
True
```

Объекты object и type - это объекты-типы в языке Python.

Это означает, что

- они могут представлять абстрактные типы данных в программе;
- они могут быть унаследованы другими объектами;
- можно создавать их экземпляры;
- типом любого объекта-типа является <type 'type'>;
- одни их называют типами, другие - классами.

Второе правило

- ~~Никому не говорить про Python ООП~~
- **Class is Type is Class**

- **Type Or Non-type Test Rule**

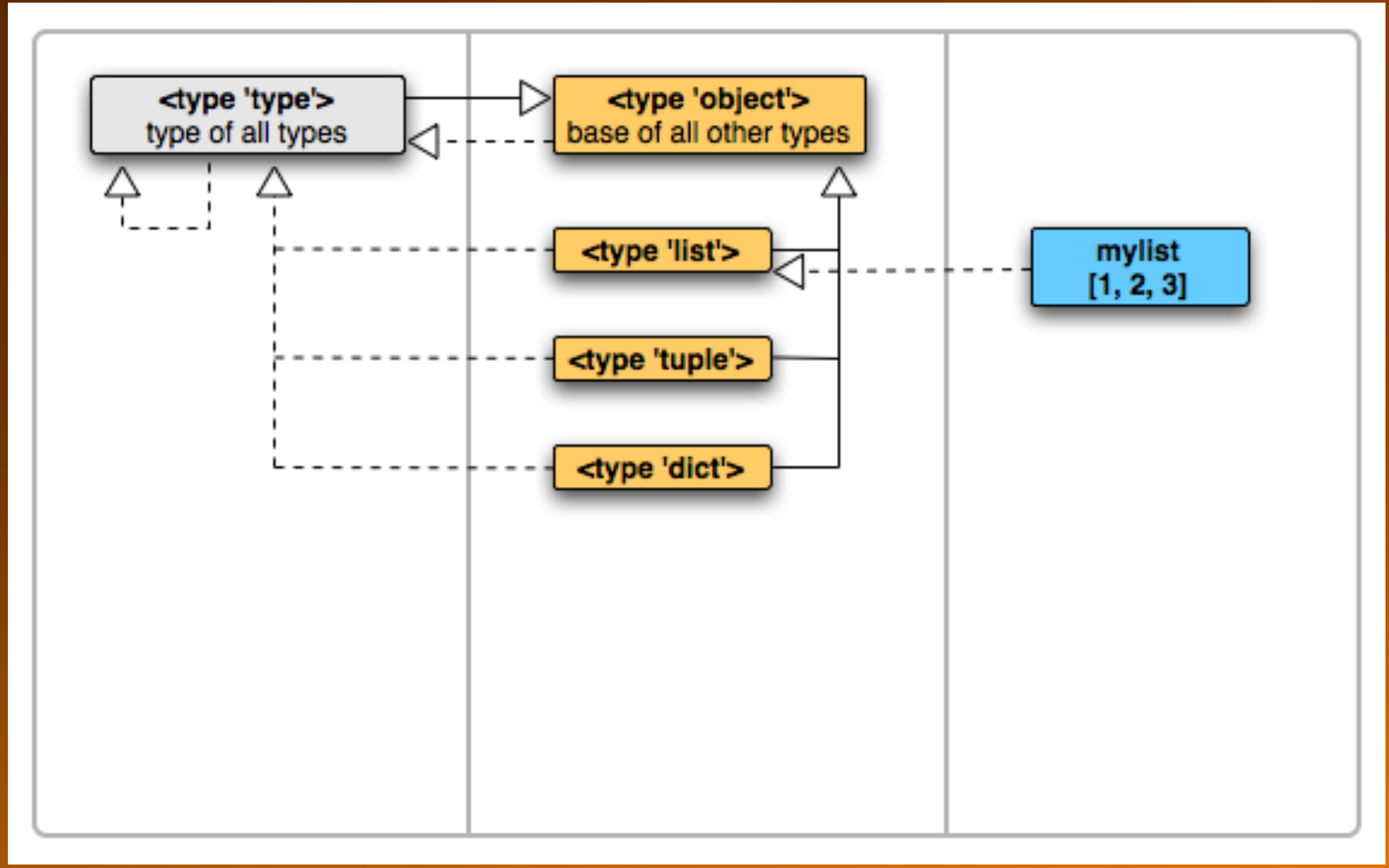
If an object is an instance of <type 'type'>, then it is a type. Otherwise, it is not a type.

Встроенные типы

```
>>> list
<type 'list'>
>>> list.__class__
<type 'type'>
>>> list.__bases__
(<type 'object'>,)
>>> tuple.__class__, tuple.__bases__
(<type 'type'>, (<type 'object'>,))
>>> dict.__class__, dict.__bases__
(<type 'type'>, (<type 'object'>,))

>>> mylist = [1,2,3]
>>> mylist.__class__
<type 'list'>
```

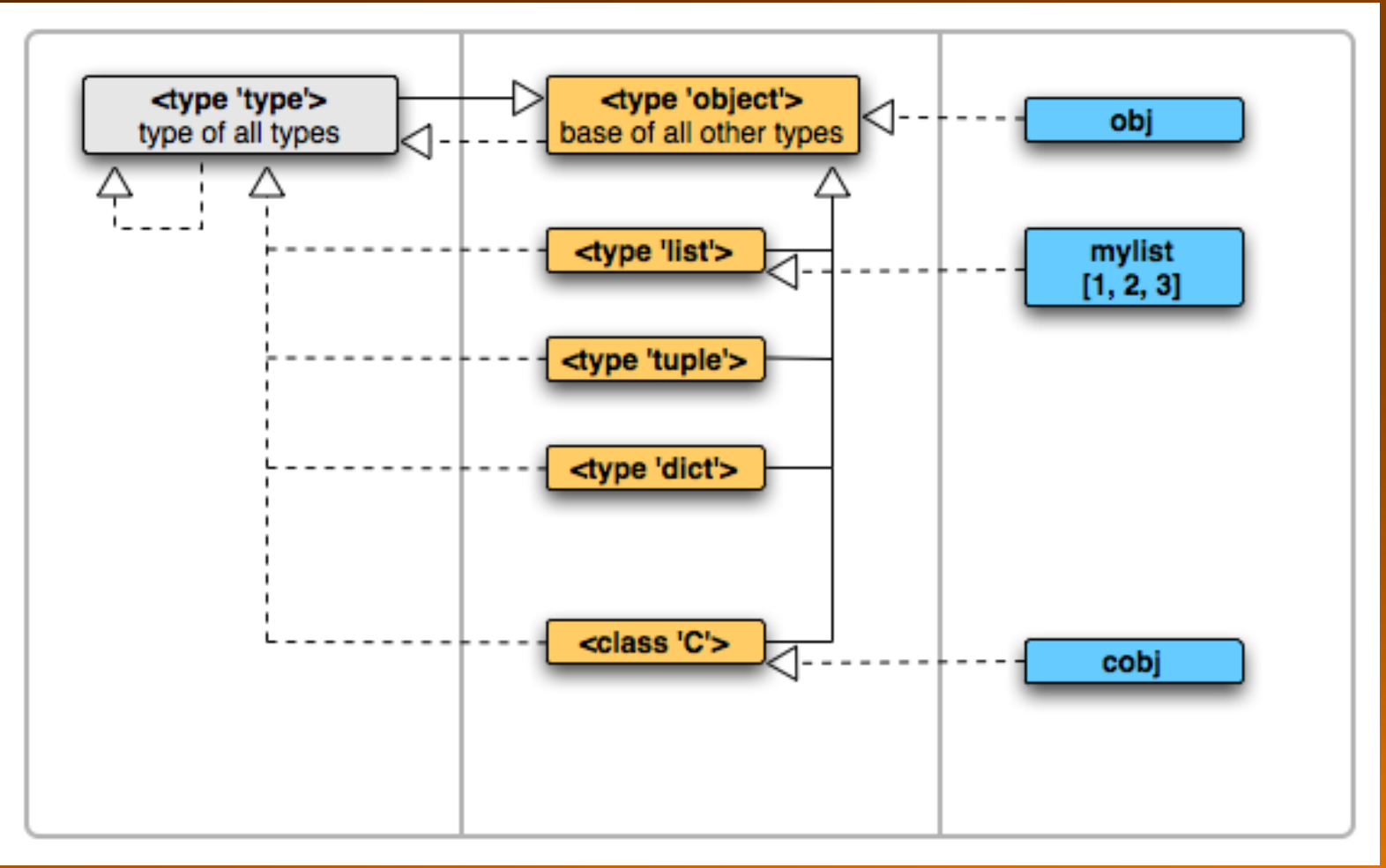
Диаграмма типов



Пользовательские типы

```
class New:
    pass
 new = New()
>>> type(new)
<class '__main__.New'>
>>> type(New)
<type 'type'>
```

Диаграмма типов



Инстанцирование

- Как Python на самом деле создает объекты?

При вызове a = A()…

- вызывается метод __call__ объекта-класса A, который
- вызывает __new__ и __init__ класса-предка A

- Но откуда он это знает???

- Потому что __call__ берется из type(A)

- А можно чтобы он брался из другого места?

Да. Метаклассы

Метаклассы. Нано-введение

Хотим:

```
class Man(object):
        pass
>>> me = Man(height = 180, weight = 80)
Traceback (most recent call last):
File "<stdin>", line 20, in <module>
      TypeError: object.__new__() takes no
parameters
```

Метаклассы. Нано-введение

```
class AttributeInitType(type):
        def __call__(self, *args, **kwargs):
                obj = type.__call__(self, *args)
                # добавим ему переданные в вызове аргументы в качестве атрибутов.
                for name in kwargs:
                        setattr(obj, name, kwargs[name])
                # вернем готовый объект
                return obj

class Man(object):
        __metaclass__ = AttributeInitType

>>>me = Man(height = 180, weigth = 80)
>>>print(me.height)
180
```

Декораторы (Аннотации)

Python

Что это?

```
def makebold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

def makeitalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello world"

print hello() ## выведет <b><i>hello world</i></b>
```

Что нужно понимать

- Функция – объект
- Его можно возвращать, хранить на него ссылку, передавать и т.д.

```
def talk():
        def whisper(word="да"):
            return word.lower()+"...";

        return whisper

foo = talk()
foo(“Мама”)
```

Декоратор - это функция, ожидающая ДРУГУЮ функцию в качестве параметра. Внутри себя декоратор определяет функцию-"обёртку". Она будет обёрнута вокруг декорируемой, исполняя произвольный код до и после неё.

```
def my_shiny_new_decorator(a_function_to_decorate):
    def the_wrapper_around_the_original_function():
        # код, который мы хотим запускать ДО вызова оригинальной функции
            print("Я - код, который отработает до вызова функции“)
        # ВЫЗОВЕМ саму декорируемую функцию
            a_function_to_decorate()
        # код, который мы хотим запускать ПОСЛЕ вызова оригинальной функции
             print("А я - код, срабатывающий после“)

    # Теперь, вернём функцию-обёртку, которая содержит в себе декорируемую функцию, и код, который необходимо выполнить до и после.
    return the_wrapper_around_the_original_function
```

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

```
def a_alone_function():
    print("Я простая функция, ты не посмеешь меня изменять?..“)

a_alone_function()
# выведет: Я простая функция, ты 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:

```
a_alone_function_decorated = my_shiny_new_decorator(a_alone_function)
a_alone_function_decorated()
```

выведет:

```
 Я - код, который отработает до вызова функции
 Я простая функция, ты не посмеешь меня изменять?..
 А я - код, срабатывающий после
```

Почти декторатоpы

```
a_alone_function = my_shiny_new_decorator(a_alone_function)
a_alone_function()
```

```
@my_shiny_new_decorator – это синтаксический сахар для
a_alone_function = my_shiny_new_decorator(a_alone_function)
```

Декораторы

```
def bread(func):
    def wrapper():
        print "</------\>"
        func()
        print "<\______/>"
    return wrapper

def ingredients(func):
    def wrapper():
        print "#помидоры#"
        func()
return wrapper

def sandwich(food="--ветчина--"):
    print(food)

sandwich = bread(ingredients(sandwich))
sandwich()

# </------\>
# #помидоры#
# --ветчина--
# <\______/>
```

Декораторы

```
def bread(func):
    def wrapper():
        print "</------\>"
        func()
        print "<\______/>"
    return wrapper

def ingredients(func):
    def wrapper():
        print "#помидоры#"
        func()
return wrapper

@bread
@ingredients
def sandwich(food="--ветчина--"):
    print(food)
```

Порядок декорирования важен!

Декораторы

Все декораторы, которые мы до этого рассматривали не имели одного очень важного функционала — передачи аргументов декорируемой функции.

Передача параметров

```
def a_decorator_passing_arguments(func):
    def a_wrapper (arg1, arg2): # аргументы прибывают отсюда
        print("Смотри, что я получил:", arg1, arg2)
        func (arg1, arg2)
    return a_wrapper

@a_decorator_passing_arguments
def print_full_name(first_name, last_name):
    print("Меня зовут", first_name, last_name)

print_full_name("Александр", "Омельченко")
# Смотри, что я получил: Александр Омельченко
# Меня зовут Александр Омельченко
```

Декорирование методов

- То же самое что и функции, но первый параметр self

```
def women_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie = lie - 3
        return method_to_decorate(self, lie)
    return wrapper

class Lucy(object):
    def __init__(self):
        self.age = 32

    @women_decorator
    def sayYourAge(self, lie):
        print("Мне %s, а ты бы сколько дал?" % (self.age + lie))

>>>  l = Lucy()
>>>  l.sayYourAge(-3)
```

Фабрика декораторов

```
def decorator_maker():
    print(“Magic!“)
    def my_decorator(func):
        print(“I am decorator”)
        def wrapped():
            print (“Wrapper")
            return func()
        print(“Return”)
        return wrapped
    print(“Return decorator”)
    return my_decorator

new_decorator = decorator_maker()
# Magic
# Return decorator
def decorated_function():
    print(“Function”)
decorated_function = new_decorator(decorated_function)
# I am decorator
# Return
decorated_function()
# Wrapper
# Function
```

Фабрика декораторов

```
def decorator_maker():
    print(“Magic!“)
    def my_decorator(func):
        print(“I am decorator”)
        def wrapped():
            print (“Wrapper")
            return func()
        print(“Return”)
        return wrapped
    print(“Return decorator”)
    return my_decorator

def decorated_function():
    print(“Function”)
decorated_function = decorator_maker()(decorated_function)
decorated_function()
```

Фабрика декораторов

```
def decorator_maker():
    print(“Magic!“)
    def my_decorator(func):
        print(“I am decorator”)
        def wrapped():
            print (“Wrapper")
            return func()
        print(“Return”)
        return wrapped
    print(“Return decorator”)
    return my_decorator

@decorator_maker()
def decorated_function():
    print(“Function”)
```

Фабрика декораторов

```
def decorator_maker(arg1, arg2):
    print(“Magic!“ + arg1 + arg2)
    def my_decorator(func):
        print(“I am decorator”)
        def wrapped():
            print (“Wrapper“ + arg1)
            return func()
        print(“Return”)
        return wrapped
    print(“Return decorator”)
    return my_decorator

@decorator_maker(“Mama”, “Papa”)
def decorated_function():
    print(“Function”)

>>> decorated_function()
#Wrapper Mama
#Function
```