Объектно-ориентированное программирование на Python

Осипов Никита Алексеевич

ЛЕКЦИЯ 7. МЕТАКЛАССЫ

Учебные вопросы:

1. Декораторы и их применение.
2. Понятие метакласса.
3. Динамическое создание классов.

Изучение метаклассов

- Цель лекции:
  - изучить метаклассы и возможности их применения
- Для понимания метаклассов требуется знать и уметь применять:
  - Концепции ООП в Python
  - Декораторы (Decorators) в Python

Функции как объекты

DecorFun00.py

- В Python функции являются первыми объектами класса:
  - Функции являются объектами – на них можно ссылаться, передавать в переменную и возвращать из других функций
  - Функции могут быть определены внутри другой функции, а также могут быть переданы в качестве аргумента другой функции

```
def messageWithWelcome(str):
    # Вложенная функция
    def addWelcome():
        return "Welcome to "
    # Возвращаем конкатенацию
    return  addWelcome() + str
```

```
def site(site_name):
    return site_name
```

```
strm = messageWithWelcome(site("ITMO"))
```

Декорирование – способ управления функциями и классами

- Декораторы — это “обёртки”, которые дают возможность *изменить* поведение функции, не изменяя её код
- Декораторы в языке Python имеют две разновидности:
  - Декораторы функций связывают имя функции с другим вызываемым объектом на этапе определения функции, добавляя дополнительный уровень логики, которая управляет функциями и методами или выполняет некоторые действия в случае их вызова.
  - Декораторы классов связывают имя класса с другим вызываемым объектом на этапе его определения, добавляя дополнительный уровень логики, которая управляет классами или экземплярами, созданными при обращении к этим классам

Декорация является просто способом запуска добавочных шагов обработки на стадии определения функций и классов с помощью явного синтаксиса

DecorFun01.py

Decorator01.py

Декораторы функций

- Декоратор — это функция, которая принимает функцию в качестве единственного параметра и возвращает функцию
  - Это полезно, чтобы «обернуть» функциональность одним и тем же кодом снова и снова – декораторы Python — это мощный инструмент для удаления избыточности

```
def messageWithWelcome(fstr):
    # Вложенная функция
    def addWelcome(str_name):
        return "Welcome to " + fstr(str_name)
    # Возвращаем функцию
    return  addWelcome
```

Используем **@func_name**, чтобы указать **декоратор**, который будет применен к другой функции

```
@messageWithWelcome
def site(site_name):
    return site_name
```

```
strm = site("ITMO")
```

Декораторы функций

DecorFun01.py

Decorator01.py

Decorator_hello.py

- Порядок использования – Декоратор
  - указывается в отдельной строке непосредственно перед инструкцией **def**, определяющей функцию или метод
  - состоит из символа @, за которым следует **имя метафункции** – функции (или другого вызываемого объекта), управляющей другой функцией

```
@decorator
def F(arg):
    ...
```

Используем **@func_name**, чтобы указать **декоратор**, который будет применен к другой функции

```
F(99) # Вызов функции
```

Декораторы можно вкладывать друг в друга

Decorator02.py

Декораторы методов

Decorator03.py

- Можно создавать декораторы для методов так же, как и для функций, учитывая параметр **self**

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie = lie * 0.13
        return method_to_decorate(self, lie)
    return wrapper
```

```
class Lucy():
    def __init__(self):
        self.cost = 30

    @method_friendly_decorator
    def sayYourCost(self, lie):
        print("Сейчас %s" % (self.cost + lie))
```

```
lu = Lucy()
lu.sayYourCost(15)
```

Общий декоратор

Decorator04.py

- Для создания общего декоратора (можно было применить его к любой функции) можно воспользоваться тем, что *args распаковывает список args, а **kwargs распаковывает словарь kwargs:

```
def decorator_pass_arbitrary_arguments(function_to_decorate):
    # Данная "обёртка" принимает любые аргументы
    def wrapper_accepting_arbitrary_arguments(*args, **kwargs):
        . . .

        # Теперь распакуем *args и **kwargs
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments
```

Декоратор с различными аргументами

Decorator05.py

- Для создания общего декоратора (можно было применить его к любой функции) можно воспользоваться тем, что *args распаковывает список args, а **kwargs распаковывает словарь kwargs:

```
def decorator_with_arguments(decorator_arg1, decorator_arg2):
    def my_decorator(func):
        print("декоратор, получил аргументы:", decorator_arg1,
decorator_arg2)
 # Не перепутайте аргументы декораторов с аргументами функций!
        def wrapped(function_arg1, function_arg2) :
            print ("".format(decorator_arg1, decorator_arg2,
                             function_arg1, function_arg2))
            return func(function_arg1, function_arg2)
         return wrapped
     return my_decorator
```

functools.wraps

decorator03wraps.py

- Декоратор, дополняющий функцию-обёртку, данными из некоторых атрибутов оборачиваемой функции

```
from functools import wraps

def a_decorator_passing_arguments(function_to_decorate):
    @wraps(function_to_decorate)
    def a_wrapper_accepting_arguments(arg1, arg2):
        print("Получено:", arg1, arg2)
        function_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments
```

Применяя декоратор «wraps» ко внутренней функции, копируем имя, строку документации и сигнатуру функции во внутреннюю функцию

Особенности применения

- Декораторы были введены в Python 2.4.
- Декораторы несколько замедляют вызов функции, не забывайте об этом.
- Вы не можете «раздекорировать» функцию. Правильный стиль: если функция декорирована — это не отменить.
- Декораторы оборачивают функции, что может затруднить отладку

Декораторы класса

DecoratorClass01.py

- В данном случае декоратор принимает на вход класс (объект с типом **type** в Python) и возвращает модифицированный класс

```
def log_method_calls(time_format):
    def decorator(cls):
        for o in dir(cls):
            if o.startswith('__'):
                continue
            a = getattr(cls, o)
            if hasattr(a, '__call__'):
                decorated_a = logged(time_format, cls.__name__ + ".")(a)
                setattr(cls, o, decorated_a)
        return cls
    return decorator
```

```
@log_method_calls("%b %d %Y - %H:%M:%S")
class A():
    def test1(self):
        print("test1")
```

Классы как объекты

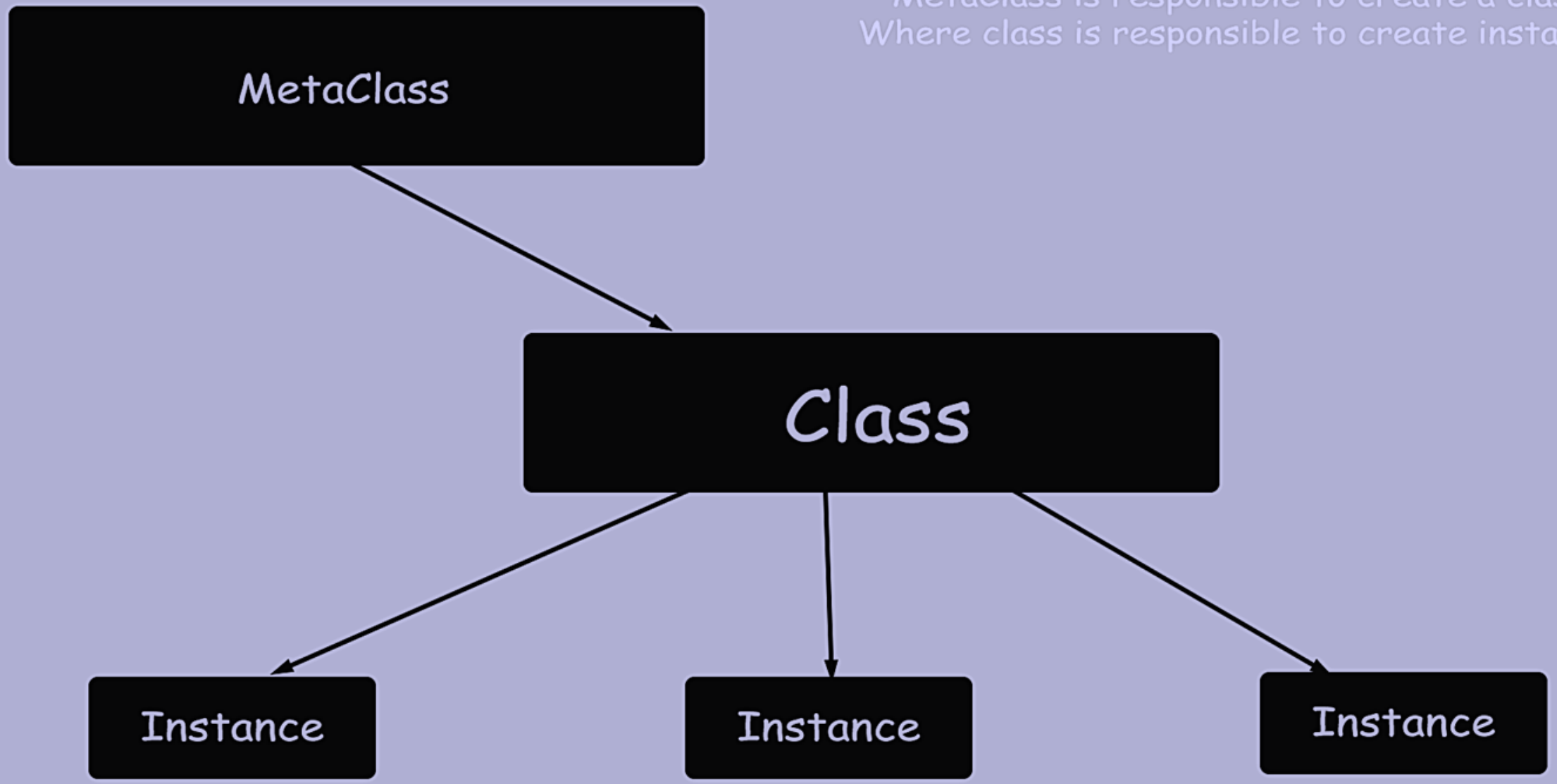
Metaclass00.py

- Классы в Python это объекты

```
type(Person)    # Тип класса Person: <class 'type'>
```

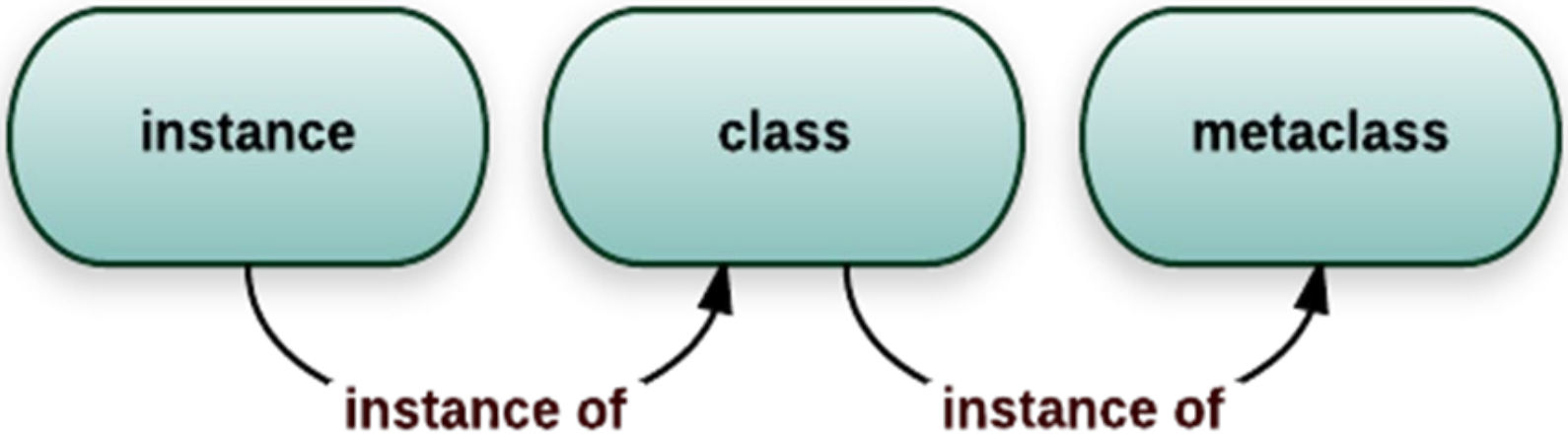
- Объект:
  - его можно назначить в качестве переменной
  - копируется
  - есть возможность добавить к нему атрибуты
  - передается в роли параметра функции

MetaClass is responsible to create a class
Where class is responsible to create instances



Понятие метакласса

- Метакласс (Metaclass) — это класс, экземпляры которого в свою очередь являются классами
  - Метаклассы являются подклассами объекта **type** и реализуют операции создания классов



- Метакласс отвечает за генерацию классов, поэтому можно написать свои собственные метаклассы, чтобы изменить способ генерации классов путем выполнения дополнительных действий или внедрения кода

Динамическое создание классов

Metaclass01.py

- Если классы в Python – это объекты, значит, как и любой другой объект, их можно создавать на ходу

Создание класса в функции:

```
def choose_class(name):
    if name == 'foo':
        class Foo():
            pass
        return Foo    # возвращается класс
    else:
        class Bar():
            pass
        return Bar    # возвращается класс
```

Динамическое создание классов

Metaclass01.py

Metaclass02.py

- Функция **type** может создавать классы на ходу
- В качестве параметра **type** принимает описание класса, и возвращает класс:

```
type(name of the class,
     tuple of the parent class (inheritance, can be empty),
     dictionary containing attributes names and values)
```

```
Person = type("Person", (), {})
```

```
class Person():
    pass
```

Динамическое создание классов

Metaclass01.py

Metaclass02.py

- Функция **type** может создавать классы на ходу.
- В качестве параметра **type** принимает описание класса, и возвращает класс:

```
type(name of the class,
     tuple of the parent class (inheritance, can be empty),
     dictionary containing attributes names and values)
```

```
class Base:
    def myfun(self):
        print("This is inherited method!")
```

```
Test = type('Test', (Base, ), dict(x="atul",
my_method=test_method))
```

Динамическое создание классов

Metaclass02b.py

- Можно создавать свои метаклассы: любой вызываемый объект, который способен принять три параметра и вернуть объект класса
  - Такие метаклассы можно применять к классу
- Метакласс можно указать при объявлении класса:

```
def my_metaclass(name, parents, attributes):   # объект - метакласс
    return 'Hello'

""" C будет переменной, которая указывает на строку 'Hello' """
class C(metaclass=my_metaclass):
    pass
```

Динамическое создание классов

Metaclass02c.py

- Основной целью метакласса является автоматическое изменение класса во время его создания
  - Например, для API, когда нужно создать классы, соответствующие текущему контексту

```
class UpperAttrMetaclass(type):

    def __new__(cls, clsname, bases, dct):
        uppercase_attr = {}
        for name, val in dct.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val
        return super(UpperAttrMetaclass, cls).__new__(cls, clsname,
bases, uppercase_attr)
```

Что использовать – классы метаклассов или функции?

Metaclass02c.py

- Можно использовать ООП
  - Метакласс может наследоваться от метакласса, переопределять родительские методы
- Можно лучше структурировать свой код
  - Для решения сложных задач – возможность создавать несколько методов и группировать их в одном классе очень полезна, чтобы сделать код более удобным для чтения
- Можно использовать __new__, __init__ и __call__

Примеры применения метаклассов

Metaclass03use.py

- Создать метакласс MultiBases, который будет проверять, наследуются ли создаваемые классы от нескольких базовых классов – если так, это вызовет ошибку

```
class MultiBases(type):
    def __new__(cls, clsname, bases, clsdict):
        if len(bases)>1:
            raise TypeError("Inherited multiple base classes!!!")

        # в противном случае вызвать метод __new__ базового класса, т.е.
        # вызывается конструктор __init__
        return super().__new__(cls, clsname, bases, clsdict)
```

```
class Base(metaclass=MultiBases):
    pass
```

```
class B(Base):
    pass
```

```
class A(Base):
    pass
```

```
class C(A, B):
    pass
```

Примеры применения метаклассов

Metaclass04use.py

- Требуется, чтобы в процессе отладки, когда метод класса выполнялся, он выводил полное имя перед выполнением своего тела
  - классы будут создаваться, а затем сразу же оборачиваться декоратором метода отладки

```
class debugMeta(type):
    '''meta class передает созданный класс для получения требуемой
       функциональности при отладке (debugmethod)'''

    def __new__(cls, clsname, bases, clsdict):
        obj = super().__new__(cls, clsname, bases, clsdict)
        obj = debugmethods(obj)
        return obj
```

```
class Base(metaclass=debugMeta):
    pass
```

```
class Calc(Base):
    def add(self, x, y):
        return x+y
```

Примеры применения метаклассов

- Создание финального класса (final, sealed) — этот класс не позволяет создавать свои подклассы

Metaclass05use.py

- Использовать метакласс, чтобы следить за временем исполнения кода

Metaclass06use.py

- Другие примеры применения