Оглавление

Неделя 4

Углубленный Python

4.1. Особые методы классов

4.1.1. Магические методы

Магический метод — это метод, определённый внутри класса, который начинается и заканчивается двумя подчёркиваниями. Например, магическим методом является метод `__init__`, который отвечает за инициализацию созданного объекта. Давайте определим класс `User`, который будет переопределять магический метод `__init__`. В нём будем записывать полученые имя и e-mail в атрибуты класса. Также определим метод, который возвращает атрибуты класса в виде словаря. С этим вы уже должны быть знакомы:

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def get_email_data(self):
        return {
            'name': self.name,
            'email': self.email
        }

jane = User('Jane Doe', 'janedoe@example.com')

print(jane.get_email_data())
```

```
{'name': 'Jane Doe', 'email': 'janedoe@example.com'}
```

Ещё одним магическим методом является метод `__new__`, в котором прописано, что происходит в момент создания объекта класса. Метод `__new__` возвращает только что созданный объект класса. Например, создадим класс `Singleton`, который гарантирует то, что не может быть создано больше одного объекта данного класса. Например, мы можем попытаться создать два объекта `a` и `b`, которые в итоге окажутся одним и тем же объектом:

```
class Singleton:
    instance = None

    def __new__(cls):
        if cls.instance is None:
            cls.instance = super().__new__(cls)

        return cls.instance


a = Singleton()
b = Singleton()

a is b
```

```
True
```

Существует также метод `__del__`, который определяет поведение при удалении объекта. Однако, он работает не всегда очевидно. Он вызывается не когда мы удаляем объект оператором `del`, а когда количество ссылок на наш объект стало равным нулю и вызывался garbage collector. Это не всегда происходит тогда, когда мы думаем, что это должно произойти, поэтому переопределять метод `__del__` нежелательно.

Одним из магических методов является метод `__str__`, который определяет поведение, при вызове функции `print` от класса. Метод `__str__` должен определить человекочитаемое описание нашего класса, которое пользователь может потом вывести в интерфейсе. В следующем примере мы используем ранее написанный класс `User`, но теперь, если мы будем принтить наш объект, у нас будет выводиться понятное и читаемое название:

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def __str__(self):
        return '{} <{}>'.format(self.name, self.email)


jane = User('Jane Doe', 'janedoe@example.com')

print(jane)
```

```
Jane Doe <janedoe@example.com>
```

Ещё двумя полезными методами магическими являются методы `__hash__` и `__eq__`, которые определяют то, что происходит при вызове функции `hash` и как сравниваются

объекты соответственно. Магический метод `__hash__` переопределяет функцию хеширования, которая используется, например, когда мы получаем ключи в словаре. В следующем примере мы указываем в классе `User`, что при вызове функции `hash` в качестве хеша всегда берётся e-mail пользователя, и также при сравнении пользователей сравниваются их e-mail-ы. Таким образом, если мы создадим двух юзеров с разными именами, но одинаковыми e-mail-ами, при вызове функции сравнения Python покажет, что это один и тот же объект, потому что вызывается переопределённый метод `__eq__`, который сравнивает только e-mail-ы:

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def __hash__(self):
        return hash(self.email)

    def __eq__(self, obj):
        return self.email == obj.email


jane = User('Jane Doe', 'jdoe@example.com')
joe = User('Joe Doe', 'jdoe@example.com')


print(jane == joe)
```

```
True
```

Точно так же функция `hash` возвращает теперь одно и то же значение, потому что сравниваются только e-mail-ы, которые в данном случае одинаковы:

```
print(hash(jane))
print(hash(joe))
```

```
7885430882792781082
7885430882792781082
```

Также, если мы попробуем создать словарь, где в качестве ключа будут использоваться уже созданные объекты класса `User`, то создастся словарь только с одним ключом, потому что оба объекта имеют одинаковое значение хеша:

```
user_email_map = {user: user.name for user in [jane, joe]}

print(user_email_map)
```

```
{<__main__.User object at 0x107415908>: 'Joe Doe'}
```

Очень важными магическими методами являются методы, определяющие доступ к атрибутам. Это методы `__getattr__` и `__getattribute__`. Важно понимать отличия между ними. Итак, метод `__getattr__` определяет поведение, когда наш атрибут, который мы пытаемся получить, не найден. Метод `__getattribute__` вызывается в любом случае, когда мы обращаемся к какому-либо атрибуту объекта. Например, мы можем возвращать всегда какую-то строчку и ничего не делать, как в следующем примере. Мы определили класс и переопределили метод `__getattribute__`, который всегда возвращает одну и ту же строку. Таким образом, к какому бы атрибуту мы ни обратились, у нас всегда выведется эта строка:

```
class Researcher:
    def __getattr__(self, name):
        return 'Nothing found :('

    def __getattribute__(self, name):
        return 'nope'


obj = Researcher()

print(obj.attr)
print(obj.method)
print(obj.DFG2H3J00KLL)
```

```
nope
nope
nope
```

`__getattr__` вызывается в том случае, если атрибут не найден. В следующем примере внутри `__getattribute__`, который вызывается всегда, мы логируем, что пытаемся найти соответствующий атрибут и продолжаем выполнение, используя класс `object`, затем, если объект не найден, вызывается метод `__getattr__`:

```
class Researcher:
    def __getattr__(self, name):
        return 'Nothing found :()\n'

    def __getattribute__(self, name):
        print('Looking for {}'.format(name))
        return object.__getattribute__(self, name)


obj = Researcher()

print(obj.attr)
print(obj.method)
print(obj.DFG2H3J00KLL)
```

```
Looking for attr
Nothing found :()

Looking for method
Nothing found :()

Looking for DFG2H3J00KLL
Nothing found :()
```

Магический метод `__setattr__`, как вы могли догадаться, определяет поведение при присваивании значения к атрибуту. Например, вместо того, чтобы присвоить значение, мы можем опять же вернуть какую-то строчку и ничего не делать. В данном случае, если мы попытаемся присвоить значение атрибуту, у нас ничего не выйдет — атрибут не создастся:

```
class Ignorant:
    def __setattr__(self, name, value):
        print('Not gonna set {}!'.format(name))


obj  = Ignorant()
obj.math = True
```

```
Not gonna set math!
```

```
print(obj.math)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-677c3efbe80d> in <module>()
----> 1 print(obj.math)

AttributeError: 'Ignorant' object has no attribute 'math'
```

Наконец, метод `__delattr__` управляет поведением при удалении атрибута объекта. Например, его имеет смысл использовать, если мы хотим каскадно удалить объекты, связанные с нашим классом. В данном случае мы просто продолжаем удаление с помощью класса `object` и логируем то, что у нас происходит удаление:

```
class Polite:
    def __delattr__(self, name):
        value = getattr(self, name)
        print(f'Goodbye {name}, you were {value}!')

        object.__delattr__(self, name)


obj = Polite()

obj.attr = 10
del obj.attr
```

```
Goodbye attr, you were 10!
```

Ещё одним магическим методом является метод `__call__`, который определяет поведение программы при вызове класса. Например, с помощью метода `__call__` мы можем определить logger, который будем потом использовать в качестве декоратора (да, декоратором может быть не только функция, но и класс!). В примере ниже при инициализации класса `Logger` объект этого класса запоминает `filename`, который ему передан. Каждый раз, когда мы будем вызывать наш класс, он будет возвращать новую функцию в соответствии с протоколом декораторов и записывать в лог-файл строчку о вызове функции. В данном случае мы определяем пустую функцию, и декоратор записывает все её вызовы:

```
class Logger:
    def __init__(self, filename):
        self.filename = filename

    def __call__(self, func):
        def wrapped(*args, **kwargs):
            with open(self.filename, 'a') as f:
                f.write('Oh Danny boy...')

            return func(*args, *kwargs)
        return wrapped

logger = Logger('log.txt')

@logger
def completely_useless_function():
    pass
```

```
completely_useless_function()

with open('log.txt') as f:
    print(f.read())
```

```
Oh Danny boy...
```

Классическим примером на перегрузку операторов в других языках программирования является перегрузка оператора сложения. В Python-е за операцию сложения отвечает оператор `__add__` (в свою очередь, вычитание можно перегрузить с помощью метода `__sub__`). В качестве примера определим класс `NoisyInt`, который будет работать почти как integer, но добавлять шум при сложении:

```
import random


class NoisyInt:
    def __init__(self, value):
        self.value = value

    def __add__(self, obj):
        noise = random.uniform(-1, 1)
        return self.value + obj.value + noise


a = NoisyInt(10)
b = NoisyInt(20)
```

```
for _ in range(3):
    print(a + b)
```

```
30.605646527205856
30.170967742734117
29.071231797981817
```

В качестве упражнения вам предлагается написать свой контейнер с помощью методов `__getitem__` (определяет поведение объекта при доступе по индексу или ключу — `obj[key]`), `__setitem__` (определяет поведение объекта при присваивании по индексу или ключу — `obj[key] = value`).

Вот одно из возможных решений данного упражнения. Мы реализовали свой собственный класс `PascalList`, который имитирует поведение списков в Паскале. Как вы знаете, в Python-е списки нумеруются с нуля, а в Паскале — с единицы. Мы можем переопределить методы `__getitem__` и `__setitem__` так, чтобы они работали как в Паскале:

```
class PascalList:
    def __init__(self, original_list=None):
        self.container = original_list or []

    def __getitem__(self, index):
        return self.container[index - 1]

    def __setitem__(self, index, value):
        self.container[index - 1] = value

    def __str__(self):
        return self.container.__str__()


numbers = PascalList([1, 2, 3, 4, 5])

print(numbers[1])
```

```
1
```

```
numbers[5] = 25

print(numbers)
```

```
[1, 2, 3, 4, 25]
```

4.1.2. Итераторы

С итераторами вы уже работали раньше, когда, например, использовали функцию `range` для цикла `for`. Цикл `for` позволяет пробегать по итератору и, например, выводить подряд числа, как в случае с функцией range:

```
for number in range(5):
    print(number & 1)
```

```
0
1
0
1
0
```

Также простейшим итератором является строка или коллекция:

```
for letter in 'python':
    print(ord(letter))
```

```
112
121
```

```
116
104
111
110
```

Итератор — это объект, по которому вы можете "пробегаться" или итерироваться. Можно создать свой простейший итератор при помощи встроенной функции `iter()` и, например, передать ей список. Внутри протокол итерации работает очень просто. Для получения следующего элемента каждый раз вызывается функция `next()`, которая возвращает следующий элемент. В данном случае это 1, 2 и 3. Когда элементы исчерпаны, то есть итератор закончился, выбрасывается исключение `StopIteration`, которое говорит о том, что, например, нужно выйти из цикла `for`:

```
iterator = iter([1, 2, 3])

print(next(iterator))
```

```
1
```

```
print(next(iterator))
```

```
2
```

```
print(next(iterator))
```

```
3
```

```
print(next(iterator))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-7-369d18de2e56> in <module>()
----> 1 print(next(iterator))

StopIteration:
```

В Python-е вы, конечно, можете реализовать свой собственный итератор, написав класс с соответствующими магическими методами. Эти магические методы — это методы `__iter__` и `__next__`. Метод `__iter__` должен возвращать сам итератор, а метод `__next__` определяет то, какие элементы возвращаются при каждой следующей итерации.

Давайте напишем свой класс `SquareIterator`, который будет аналогом функции `range`, но возвращающим не сами числа в определённом диапазоне, а квадраты чисел. В функции `__init__` сохраняются пределы итерирования, а в функции `__next__` указано, что происходит при вызове следующего элемента. Если элементы исчерпаны (`current` превысил `end`), выбрасываем исключение `StopIteration`, которое говорит протоколу итерации о том, что итерация должна закончиться. В любом другом случае мы возводим число в квадрат и инкрементируем счётчик:

```
class SquareIterator:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.end:
            raise StopIteration

        result = self.current ** 2
        self.current += 1
        return result


for num in SquareIterator(1, 4):
    print(num)
```

```
1
4
9
```

Python позволяет вам создавать собственные итераторы, и иногда это бывает полезно, когда вам нужно поддержать протокол итерации в своём классе. Что интересно, можно также определить свой собственный итератор, не определяя `__iter__` и `__next__`. Это можно сделать, написав у класса метод `__getitem__`, который определяет работу класса при обращении к его объектам с помощью квадратных скобок (как к контейнеру). Мы можем создать свой собственный контейнер `IndexIterable`, прописав метод `__getitem__`:

```
class IndexIterable:
    def __init__(self, obj):
        self.obj = obj

    def __getitem__(self, index):
        return self.obj[index]


for letter in IndexIterable('str'):
    print(letter)
```

```
s
t
r
```

Это делается довольно редко. Чаще всего для того чтобы определить свой итератор, используются именно методы `__iter__` и `__next__`.

4.1.3. Контекстные менеджеры

С контекстными менеджерами вы уже работали, когда открывали файлы. Вам известно, что если использовать контекстный менеджер `with` для открытия файла, вам не нужно заботиться о том, чтобы потом его закрыть — контекстный менеджер сделает это сам.
Контекстные менеджеры определяют поведение, которое происходит в начале и в конце блока исполнения (блока `with`). Часто после использования ресурса его необходимо закрыть (как, например, в случае с файлами, сокетами, соединениями). Чтобы об этом не заботиться, можно использовать контекстный менеджер. Также они используются при работе с транзакциями (когда обязательно нужно либо закончить транзакцию, либо ее откатить).

Чтобы определить свой контекстный менеджер, нужно написать свой класс с магическими методами. Эти магические методы — `__enter__` и `__exit__`, которые говорят о том, что происходит в начале и в конце исполнения кода внутри контекстного менеджера. Давайте попробуем написать аналог стандартного контекстного менеджера для открытия файлов и назовём его `open_file`. (Обратите внимание, что название класса пишется snake_case-ом, так как это контекстный менеджер.)

```
with open('access_log.log', 'a') as f:
    f.write('New Access')
```

Итак, наш контекстный менеджер используется точно так же, как и стандартный. При вызове `open_file` создается файловый объект (вызывается метод `__init__`), который записывается в переменную класса `f`. Переменная `f` возвращается из метода `__enter__` (метод `__enter__` возвращает то, что требуется потом записать с помощью оператора `as` — мы можем ничего не возвращать из `__enter__`, но тогда не будет смысла использовать `as`). Соответственно, в методе `__exit__` определяется поведение, которое происходит при выходе из блока контекстного менеджера:

```
class open_file:
    def __init__(self, filename, mode):
        self.f = open(filename, mode)

    def __enter__(self):
        return self.f

    def __exit__(self, *args):
        self.f.close()
```

```
with open_file('test.log', 'w') as f:
    f.write('Inside `open_file` context manager')

with open_file('test.log', 'r') as f:
    print(f.readlines())
```

```
['Inside `open_file` context manager']
```

Итак, мы открыли файл и записали в него строчку. Если попробовать прочитать этот файл, окажется, что строчка действительно там, а файл открылся и закрылся сам.

Контекстные менеджеры позволяют управлять исключениями, которые произошли внутри блока. Например, мы можем определить контекстный менеджер `suppress _exception`, который будет работать с exception-ами, которые произошли внутри. Обратите внимание, что в данном случае мы не используем оператор `as`, поэтому нам не важно, что возвращается из `__enter__`. Пусть наш контекстный менеджер будет принимать тип exception-а и затем проверять, произошло ли исключение данного типа. Если да — делаем вид, что ничего не произошло. (Нужно обязательно вернуть true из `__exit__` при исключении, чтобы воспроизведение кода продолжилось и exception не был выброшен.)

```
class suppress_exception:
    def __init__(self, exc_type):
        self.exc_type = exc_type

    def __enter__(self):
        return

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type == self.exc_type:
            print('Nothing happend.')
            return True

with suppress_exception(ZeroDivisionError):
    really_big_number = 1 / 0
```

```
Nothing happend.
```

Что интересно, такой контекстный менеджер уже есть в стандартной библиотеке в `contextlib`:

```
import contextlib


with contextlib.suppress(ValueError):
    raise ValueError
```

В качестве примера реализуем контекстный менеджер, который считает время, за которое выполняется код внутри него. Для этого нужно завести переменную, которая фиксирует время запуска контекстного менеджера. Происходит это в методе `__init__`, когда создается объект класса. В `__exit__` вернём разность текущего времени и сохранённого в методе `__init__`. Также, чтобы иметь возможность выводить внутри контекстного менеджера текущее время выполнения, пропишем `return self` в `__enter__` и определим метод класса `current_time`:

```
import time


class timer():
    def __init__(self):
        self.start = time.time()

    def current_time(self):
        return time.time() - self.start

    def __enter__(self):
        return self

    def __exit__(self, *args):
        print('Elapsed: {}'.format(self.current_time()))


with timer() as t:
    time.sleep(1)
    print('Current: {}'.format(t.current_time()))
    time.sleep(1)
```

4.2. Механизм работы классов

4.2.1. Дескрипторы

С помощью дескрипторов в Python реализована практически вся магия при работе с объектами, классами и методами. Чтобы определить свой собственный дескриптор, нужно определить методы класса `__get__`, `__set__` или `__delete__`. После этого мы можем создать какой-то новый класс и в атрибут этого класса записать объект типа дескриптор. Теперь у данного объекта будет переопределено поведение при доступе к атрибуту (метод `__get__`), при присваивании значений (метод `__set__`) или при удалении (метод `__delete__`). Мы создадим объект класса `Class` и посмотрим, что будет происходить при обращении к атрибуту:

```
class Descriptor:
    def __get__(self, obj, obj_type):
        print('get')

    def __set__(self, obj, value):
        print('set')

    def __delete__(self, obj):
        print('delete')


class Class:
    attr = Descriptor()


instance = Class()
```

```
instance.attr
```

```
get
```

```
instance.attr = 10
```

```
set
```

```
del instance.attr
```

```
delete
```

Дескрипторы являются мощным механизмом, который позволяет вам незаметно от пользователя переопределять поведение атрибутов в ваших классах. Например, мы можем определить дескриптор `Value`, который будет переопределять поведение при присваивании ему значения. Определим класс с атрибутом, который будет являться дескриптором, и будем наблюдать модифицированное поведение (умножение на 10) при присваивании значения:

```
class Value:
    def __init__(self):
        self.value = None

    @staticmethod
    def _prepare_value(value):
        return value * 10

    def __get__(self, obj, obj_type):
        return self.value

    def __set__(self, obj, value):
        self.value = self._prepare_value(value)
```

```
class Class:
    attr = Value()


instance = Class()
instance.attr = 10

print(instance.attr)
```

```
100
```

Для примера реализуем дескриптор, который записывает в файл все присваиваемые ему значения. Таким образом, если мы создадим класс с какой-то важной информацией (например, класс `Account`), где важная информация -- это `amount`, денежное значение, которое всегда нужно сохранять:

```
class ImportantValue:
    def __init__(self, amount):
        self.amount = amount

    def __get__(self, obj, obj_type):
        return self.amount

    def __set__(self, obj, value):
        with open('log.txt', 'w') as f:
            f.write(str(value))

        self.amount = value


class Account:
    amount = ImportantValue(100)

bobs_account = Account()
bobs_account.amount = 200

with open('log.txt', 'r') as f:
    print(f.read())
```

```
200
```

Несмотря на то, что вы пользовались функциями и методами уже довольно давно, вы могли не знать, что на самом деле функции и методы реализованы с помощью дескрипторов. Чтобы понять, что это действительно так, можно попробовать обратиться к одному и тому же методу с помощью объекта класса и самого класса. Оказывается, когда мы обращаемся к методу с помощью `obj.method`, возвращается bound method — метод, привязанный к определённому объекту. А если мы обращаемся к методу от `Class`, получаем unbound method — это просто функция. Как видите, один и тот же метод возвращает

разные объекты в зависимости от того, как к нему обращаются. Это и есть поведение дескриптора:

```
class Class:
    def method(self):
        pass


obj = Class()

print(obj.method)
print(Class.method)
```

```
<bound method Class.method of <__main__.Class object at 0x10ee77278>>
<function Class.method at 0x10ee3bea0>
```

Вам уже знаком декоратор `@property`, который позволяет использовать функцию как атрибут класса. В данном случае мы можем определить `full_name`, который хоть и является функцией, которая возвращает строчку, используется потом так же, как и обычный атрибут, то есть без вызова скобок. При вызове `full_name` от объекта у нас вызывается функция `full_name`. Однако если мы пытаемся обратиться к `full_name` от класса, вернётся объект типа `property`. На самом деле, `property` реализовано с помощью дескрипторов, чем и объясняется разное поведение в зависимости от того, как вызывается этот объект:

```
class User:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'


amy = User('Amy', 'Jones')

print(amy.full_name)
print(User.full_name)
```

```
Amy Jones
<property object at 0x10ee7b598>
```

Напишем свой собственный класс, который будет эмулировать поведение стандартного `property`. Для этого нужно сохранить функцию, которую `property` получает. Когда объект вызывается от класса, мы просто возвращаем `self`, а если атрибут вызван от объекта, вызываем сохранённую функцию:

```
class Property:
    def __init__(self, getter):
        self.getter = getter

    def __get__(self, obj, obj_type=None):
        if obj is None:
            return self

        return self.getter(obj)
```

Протестируем работу только что созданного декоратора вместе со стандартным `@property`:

```
class Class:
    @property
    def original(self):
        return 'original'

    @Property
    def custom_sugar(self):
        return 'custom sugar'

    def custom_pure(self):
        return 'custom pure'

    custom_pure = Property(custom_pure)
obj = Class()

print(obj.original)
print(obj.custom_sugar)
print(obj.custom_pure)
```

```
original
custom sugar
custom pure
```

Точно так же реализованы `@staticmethod` и `@classmethod`. Давайте опять напишем свою реализацию этих декораторов. `StaticMethod` будет просто сохранять функцию и возвращать её при вызове:

```
class StaticMethod:
    def __init__(self, func):
        self.func = func

    def __get__(self, obj, obj_type=None):
        return self.func
```

В то же время, `ClassMethod` возвращает функцию, которая первым аргументом принимает `obj_type`, то есть класс:

```
class ClassMethod:
    def __init__(self, func):
        self.func = func

    def __get__(self, obj, obj_type=None):
        if obj_type is None:
            obj_type = type(obj)

        def new_func(*args, **kwargs):
            return self.func(obj_type, *args, **kwargs)

        return new_func
```

Последний пример дескрипторов в стандартной библиотеке Python — конструкция `__slots__`, которая позволяет определить класс с жестко заданным набором атрибутов. Как вы помните, у каждого класса есть словарь, в котором хранятся все его атрибуты. Очень часто это бывает излишне. У вас может быть огромное количество объектов, и вы не хотите создавать для каждого объекта словарь. В таком случае конструкция `__slots__` позволяет жестко задать количество элементов, которые ваш класс может содержать. В следующем примере мы постулируем, что в нашем классе должен быть только атрибут `anakin`. Если мы попытаемся добавить в наш класс еще один атрибут, ничего не получится:

```
class Class:
    __slots__ = ['anakin']

    def __init__(self):
        self.anakin = 'the chosen one'


obj = Class()

obj.luke = 'the chosen too'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-66c0c798df1f> in <module>()
      8 obj = Class()
      9
---> 10 obj.luke = 'the chosen too'

AttributeError: 'Class' object has no attribute 'luke'
```

Конструкция `__slots__` реализуется с помощью определения дескрипторов для каждого из атрибутов.

4.2.2. Метаклассы

Как вы уже знаете, всё в Python-е является объектом, и классы не исключение, а значит, эти классы кто-то создаёт. Давайте определим класс с названием `Class` и его объект. Тип нашего объекта является `Class`, потому что `Class` создал наш объект:

```
class Class:
    ...

obj = Class()

type(obj)
```

```
__main__.Class
```

Однако, у класса тоже есть тип — `type`, потому что `type`, создал наш класс. В данном случае `type`, является метаклассом, т.е. он создаёт другие классы:

```
type(Class)
```

```
type
```

Типом самого `type`, кстати, является он сам. Это рекурсивное замыкание, которое реализовано внутри Python с помощью C:

```
type(type)
```

```
type
```

Очень важно понимать разницу между созданием и наследованием. В данном случае класс не является subclass-ом `type`. `Type` его создаёт, но класс не наследуется от него, а наследуется от класса `object`:

```
issubclass(Class, type)
```

```
False
```

```
issubclass(Class, object)
```

```
True
```

Чтобы понять, как задаются классы, можно написать простую функцию, которая возвращает класс. В следующем примере мы определяем функцию, которая возвращает класс `dummy_factory`. Классы можно создавать на лету, и в данном случае мы создаём два разных объекта и возвращаем их:

```
def dummy_factory():
    class Class:
        pass

    return Class


Dummy = dummy_factory()

print(Dummy() is Dummy())
```

```
False
```

Однако на самом деле, Python работает не так. Для создания классов используется метакласс `type`, и вы можете на лету создать класс, вызвав `type` и передав ему название класса. Для примера создадим класс `NewClass` без родителей и атрибутов. Это настоящий класс, мы создали его на лету без использования литерала `class`:

```
NewClass = type('NewClass', (), {})

print(NewClass)
print(NewClass())
```

```
<class '__main__.NewClass'>
<__main__.NewClass object at 0x110cd7438>
```

Чаще всего классы создаются с помощью метаклассов. Давайте определим свой собственный метакласс `Meta`, который будет управлять поведением при создании класса. Для того чтобы он бы метаклассом, он должен наследоваться от другого метакласса (`type`). Метод метакласса `__new__` принимает название класса, его родителей и атрибуты. Мы можем определить новый класс `A` и указать, что его метаклассом является `Meta`. Именно этот метакласс и будет управлять поведением при создании нового класса. Таким образом, мы можем переопределить поведение при создании класса (например, добавить ему атрибут или сделать что-нибудь другое):

```
class Meta(type):
    def __new__(cls, name, parents, attrs):
        print('Creating {}'.format(name))

        if 'class_id' not in attrs:
            attrs['class_id'] = name.lower()

        return super().__new__(cls, name, parents, attrs)


class A(metaclass=Meta):
    pass
```

```
Creating A
```

```
print('A.class_id: "{}"'.format(A.class_id))
```

```
A.class_id: "a"
```

Например, мы можем определить метакласс, который переопределяет функцию `__init__`, и тогда каждый класс, созданный этим метаклассом, будет запоминать все созданные подклассы. Новый `__init__` записывает свой собственный атрибут, в котором будет храниться словарь созданных классов. В следующем примере у нас вначале создаётся класс `Base`, метаклассом которого является `Meta`, и у него создаётся атрибут класса `registry`, в который мы будем записывать все его подклассы. Каждый раз, когда у нас создаётся какой-то класс, который наследуется от `Base`, мы записываем в `registry` соответствующее значение, то есть название созданного класса и ссылку на него:

```
class Meta(type):
    def __init__(cls, name, bases, attrs):
        print('Initializing — {}'.format(name))

        if not hasattr(cls, 'registry'):
            cls.registry = {}
        else:
            cls.registry[name.lower()] = cls

        super().__init__(name, bases, attrs)


class Base(metaclass=Meta): pass

class A(Base): pass

class B(Base): pass
```

```
Initializing — Base
Initializing — A
Initializing — B
```

```
print(Base.registry)
print(Base.__subclasses__())
```

```
{'a': <class '__main__.A'>, 'b': <class '__main__.B'>}
[<class '__main__.A'>, <class '__main__.B'>]
```

Очень часто при работе с объектно-ориентированной парадигмой в Python-е возникают вопросы про *абстрактные методы*, потому что они являются центральным понятием, например, в языке программирования C++. В Python-е абстрактные методы реализованы в стандартной библиотеки `abc`. Здесь также работают метаклассы — они могут создать абстрактный класс с методом `@abstractmethod`. Декоратор `@abstractmethod` гарантирует, что у нас не получится создать класс-наследник, не определив этот метод — мы

обязаны его переопределить в классе, который наследуется от нашего класса. В следующем примере `Child` не переопределяет метод `send`, и поэтому вызывается ошибка:

```
from abc import ABCMeta, abstractmethod

class Sender(metaclass=ABCMeta):
    @abstractmethod
    def send(self):
        """Do something"""


class Child(Sender): pass

Child()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-5e10f1ccf1fd> in <module>()
      1 class Child(Sender): pass
      2
----> 3 Child()

TypeError: Can't instantiate abstract class Child with abstract
methods send
```

Переопределим метод `send`, и программа будет работать:

```
class Child(Sender):
    def send(self):
        print('Sending')


Child()
```

```
<__main__.Child at 0x110cfa860>
```

На самом деле, абстрактные методы используются в Python-е довольно редко, чаще всего вызывается исключение `NotImplementedError`, которое говорит о том, что этот метод нужно реализовать. Программист видит в определении класса, что в методе вызывается `raise NotImplementedError`, и понимает, что этот метод нужно переопределить в потомке:

```
class PythonWay:

    def send(self):
        raise NotImplementedError
```

4.3. Отладка и тестирование

4.3.1. Отладка

Скорее всего отладкой вы уже занимались, когда пытались выяснить, почему ваша программа не работает или работает некорректно. Если вы программируете в IDE, скорее всего, там есть инструментарий для отладки кода, и вы можете запускать вашу программу под отладчиком, ставить брейкпоинты, следить за переменными и т.д. Мы разберём классический механизм отладки с помощью Python Debugger-а. Например, мы написали программу, которая принимает на вход сайт, URL сайта и какую-то строчку, и ищет в коде сайта эту строчку и считает, сколько раз там встретилась эта строка. Затем запустили программу в Jupyter Notebook. Эта программа выдаёт неочевидную ошибку (проверьте, запустив самостоятельно), которую мы хотим отладить:

```
import re
import requests

def main(site_url, substring):
    site_code = get_site_code(site_url)
    matching_substrings = get_matching_substrings(site_code,
substring)
    print('"{}" found {} times in {}'.format(
        substring, len(matching_substrings), site_url
    ))

def get_site_code(site_url):
    if not site_url.startswith('http'):
        site_url = 'http://' + site_url

    return requests.get(site_url).text

def get_matching_substrings(source, substring):
    return re.findall(source, substring)


main('mail.ru', 'script')
```

Чтобы запустить отладчик в нашей программе, мы можем импортировать `pdb` и вызвать команду `pdb.set_trace` в том месте кода, где мы хотим остановить выполнение и начать отладку:

```
def main(site_url, substring):
    import pdb
    pdb.set_trace()

    site_code = get_site_code(site_url)
    matching_substrings = get_matching_substrings(site_code,
substring)
    print('"{}" found {} times in {}'.format(
        substring, len(matching_substrings), site_url
    ))
```

Если мы теперь запустим программу, она не дойдёт до ошибки, а остановится именно в том месте, где мы определили наш отладчик. Можем посмотреть, где это находится, с помощью команды `ll (long list)`. Печатаем эту команду прямо в окошко Pdb и получаем:

```
> <ipython-input-4-0092ef746d04>(8)main()
-> site_code = get_site_code(site_url)
(Pdb) ll
  4      def main(site_url, substring):
  5          import pdb
  6          pdb.set_trace()
  7
  8  ->        site_code = get_site_code(site_url)
  9          matching_substrings = get_matching_substrings(site_code,
substring)
 10          print('"{}" found {} times in {}'.format(
 11              substring, len(matching_substrings), site_url
 12          ))
```

Все команды отладчика посмотреть с помощью функции `help`. Чтобы узнать, что делает конкретная команда, пишут знак вопроса и затем название команды. Например:

```
(Pdb) ? p
p expression
        Print the value of the expression.
```

Команда `p` — это сокращение от `print`, она выводит выражение (например, можем вывести переменную, которая находится в области нашей видимости).

Есть полезная команда `args`, которая говорит о том, какие аргументы переданы функции, в которой мы находимся. Сейчас мы находимся в функции `main`, аргументы переданы правильно и всё корректно:

```
(Pdb) args
site_url = 'mail.ru'
substring = 'script'
```

Итак, дальше есть несколько опций. Мы можем написать `continue` или `c` и продолжить исполнение до конца, пока не закончится программа, не упадёт исключение или не появится брейкпоинт. Также мы можем написать `step`, чтобы попасть внутрь функции, однако у нас ошибка не внутри функции, а дальше, поэтому мы напишем `next` и перейдём на следующую строку. Следующая строка — это вызов функции `get_matching_substrings`, именно он нас интересует. Пройдём внуть функции, набрав `step`. Дальше — вызов функции `findall`, в котором и была ошибка. Давайте посмотрим, что же у нас происходит:

```
(Pdb) next
> <ipython-input-4-0092ef746d04>(9)main()
-> matching_substrings = get_matching_substrings(site_code,
substring)
(Pdb) step
--Call--
> <ipython-input-4-0092ef746d04>(20)get_matching_substrings()
-> def get_matching_substrings(source, substring):
(Pdb) n
> <ipython-input-4-0092ef746d04>(22)get_matching_substrings()
-> return re.findall(source, substring)
```

Итак, мы в `findall` передаём `source` и `substring` и пытаемся найти `substring` в `source`. Давайте зайдём в `findall`, которая определена в стандартной библиотеке:

```
(Pdb) s
--Call--
> /usr/lib/python3.5/re.py(205)findall()
-> def findall(pattern, string, flags=0):
```

Внимательный читатель уже заметил, в чём была ошибка, посмотрев на определение `findall`. Функция `findall` принимает вначале паттерн, а потом строку, и ищет этот паттерн в строке, а мы передаём ровно наоборот. Именно это и является ошибкой. Давайте выйдем или продолжим исполнение. Чтобы выйти, нужно написать `q` или `quit`.

Если мы исправим нашу ошибку, программа заработает и выдаст результат:

```
"script" found 272 times in mail.ru
```

Часто вам необходимо отлаживать не в одном месте, и вы хотите исполнение до определённого момента. На помощь приходят брейкпоинты, которые очень удобно ставятся в

IDE. В pdb можно воспользоваться командой b, которая, если написать её без аргументов, выводит все брейкопоинты, которые мы уже определили. Если мы используем эту команду с номером строки (например, `b 10`), она поставит брейкопоинт на десятую строку. И когда исполнение программы дойдёт до этой строки, интерпретатор Python остановится. Мы можем написать `continue` и остановиться ровно на нашем брейкпоинте.

Итак, мы познакомились с Python Debugger-ом — простым и удобным инструментом для отладки программ на Python, которым можно пользоваться как в Jupyter Notebook, так и в консоли.

4.3.2. Тестирование

Настало время поговорить о тестировании, которого так бояться многие программисты. Если вы работаете над большим, быстро изменяющимся проектом с большим количеством разработчиков, вам нужно постоянно проверять, правильно ли работает ваша программа в различных условиях. Именно это и называется тестированием. Тестированию можно посвятить отдельную тему, курс и даже специализацию, потому что это огромная область. Мы с вами разберем наиболее популярный и распространенный вид тестирования — это unit-тестирование.

Unit-тесты призваны протестировать небольшую функцию, класс или модуль — посмотреть, корректно ли он работает. Чтобы определить свой unittest можно воспользоваться стандартной библиотекой модулей `unittest` и определить свой класс, который наследуется от `TestCase` из модуля `unittest`. Внутри класса вы можете определить функции, которые и будут являться тестами. Каждая функция, которая начинается с `test_` является тестом. В следующем примере мы хотим проверить, правильно ли у нас приводится типы, и например, корректно ли у нас работает функция `get` у пустого словаря. Делается это с помощью методов `TestCase`: `assertEqual`, `assertIsNone`, `assertRaises` и т.д. (подробнее можно прочесть в документации). Все они делают одно: проверяют, корректно ли работает выражение, правильно ли вызывается функция и так далее:

```
# test_python.py

import unittest


class TestPython(unittest.TestCase):
    def test_float_to_int_coercion(self):
        self.assertEqual(1, int(1.0))

    def test_get_empty_dict(self):
        self.assertIsNone({}.get('key'))

    def test_trueness(self):
        self.assertTrue(bool(10))
```

Для запуска тестов можно воспользоваться консолью. Ещё чаще тесты запускает автоматическая система сборки или тестирования или IDE. Давайте перейдем в консоль и запустим наши тесты:

```
python3 -m unittest test_python.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Наши тесты прошли --- об этом говорят три точки и надпись.

Давайте запустим тест, который не должен проходить, и посмотрим, что из этого выйдет:

```
# test_division.py

import unittest


class TestDivision(unittest.TestCase):
    def test_integer_division(self):
        self.assertIs(10 / 5, 2)
```

```
python3 -m unittest test_division.py
F
======================================================================
FAIL: test_integer_division (test_division.TestDivision)
----------------------------------------------------------------------
Traceback (most recent call last):
  File
"/Users/alexander/Dropbox/Teaching/coursera/tests/test_division.py",
line 6, in test_integer_division
    self.assertIs(10 / 5, 2)
AssertionError: 2.0 is not 2

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Да, наш тест упал. Об этом говорит буква `F` и дальнейшее описание. Можем посмотреть, какая функция упала и с каким `AssertionError`.

Давайте посмотрим на конкретный пример — напишем свой собственный класс и попробуем его протестировать. Пусть это будет `Asteroid` для работы с открытым API NASA

по астероидам и прочим телам, которые летают вокруг Земли:

```
import requests


class Asteroid:
    BASE_API_URL =
'https://api.nasa.gov/neo/rest/v1/neo/{}?api_key=DEMO_KEY'

    def __init__(self, spk_id):
        self.api_url = self.BASE_API_URL.format(spk_id)

    def get_data(self):
        return requests.get(self.api_url).json()

    @property
    def name(self):
        return self.get_data()['name']

    @property
    def diameter(self):
        return int(self.get_data()['estimated_' +
        'diameter']['meters']['estimated_diameter_max'])

    @property
    def closest_approach(self):
        closest = {
            'date': None,
            'distance': float('inf')
        }
        for approach in self.get_data()['close_approach_data']:
            distance = float(approach['miss_distance']['lunar'])
            if distance < closest['distance']:
                closest.update({
                    'date': approach['close_approach_date'],
                    'distance': distance
                })
        return closest
```

Вы можете заметить, что мы каждый раз вызываем функцию `get_data` и каждый раз идем в Интернет. Действительно, это можно оптимизировать ☺

Давайте протестируем наш класс и посмотрим, корректно ли работают функции `name` и `diameter`. Однако, есть некоторая тонкость. Каждый раз, когда мы будем запускать тесты, `TestCase` будет ходить в Интернет, потому что запускается функция `get_data`. Это не всегда будет работать, т.к. мы можем запускать наши тесты в окружении без Интернета (или Интернет медленный, или мы экономим трафик). То же самое можно сказать про

работу с сетями вообще или другими ресурсами, например, с диском (возможно, мы не хотим загружать диск). Что же делать в таком случае? Рассмотрим на примере несколько полезных механизмов.

Итак, давайте протестируем наш класс `Asteroid` на примере астероида Апофис (довольно большой астероид, сближающийся с Землёй):

```
apophis = Asteroid(2099942)

print(f'Name: {apophis.name}')
print(f'Diameter: {apophis.diameter}m')
```

```
Name: 99942 Apophis (2004 MN4)
Diameter: 682m
```

Напишем `TestCase` для нашего класса. Сначала определяем новую функцию `setUp`, призванную "засетапить" окружение, которое будет работать во время исполнения тестовой функции. Таким образом, если нам нужно работать с объектом класса `Asteroid`, функция `setUp` будет в начале исполнения каждой функции создавать этот объект, чтобы не дублировать код в начале тестовых функций. (Существует симметричный метод, который называется `tearDown`, который позволяет закрывать ресурсы и удалять объекты в конце каждой тестовой функции.)

Давайте напишем две тестовые функции, которые будут проверять работу `name` и `diameter`. Однако, что если мы тестируем наши функции в окружении без Интернета? На помощь приходит механизм mock-ов из модуля `unittest.mock`, который позволяет подменять одни функции другими. Таким образом, мы можем на самом деле не ходить в Интернет, а читать информацию из файла. В следующем примере мы подменим функцию `get_data` другой функцией, которая просто читает из файла. Делается это с помощью декоратора `@patch`. Мы можем проверять внутри тестовой функции определенные условия — в данном случае корректность имени астероида и значения его размера:

```
import json
import unittest
from unittest.mock import patch

from asteroid import Asteroid


class TestAsteroid(unittest.TestCase):
    def setUp(self):
        self.asteroid = Asteroid(2099942)

    def mocked_get_data(self):
        with open('apophis_fixture.txt') as f:
            return json.loads(f.read())

    @patch('asteroid.Asteroid.get_data', mocked_get_data)
    def test_name(self):
        self.assertEqual(
            self.asteroid.name, '99942 Apophis (2004 MN4)'
        )

    @patch('asteroid.Asteroid.get_data', mocked_get_data)
    def test_diameter(self):
        self.assertEqual(self.asteroid.diameter, 682)
```

Запустим наш тестовый класс:

```
python3 -m unittest test_asteroid.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.004s

OK
```

Скорее всего вы не будете запускать тесты вручную. Это будет запускать автоматическая система. Также существует возможность автоматического нахождения тестов, которые лежат в директории tests.

Мы написали тест для нашего класса и знаем, что у атрибуты `name` и `diameter` работают корректно. Осталось только выяснить, когда прилетит астероид и насколько близко:

```
print(f'Date: {apophis.closest_approach["date"]}')
print(f'Distance: {apophis.closest_approach["distance"]:.2} LD')
```

```
Date: 2029-04-13
Distance: 0.099 LD
```

(Как видите, довольно близко, но специалисты лаборатории реактивного движения NASA заявили, что возможность столкновения с Землёй в 2029 году исключена.)