Moscow Coding School

Python как первый язык

Преподаватель: Захарчук Сергей Сергеевич

Сегодня

- Вспомним про ООП
- Декораторы
- Итераторы и генераторы
- Comprehensions
- Контекстные менеджеры
- Слоты
- Функтор
- Дескрипторы
- Sequence
- Мета-классы

Инкапсуляция

Инкапсуляция — ограничение доступа к объектам и компонентам. «Разделение на публичный и приватный доступ»

Вспомним, что 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.

Пример

```
class A:
        def _private(self):
                print("Это приватный метод!")
>>> a = A()
>>> a._private()
"Это приватный метод!"
```

Пример

```
>>> 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’
```

НО! атрибут остаётся доступным под именем _ИмяКласса__ИмяАтрибута:

```
>>> b._B__private()
```

Наследование

Наследование подразумевает то, что дочерний класс содержит все атрибуты родительского класса, при этом некоторые из них могут быть переопределены или добавлены в дочернем.

Пример

```
>>> class Diction(dict):
        ... def get(self, key, default = 0):
            ... return dict.get(self, key, default)

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

```
>>> 1 + 1
2
>>> "1" + "1"
'11'
```

Декораторы

Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения какого либо дополнительного поведения к объекту.

Декораторы

```
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()
Выведет: --ветчина–

sandwich = bread(ingredients(sandwich))
sandwich()
Выведет:
# </------\> #
#помидоры#
# --ветчина-- #
 ~салат~
 # <\_______/>
```

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

sandwich()
Выведет:
# </------\> #
#помидоры#
# --ветчина-- #
~салат~
# <\_______/>
```

Генераторы

С помощью генераторов списков можно легко отобразить один список в другой, применив некоторую функцию к каждому элементу.

```
>>> a_list = [1, 9, 8, 4]
>>> [elem * 2 for elem in a_list]
[2, 18, 16, 8]
```

Генераторы списков

```
[elem * 2 for elem in a_list]
```

1. Читаем генератор справа налево. a_list — отображаемый список. Python последовательно перебирает элементы списка a_list, временно присваивая значение каждого элемента переменной elem. Затем применяет функцию elem * 2 и добавляет результат в возвращаемый список.
2. Генератор создаёт новый список, не изменяя исходный.
3. Можно присвоить результат работы генератора списка отображаемой переменной. Python создаст новый список в памяти и, когда результат работы генератора будет получен, присвоит его исходной переменной.

Генераторы списков

```
>>> import os, glob
>>> glob.glob('*.jpg')
[’img1.jpg', 'img2.jpg', 'img3.jpg']
```

```
>>> [os.path.realpath(f) for f in glob.glob('*.jpg')]
['c:\\Users\\rtridz\\test\\4\\img1.jpg',
 'c:\\Users\\ rtridz \\ test \\4\\img2.jpg',
 'c:\\Users\\ rtridz \\ test \\4\\img3.jpg ']
```

Это выражение возвращает список всех .jpg-файлов в текущем рабочем каталоге.

Генераторы словарей

```
>>> import os, glob
>>> metadata = [(f, os.stat(f)) for f in glob.glob('*example*.py')]
>>> metadata[0]
('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0,
st_dev=0,
 st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
 st_mtime=1247520344, st_ctime=1247520344))
```

```
>>> metadata = [(f, os.stat(f)) for f in glob.glob('*test*.py')]
```

Это не генератор словаря, это генератор списка. Он находит все файлы с расширением .py, проверяет их имена, а затем создает кортеж из имени файла и метаданных файла (вызывая функцию os.stat()).

```
>>> metadata[0]
('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0,
st_dev=0,
 st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
 st_mtime=1247520344, st_ctime=1247520344))
```

Каждый элемент полученного списка является кортежем.

```
>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*test*.py')}
```

Это генератор словаря.

Отличается от генератора списка тем, что он заключён в фигурные скобки, а не в квадратные.

А также, вместо одного выражения для каждого элемента он содержит два, разделённые двоеточием. Выражение слева от двоеточия (f) является ключом словаря; выражение справа от двоеточия (os.stat(f)) — значением.

```
>>> type(metadata_dict)
<class 'dict'>
```

Генератор словаря возвращает словарь.

Ключи данного словаря — это просто имена файлов, полученные с помощью glob.glob('*test*.py').

```
>>> list(metadata_dict.keys())
['test1.py', 'test2.py', 'test3.py']
```

Также, как и в генераторах списков, вы можете включать
в генераторы словарей условие if

```
>>> dict =
{os.path.splitext(f)[0]:some_list.approximate_size(meta.st_size) \
...                 for f, meta in metadata_dict.items() if meta.st_size > 6000}
```

Перестановка местами ключей и значений словаря.

```
>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> {value:key for key, value in a_dict.items()}
{1: 'a', 2: 'b', 3: 'c'}
```

Конечно же, это сработает, только если значения элементов словаря неизменяемы, как, например, строки или кортежи.

```
>>> a_dict = {'a': [1, 2, 3], 'b': 4, 'c': 5}
>>> {value:key for key, value in a_dict.items()}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <dictcomp>
TypeError: unhashable type: 'list'
```

Генераторы множеств

Нельзя оставить за бортом и множества, они тоже могут создаваться с помощью генераторов. Единственное отличие — вместо пар ключ:значение, они строятся на основе одних значений.

```
>>> a_set = set(range(10))
>>> a_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

В качестве входных данных генераторы множеств могут получать другие множества. Этот генератор рассчитывает квадраты множества чисел в диапазоне от 0 до 9.

Генераторы множеств

```
>>> {x ** 2 for x in a_set}
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
```

```
>>> {x for x in a_set if x % 2 == 0}
{0, 8, 2, 4, 6}
```

Подобно генераторам списков и словарей, генераторы множеств могут содержать условие if для проверки каждого элемента перед включением его в результирующее множество.

```
>>> {2**x for x in range(10)}
{1, 2, 4, 8, 16, 32, 64, 128, 256, 512}
```

Итераторы

В Python есть два понятия, которые звучат практически одинаково, но обозначают разные вещи, — iterator и iterable.

Итератор — это объект-абстракция, который позволяет брать из источника, будь это stdin или, скажем, какой-то большой контейнер, элемент за элементом, при этом итератор знает только о том объекте, на котором он в текущий момент остановился, а второе — контейнер, который может служить источником данных для итератора.

Итераторы

Чтобы экземпляр класса можно было засунуть куда-нибудь в for, класс должен реализовывать два метода — **iter**() и next() (в третьем Python **next**()).

Действительно, по list’у можно итерироваться, но сам по себе list никак не следит, где там мы остановились в проходе по нему. А следит объект по имени listiterator, который возвращается методом iter() и используется, скажем, циклом for или вызовом map(). Когда объекты в перебираемой коллекции кончаются, возбуждается исключение StopIteration.

Итераторы

```
class Fibonacci:
    '''iterator that return numbers in the Fibonacci sequence'''

    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

Основная фишка генератора в том, что он, подобно итератору, запоминает последний момент, когда к нему обращались, но при этом оперирует не абстрактными элементами, а вполне конкретными блоками кода. То есть если итератор по умолчанию будет перебирать элементы в контейнере, пока они не кончатся, то генератор будет гонять код, пока не выполнится какое-нибудь конкретное условие возврата.

Чтобы получить объект-итератор, нужно создать объект, который будет иметь два метода со специальными именами:

- __iter__() - метод, который возвращает сам объект.
- next() - метод, который возвращает следующее значение итератора.

Пример итератора

Предположим, что нам нужно сделать объект, который берет данные из строк очень большого файла. Данные нужны порциями, которые записаны в строках текстового файла, одна порция, одна строка. Обрабатывать нужно в цикле "for...in”

```
class SimpleIterator(object):
        def __init__(self,fname):
                self.fd = open(fname,'r')
        def __iter__(self): 7
                return self
        def next(self):
                l = self.fd.readline()
                if l != '':
                        l = l.rstrip('\n')
                        num = int(l)
                        return num*2
                raise StopIteration
```

```
>>> simple_iter = SimpleIterator('test_data.txt')
>>> for i in simple_iter:
...             print (i)
```

Итераторы

Mylist является итерируемым объектом. Когда вы создаёте список, используя генераторное выражение, вы создаёте также итератор:

```
>>> mylist = [x*x for x in range(3)]
>>> for i in mylist :
... print(i)
0
1
4
```

Итераторы

Всё, к чему можно применить конструкцию «for… in...», является итерируемым объектом: списки, строки, файлы…

Это удобно, потому что можно считывать из них значения сколько потребуется — однако все значения хранятся в памяти, а это не всегда желательно, если у вас много значений.

Ещё раз про генераторы и yield

Генератор, это очень похоже на итератор, только это функция. При вызове этой функции в цикле, она при каждом новом цикле выдает следующее значение. Пишется эта функция с использованием оператора *yield*

Итераторы

Когда вы создаёте список, вы можете считывать его элементы один за другим — это называется итерацией:

```
>>> mylist = [1, 2, 3]
>>> for i in mylist :
 ... print(i)
1
2
3
```

Генераторы

Генераторы это тоже итерируемые объекты, но прочитать их можно лишь один раз. Это связано с тем, что они не хранят значения в памяти, а генерируют их на лету:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator :
 ... print(i)
0
1
4
```

Генераторы

Нельзя применить конструкцию for i in mygenerator второй раз, так как генератор может быть использован только единожды: он вычисляет 0, потом забывает про него и вычисляет 1, завершаяя вычислением 4 — одно за другим.

yield

yield работает как *return*, с одной разницей, что между вызовами функций, все состояния и данные будут при следующем вызове функции такими, какими они были на момент предыдущего исполнения *yield*.

yield создает итератор, просто создание итератора упрощается за счет того, что методы *__iter__()* и *next()* создаются автоматически. При выходе из функции не по оператору *yield* автоматически генерируется *StopIteration*

Отличие от return в том, что функция вернёт генератор.

yield

```
>>> def createGenerator() :
... mylist = range(3)
... for i in mylist :
        ... yield i*i
>>> mygenerator = createGenerator() # создаём генератор
>>> print(mygenerator) # mygenerator является объектом!
<generator object createGenerator at 0xb7555c34> >>>
for i in mygenerator: ... print(i)
0
1
4
```

yield

Полезно в том случае, когда функция вернёт большой набор значений, который надо будет прочитать только один раз.

Функция будет исполняться от начала до того момента, когда она наткнётся на yield — тогда она вернёт первое значение из цикла. На каждый следующий вызов будет происходить ещё одна итерация написанного вами цикла, возвращаться будет следующее значение — и так пока значения не кончатся.

Генератор считается пустым, как только при исполнении кода функции не встречается yield. Это может случиться из-за конца цикла, или же если не выполняется какое-то из условий «if/else».

Itertools

Модуль itertools содержит специальные функции для работы с итерируемыми объектами

- продублировать генератор
- Соединить два генератора последовательно
- Сгруппировать значения вложенных списков в одну строчку

```
>>> a= [1, 2, 3, 4]
>>> b= itertools.permutations(a)
>>> print(list(itertools.permutations(a)))
```

Итерация

Итерация это процесс, включающий итерируемые объекты (реализующие метод __iter__()) и итераторы (реализующие __next__()). Итерируемые объекты это любые объекты, из которых можно получить итератор. Итераторы это объекты, позволяющие итерировать по итерируемым объектам.

List Comprehensions

```
>>> res = []
>>> for x in range(1, 25, 2):
        ... res.append(x) ...
>>> print (res)

>>> res = [x for x in range(1, 25, 2)]
>>> print (res)
```

Слоты

Слоты — это список атрибутов, задаваемый в заголовке класса с помощью __slots__. В инстансе необходимо назначить атрибут, прежде чем пользоваться им:

```
class Limiter(object):
        __slots__ = ['age', 'name', 'job']


>>> x=Limiter()
>>> x.age = 20
```

Функтор

Функтор — это класс, у которого есть метод __call__ — при этом объект можно вызвать как функцию.

1) Пусть у нас имеется класс Person,

2) Имеется коллекция объектов этого класса- people,

нужно отсортировать эту коллекцию по фамилиям.

Для этого можно использовать функтор Sortkey:

Пример

```
class Person:
        def __init__(self, forename, surname, email):
                self.forename = forename
                self.surname = surname
                self.email = email
```

Пример

```
class SortKey:
        def __init__(self, *attribute_names):
                self.attribute_names = attribute_names
        def __call__(self, instance):
                values = []
                for attribute_name in self.attribute_names:
                        values.append(getattr(instance, attribute_name))
                return values
```

getattr(object, name ,[default]) - извлекает атрибут объекта или default.

Пример

```
>>> people=[]
>>> p=Person('Petrov','','')
>>> people.append(p)
>>> p=Person(u'Ivanov','','')
>>> people.append(p)
>>> p=Person('Sidorov','','')
>>> people.append(p)
>>> for p in people:
        ... print (p.forename)
```

```
>>> people.sort(key=SortKey("forename"))
>>> for p in people:
        ... print (p.forename)
```

Ещё easy пример

```
>>> class Name(object):
        ... def __call__(self, first, second):
            ... return first + second ...
 >>> f = Name()
>>> f(1,2)
3
```

Дескриптор

Дескриптор — это класс, который хранит и контролирует атрибуты других классов. Любой класс, который содержит один из специальных методов — __get__ , __set__ , __delete__, является дескриптором.

Пример

```
class Point: __slots__ = ()
        x = ExternalStorage("x")
        y = ExternalStorage("y")
        def __init__(self, x=0, y=0):
                self.x = x
                self.y = y
>>> p1=Point(1,2)
>>> p2=Point(3,4)
```

Класс Point не имеет собственных атрибутов x, y, они хранятся в дескрипторе ExternalStorage.

Пример

```
class ExternalStorage:
        __slots__ = ("attribute_name",)
        __storage = {}
        def __init__(self, attribute_name):
                self.attribute_name = attribute_name
        def __set__(self, instance, value):
                self.__storage[id(instance), \
        self.attribute_name] = value
        def __get__(self, instance, owner=None):
                if instance is None:
                        return self
                return self.__storage[id(instance), self.attribute_name]
```

Sequence

Последовательность реализуется с помощью методов __getitem__, __setitem__.

Задание

- Написать класс Sequence, который возвращает по индексу элемент последовательности неопределенной длины
- Она представляющей собой арифметическую прогрессию вида:

  1 3 5 7 9 11...
- Нельзя применить стандартные методы __del__ , __len__

Решение

```
class Sequence:
        def __init__(self, start=0, step=1):
        self.start = start
        self.step = step
        self.changed = {}
        def __getitem__(self, key):
                return self.start + key*self.step
def __setitem__(self, key, value):
        self.changed[key] = value
```

```
>>> s = Sequence(1,2)
>>> s[0]
1
>>> s[1]
3
>>> s[100]
201
```

Метаклассы

Метаклассы - это классы, экземплярами которых являются другие классы.

Помните функцию type, которая позволяет определить тип объекта:

>>> print (type(1))

<type 'int'> >>>

print (type("1"))

<type 'str'>

Она также позволяет создавать классы. type принимает на вход описание класса и возвращает класс.

type

type работает следующим образом:

type(<имя класса>, <кортеж родительских классов>,
<словарь, содержащий атрибуты и их значения>)

```
>>> class EasyClass(object):
...     pass
```

Или что тоже самое

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

type

type принимает словарь, определяющий атрибуты класса:

```
>>> class Foo(object):
        ... bar = True
```

Тоже самое только через type

```
>>> Foo = type('Foo', (), {'bar':True})
```

И использовать как обычный класс

```
>>> print (Foo)
<class '__main__.Foo'>
>>> print (Foo.bar)
True
```

type

Если потребуется добавить методов классу, то можно определить функцию с нужной сигнатурой и присвойте её в качестве атрибута:

```
>>> def echo_bar(self):
        ... print self.bar
>>> EasyClass = type('EasyClass', (Foo,), {'echo_bar': echo_bar})
>>> hasattr(EasyClass, 'echo_bar')
True
>>> hasattr(Foo, 'echo_bar')
>>> my_foo = Foo()
>>> my_foo.echo_bar()
True
```

Метаклассы

Создали класс - создавали объектыА классы являются объектами. Метакласс это то, что создаёт эти самые объекты. Они являются классами классов.

SomeClass = MetaClass()

someObject = SomeClass()

т.е. SomeClass = type('SomeClass', (), {})

type - это метакласс для создания всех классов

Атрибут __metaclass__

При написании класса можно добавить атрибут __metaclass__:
class SuperClass(object):

```
    __metaclass__ = something
```

Тогда при создании будет использоваться указанный метакласс при создании класса SuperClass

Пример

```
>>>def upper_attr(future_class_name, future_class_parents,
future_class_attr):
        ... attrs = ((name, value) for name, value in
future_class_attr.items() if not name.startswith('__'))
        ... uppercase_attr = dict((name.upper(), value) for name, value in
attrs)
        ... return type(future_class_name, future_class_parents,
uppercase_attr)
```

Пример

```
class Foo(object):
        __metaclass__ = upper_attr
        bar = 'bip'
        print (hasattr(Foo, 'bar'))
        False
        print (hasattr(Foo, 'BAR'))
        True
        f = Foo()
        print (f.BAR)
        'bip'
```