Moscow Coding School

Python как первый язык

Преподаватель: Захарчук Сергей Сергеевич

Сегодня

- Классы
- Конструкторы
- Декораторы функций

Введение в ООП

Python проектировался как объектно- ориентированный язык программирования. Это означает, что он построен с учетом следующих принципов:

- Все данные в нем представляются объектами.
- Программу можно составить как набор взаимодействующих объектов, посылающих друг другу сообщения.
- Каждый объект имеет собственную часть памяти и может состоять из других объектов.
- Каждый объект имеет тип.
- Все объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия).

Введение в ООП

- **Объектно-ориентированное программирование** (**ООП**) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.
- **Класс** — тип, описывающий устройство объектов.
- **Объект** — это экземпляр класса.

Основные понятия

При процедурном программировании программа разбивается на части в соответствии с алгоритмом: каждая часть (подпрограмма, функция, процедура) является составной частью алгоритма.

При объектно-ориентированном программировании программа строится как совокупность взаимодействующих объектов.

Объект

С точки зрения объектно-ориентированного подхода:

Объект - это нечто, обладающее значением (состоянием), типом (поведением) и индивидуальностью. Когда программист выделяет объекты в предметной области, он обычно абстрагируется (отвлекается) от большинства их свойств, концентрируясь на существенных для задачи свойствах.

Над объектами можно производить операции (посылая им сообщения). В языке Python все данные представлены в виде объектов.

Объект

Некоторые встроенные объекты имеют в Python синтаксическую поддержку (для задания литералов). Таковы числа, строки, списки, кортежи и некоторые другие типы.

Взаимодействие объектов заключается в **вызове методов** одних объектов другими.

Объекты посылают друг другу сообщения. Сообщения - это запросы к объекту выполнить некоторые действия. (Сообщения, методы, операции, функции-члены являются синонимами).

Объект

Каждый объект хранит свое состояние (для этого у него есть **атрибуты**) и имеет определённый набор методов.

(Синонимы: атрибут, поле, слот, объект-член, переменная экземпляра).

Методы определяют поведение объекта. Объекты класса имеют общее поведение.

Объект

Объекты описываются не индивидуально, а с помощью классов. Класс - объект, являющийся шаблоном объекта.

Объект, созданный на основе некоторого класса, называется экземпляром класса.

Все объекты определенных пользователем классов являются экземплярами класса. Тем не менее, объекты даже с одним и тем же состоянием могут быть разными объектами.

Класс

```
class Student:
        ... pass
s=Student()
s.name="Vasya"
s.age=22
s.univer="AFTU"
print s.name, s.age, s.univer
```

Объекты могут содержат произвольное количество собственных данных

Определение методов класса

```
class Student :
        def f (self, n, y) :
                self.name=n
                self.year=y
                print self.name, "is on the", self.year, "-th year"
s1=Student()
s2=Student()
s1.f ("Vasya", 5)
s2.f ("Petya", 6)
```

Результат выполнения :

Vasya is on the 5th year

Petya is on the 6th year

Метод__init__()

Если для класса определен метод __init__(), то он авто матически вызывается сразу после создания экземпляра класса.

```
class Student :
        def __init__(self, c) :
                self.city=c
        def f (self, n, y) :
                self.name=n
                self.year=y
                print(self.name, "is on the", self.year, " year")
s=Student("Moscow")
print(s.city)
# St.Petersburg
```

Self

Первым аргументом каждого метода класса всегда является текущий экземпляр класса.

Self ссылается на экземпляр класса для которого вызывается метод.

В методе __init__ self ссылается на только что созданный объект.

Совсем необязательно, чтобы определение функции находилось в определении класса:

```
def f (self, x) :
        print "Hello,", x
class C :
        f1=f
        def f2 (self) :
        print "Hello, students!!!"
x=C()
x.f1("students!") # Hello, students!
x.f2() # Hello, students!!!
```

Наследование

```
class Person :
        def __init__(self, n) :
                self.name=n
        def write(self) :
                print(self.name)

class Student (Person) :
        def __init__(self, gr, n) :
                Person.__init__(self, n)
                self.group=gr
        def write(self) :
                print(self.name, self.group)
```

Наследование

```
p=Person("Petya")
p.write()       # Petya
s=Student(23, "Vasya")
s.write()       # Vasya 23
```

Атрибуты класса

```
class MyClass:
        'Simple Class'
        i=123
        def f(x) :
                print(“Hello!”)
print MyClass.i          # 123
print MyClass.__doc__    # Simple Class
MyClass.i=124
print MyClass.i          # 124
x=MyClass()
x.f()                    # Hello!
MyClass.f(x) # Hello!
x.i=125
y=MyClass()
print (x.i, y.i, MyClass.i)   # 125 124 124
```

Атрибуты класса

Атрибуты классов могут быть использованы как аналоги статических переменных классов в C ++:

```
class Counter:
        count = 0
        def __init__(self) :
                self.__class__.count += 1
print(Counter.count)                    # 0
c = Counter()
print(c.count, Counter.count)           # 1 1
d = Counter()
print(c.count, d.count, Counter.count) # 2 2 2
```

Частные атрибуты классов

Все атрибуты класса открытые (public).

Атрибуты, являются частными, доступными только из методов объекта класса, если их имена содержат не менее двух символов подчеркивания в начале и не более одного символа подчеркивания в конце:

```
class Student :
        def __init__ (self, c) :
                self.__city=c
        def __f (self, n, y) :
                self.name=n
                self.year=y
                print(self.name, "is on the", self.year, "- year»)
```

Частные атрибуты классов

```
s1=Student("Moscow")
print(s1.__city)
s1.__f ("Vanya", "5")
print(s1.__city)        # AttributeError: Student instance has no attribute '__city'
print s1._Student__city
s1._Student__f("Vanya", "5")
```

Результат выполнения :

```
St.Petersburg
Vanya is on the 5 th year
```

Множественное наследование

```
class A1 :
        def fb (self) :
                print "class1"
class A2 :
        def fb (self) :
                print "class2"
class B(A1, A2) :
        def f(self) :
                self.fb()
b=B()
b.f()
```

Задача

Определить класс Граф. Граф - это множество вершин и набор ребер, попарно соединяющий эти вершины. Над графом можно проделывать операции, такие как добавление вершины, ребра, проверка наличия ребра в графе и т.п.

Разбор задачи

```
class G:
        def __init__(self, V, E):
                self.vertices = set(V)
                self.edges = set(E)
        def add_vertex(self, v):
                self.vertices.add(v)
        def add_edge(self, v1, v2):
                self.vertices.add(v1)
                self.vertices.add(v2)
                self.edges.add(v1, v2)
        def has_edge(self, v1, v2):
                return (v1, v2) in self.edges
        def __str__(self):
                return "%s; %s" % (self.vertices, self.edges)
```

Разбор задачи

```
g = G([1, 2, 3, 4], [(1, 2), (2, 3), (2, 4)]) print g
g.add_vertex(5) g.add_edge(5,6)
print(g.has_edge(1,6))

print(g)
```

что даст в результате

```
Set([1, 2, 3, 4]); Set([(2, 4), (1, 2), (2, 3)])
False
Set([1, 2, 3, 4, 5, 6]); Set([(2, 4), (1, 2), (5, 6), (2, 3)])
```

Частные атрибуты классов

- Способ повлиять на доступ к атрибутам: методы

__getattr__(), __setattr__(), __delattr__() и *__getattribute__()*.

В отличие от дескрипторов их следует определять для объекта, **содержащего атрибуты** и вызываются они при доступе **к любому** атрибуту этого объекта.

__getattr__(self, name) будет вызван в случае, если запрашиваемый атрибут не будет найден

Атрибуты классов

```
class Strong:
        def __getattr__(self, attr):
                print(You, got ", attr)
                tellme = ”something”
        str = Strong()
        str.name = ”I Don’t know my name"
        str.five
        str.tellme
        str.name
```

Атрибуты классов

```
class NoStrong :
        attr = "class attribute"
        def __setattr__(self, name, val):
                print("not setting {0}={1}".format(name,val))
no_strong = NoStrong ()
no_strong.a= 1
no_strong.attr = 1
no_strong.__dict__
no_strong.attr
no_strong.a
```

Атрибуты классов

```
class NoStrong :
        attr = "class attribute"
        def __setattr__(self, name, val):
                print("not info {0}={1}".format(name,val))
no_strong = NoStrong ()
no_strong.a= 1
no_strong.attr = 1
no_strong.__dict__
no_strong.attr
no_strong.a
```

Пример

```
class A:
        pass a = A()
        a.attr = 1
        try:
                print(a.attr)
        except:
                print(None)
                del a.attr
```

Пример

```
class A:
        pass
a = A()
setattr(a, 'attr', 1)
if hasattr(a, 'attr'):
        print getattr(a, 'attr')
else:
        print(None)
delattr(a, 'attr')
```

Отношения между классами. Наследование

Допустим у нас в предметной области выделены очень близкие, но вместе с тем неодинаковые классы. Одним из способов сокращения описания классов за счет использования их сходства является выстраивание классов в иерархию.

В корне этой иерархии стоит базовый класс, от которого нижележащие классы иерархии наследуют свои атрибуты, уточняя и расширяя поведение вышележащего класса.

Например, класс Окружность в программе - графическом редакторе может быть унаследован от класса Геометрическая Фигура. При этом Окружность будет являться подклассом (или субклассом) для класса Геометрическая Фигура, а Геометрическая Фигура - надклассом (или суперклассом) для класса Окружность.

Принадлежность к классу

Принадлежность классу можно выяснить с помощью встроенной функции isinstance():

```
print(isinstance(g, G))
```

Доступ к свойствам

- Можно получить доступ к некоторому атрибуту (не методу) напрямую, если, конечно, этот атрибут описан как часть интерфейса класса. Такие атрибуты называются свойствами (properties).
- Отличительная черта Python в том, что в других языках программирования принято для доступа к свойствам создавать специальные методы (вместо того чтобы напрямую обращаться к общедоступным членам-данным).
- В Python достаточно использовать ссылку на атрибут, если свойство ни на что в объекте не влияет (то есть другие объекты могут его произвольно менять). Если же свойство менее тривиально и требует каких-то действий в самом объекте, его можно описать как свойство

Свойства

```
class C(object):
        def getx(self): return self.__x
        def setx(self, value): self.__x = value
        def delx(self): del self.__x
        x = property(getx, setx, delx, "I'm the 'x' property.")
```

Синтаксически доступ к свойству x будет обычной ссылкой на атрибут:

```
>>> c = C()
>>> c.x = 1
>>> print(c.x)
1
>>> del(c.x)
```

А на самом деле будут вызываться соответствующие методы: setx(), getx(), delx().

Сокрытие данных

Подчеркивание (" _ ") в начале имени атрибута указывает на то, что он не входит в общедоступный интерфейс т.е. "этот метод только для внутреннего использования".

Двойное подчеркивание работает как указание на то, что атрибут - приватный. При этом атрибут все же доступен, но уже под другим именем

```
>>> class X: ... x=0
        ...  _x = 0
        ...  __x = 0 ...
>>> dir(X)
['_X__x', '__doc__', '__module__', '_x', 'x']
```

Конструкторы

Большинство классов имеют специальный метод, который автоматически при создании объекта создает ему атрибуты. Т.е. вызывать данный метод не нужно, т.к. он сам запускается при вызове класса. (Вызов класса происходит, когда создается объект.)

Такой метод называется **конструктором класса**

Называется **__init__**

Конструкторы

Первым параметром, как и у любого другого метода, у **__init__** является **self**, на место которого подставляется объект в момент его создания. Второй и последующие (если есть) параметры заменяются аргументами, переданными в конструктор при вызове класса.

Конструкторы

```
class A:
        def __init__(self,one,two):
            self.fname = one
            self.sname = two
obj1 = A("Peter","Ok")
print(obj1.fname, obj1.sname)
```

Конструкторы

```
class B:
        def names(self,one,two):
                self.fname = one
                self.sname = two
                obj1 = B()
obj1.names("Peter","Ok")
print(obj1.fname, obj1.sname)
```

Конструкторы

В первом случае атрибуты инициализируются при создании объекта и должны передаваться в скобках при вызове класса. Если какие-то атрибуты должны присутствовать у объектов класса обязательно, то использование метода **__init__** - идеальный вариант.

Во второй программе (без использования конструктора) атрибуты создаются путем вызова метода names после создания объекта. В данном случае вызов метода names необязателен, поэтому объекты могут существовать без атрибутов fname и sname.

Конструкторы

```
class A:
    def __init__(self,one="noname",two="nonametoo"):
        self.fname = one self.sname = two
        obj1 = A("Sasha","Tu")
        obj2 = A()
        obj3 = A("Spartak")
        obj4 = A="Harry")

print (obj1.fname, obj1.sname)
print (obj2.fname, obj2.sname)
print (obj3.fname, obj3.sname)
print (obj4.fname, obj4.sname)
```

Bound и unbound методы

```
class Baby:
        def __init__(self):
                self.hungry = True
        def eat(self):
                if self.hungry:
                        print ('I am hangry... ')
                        self.hungry = False
                else:
                        print('No, thanks!')
class Rus(Baby):
        def __init__(self):
                country = 'RUS'
        def get_info(self):
                return self.county
```

Bound и unbound методы

Создаем экземпляр производного класса:

```
>>> c= Rus()
RUS
>>> c.eat()
AttributeError: Rus instance has no attribute 'hungry'
```

Конструктор производного класса — перегруженный, при этом конструктор базового класса не вызывается, и его нужно явно вызвать. Это можно сделать двумя путями.

Bound и unbound методы

Первый вариант считается устаревшим:

```
class Rus(Baby):
        def __init__(self):
                Baby.__init__(self)
                country = 'RUS'
        def get_info(self):
                return self.county
```

Здесь мы напрямую вызываем конструктор базового класса, не создавая инстанс базового класса Baby— поэтому такой базовый конструктор относится к категории unbound-методов. Для вызова bound-метода в качестве первого параметра методу нужно передать инстанс класса.

Ещё раз о свойствах

Property — атрибут класса (возвращается через функцию property), которая в качестве аргументов принимает другие функции класса:

```
class DateOffset:
        def __init__(self):
                self.start = 0
        def _get_offset(self):
                self.start += 5
                return self.start
        offset = property(_get_offset)
>>> d = DateOffset()
>>> d.offset
5
>>> d.offset
10
```

Контейнеры

Под контейнером обычно понимают объект, основным назначением которого является хранение и обеспечение доступа к другим объектам. Встроенные типы, список и словарь -- яркие примеры контейнеров. Можно построить собственные типы контейнеров, которые будут иметь свою логику доступа к хранимым объектам. В контейнере хранятся не сами объекты, а ссылки на них.

Пример

Если необходимо, можно создать свои реализации.

Напишем класс Стек, реализованный на базе списка:

Пример

```
class Stack: def __init__(self):
                """Инициализация стека"""
                self._stack = []
        def top(self):
                """Возвратить вершину стека (не снимая)"""
                return self._stack[-1]
        def pop(self):
                """Снять со стека элемент"""
                return self._stack.pop()
        def push(self, x):
                """Поместить элемент на стек"""
        self._stack.append(x) def __len__(self):
                """Количество элементов в стеке"""
                return len(self._stack)
        def __str__(self):
                """Представление в виде строки"""
                return " : ".join(["%s" % e for e in self._stack])
```

Пример

Использование:

```
>>> s = Stack()
>>> s.push(1)
>>> s.push(2)
>>> s.push("abc")
>>> print s.pop()
abc >>>
print(len(s))
2
```

ОО и Процедурный подход

При процедурном подходе при появлении нового метода пишется отдельная процедура, в которой в каждой ветке алгоритма обрабатывается свой тип данных (то есть такое изменение требует редактирования одного места в коде).

При ООП изменять придется каждый класс, внося в него новый метод (то есть изменения в нескольких местах). Зато ООП выигрывает при внесении нового типа данных: ведь изменения происходят только в одном месте, где описываются все методы для данного типа.

При процедурном подходе приходится изменять несколько

Пример

Пусть имеются классы A, B, C и методы a, b, c:

```
# ООП
class A:
… def a():
... def b():
... def c():
class B:
…def a():
... def b():
... def c():
```

Пример

```
# процедурный подход
def a(x):
… if type(x) is A:
... if type(x) is B:
…
def b(x):
… if type(x) is A:
... if type(x) is B:
```

Пример

При внесении нового типа объекта изменения в ОО-программе затрагивают только один модуль, а в процедурной - все процедуры:

ООП

class C:

… def a():

... def b():

Пример

```
# процедурный подход
def a(x):
… if type(x) is A:
... if type(x) is B:
… if type(x) is C:
def b(x):
… if type(x) is A:
... if type(x) is B:
… if type(x) is C:
def c(x):
… if type(x) is A:
... if type(x) is B:
... if type(x) is C:
```

Декораторы

Функции являются объектами, соответственно, их можно возвращать из другой функции или передавать в качестве аргумента.

Функция в python может быть определена и внутри другой функции.

Декоратор — это, "обёртка", которая позволяет изменить поведение функции, не изменяя её код.

Декораторы

```
>>> def getInfo():
        print(1)
>>> def global_decorator(function):
        def before_function():
            print("До вызова ф-ии")
            function()
            print("После вызова ф-ии")
            return before_function
>>> getInfo()
"Some info"
>>> getInfo_decorated = global_decorator(getInfo)
>>> getInfo_decorated()
"До вызова ф-ии"
1
"После вызова ф-ии"
```

Декораторы

```
>>> def getInfo2():
        print(2)

>>> @global_decorator
>>> getInfo2()
"До вызова ф-ии"
2
"После вызова ф-ии"
```

Исключения

```
>>>def average(num_list):
        return sum(num_list) / len(num_list)
>>>average([])
ZeroDivisionErrorTraceback (most recent call last)


try:
        average([])
except ZeroDivisionError:
         print('Error occurred')
```

Исключения

```
try:
        average([])
except ZeroDivisionError as e:
        print(e)
```

Исключения

```
try:
        average([])
except ZeroDivisionError:
        print('Oops')
else:
        print('It\'s OK!')
finally:
        print('Hello there')
```