Объектно-ориентированное программирование в Python

* **Основные свойства ООП** — полиморфизм, наследование, инкапсуляция.

* **Полиморфизм**: в разных объектах одна и та же операция может выполнять различные функции.
* **Инкапсуляция**: можно скрыть ненужные внутренние подробности работы объекта от окружающего мира.
* **Наследование**: можно создавать специализированные классы на основе базовых.
* **(*)Композиция**: объект может быть составным и включать в себя другие объекты.

Создание классов:

Класс — это пользовательский тип.

Для создания классов предусмотрен оператор **class**.

В терминологии Питона члены класса называются **атрибутами**, функции класса — **методами.**

* **class** ИМЯКЛАССА:
    ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
  ….
    def ИМЯМЕТОДА(self, ...):
      self.ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ …
  * ….

*В Питоне класс не является чем-то статическим после определения, поэтому добавить атрибуты можно и после:

Пример:

```
class A:
    pass
def myMethod(self, x):
    return x ** 2
A.m1 = "My IQ is:"
A.m2 = myMethod
b = A()
print b.m1 + ' ' + str(b.m2(9))
```

* Для инстанциирования класса, то есть, создания экземпляра класса, достаточно вызвать класс по имени и задать параметры конструктора.

Делается это с помощью оператора _init_.

Первым параметром, как и у любого другого метода, у **__init__** является **self**, на место которого подставляется объект в момент его создания.

```
class YesInit:
    def __init__ (self,one,two):
        self.fname = one
        self.sname = two

obj1 = YesInit(“Chris",“Rock")
print obj1.fname + ' ' + obj1.sname
    >>
Chris Rock
```

* Однако создание объекта класса оператором _init_ предполагает передачу аргументов.
* Если аргументы не переданы, то присходит ошибка. Поэтому можно присваивать параметры значениям по умолчанию.

```
class YesInit:
    def __init__(self,one="noname",two="nonametoo"):
        self.fname = one
        self.sname = two
Obj1 = YesInit()
print obj.one + ” ” + obj.two
    >>
noname nonametoo
```

* Специальные методы вызываются при создании экземпляра класса и при удалении класса (деструктор). В питоне реализовано автоматическое управление памятью, поэтому деструктор требуется достаточно редко, для ресурсов, требующих явного освобождения.

* Следующий класс имеет конструктор и деструктор:

```
class Line:
    def __init__(self, p1, p2):
        self.line = (p1, p2)
    def __del__(self):
        print "Удаляется линия %s - %s" % self.line
>>> l = Line((0.0, 1.0), (0.0, 2.0))
>>> del  l
Удаляется линия (0.0, 1.0) - (0.0, 2.0)
```

Наследование

Простое наследование:

Класс, от которого произошло наследование, называется базовым или родительским. Классы, которые произошли от базового, называются потомками, наследниками или производными классами.

Множественное наследование:

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости.

#Множественное наследование — потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках.

* Python поддерживает как одиночное наследование, так и множественное, позволяющее классу быть производным от любого количества базовых классов.

```
class Par1(object):
# наследуем один базовый класс - object
        def name1 (self): return 'Par1'
 class Par2 (object):
        def name2 (self): return 'Par2'
class Child (Par1, Par2):
# создадим класс, наследующий Par1, Par2 (и, опосредованно, object)
        pass
x = Child()
print x.name1() + ‘  ’+ x.name2()
# экземпляру Child доступны методы из Par1 и Par2
'Par1','Par2'
```

* Типичная проблема,возникающая при проектировании ооп,состоит в следующем. Объект некоторого типа требуется передавать в качестве аргумента в различные функции.
* Разным функциям нужны разные свойства и методы этого объекта, которые нужны каждой функции, фиксирован.
* При этом хотелось бы все эти функции сделать полиморфными, то есть способными принимать объекты разных типов.

* В Питоне используется концепция, называемая Duck Typing:

” Если ЭТО ходит как утка, и крякает,как утка – значит это утка”.

Т.е., если у объекта есть все нужные функции, свойства и методы, то он подходит в качестве аргумента.

Например, в функцию

```
def f(x):
    return x.get_value_()
```

Можно передавать объект любого типа,лишь бы у него был метод get_value().

* Ещё одна проблема, возникающая в связи с множественным наследованием – не всегда очевидно, в каком порядке будут просматриваться родительские классы в поисках нужного свойства или метода. В Питоне для упрощения этой проблемы у каждого класса есть свойство __mro__(method resolution order):

```
*>>>class A(object): pass
...
*>>>class B(object): pass
*... x = 0
*...
*>>>class C(A,B):
*... z = 3
*...
*>>>C.__mro__
(<class 'C'>, <class 'A'>, <class 'B'>, <type object>)
```

* В Python есть метод super(), который обычно применяется к объектам. Его главная задача это возможность использования в классе потомке, методов класса-родителя.class Child(Parent):

* Пример:

```
def __init__(self):
        super(Child, self).__init__(self)
class A(object):
     def __init__(self):
             print(u'конструктор класса A')
# Потомок класса A
class B(A):
    def __init__(self):
        print(u'конструктор класса B')
        super(B,self).__init__()
```

* #Смысл примера заключается в том, что Python не запустит родительский конструктор, поскольку мы его переопределили в классе B… Поэтому методом super() мы явно вызываем родительский конструктор.

Свойства super():

```
class C(B, A):
    def __init__(self):
        # something
        super(C,self).__init__()
```

По сути, объект класса super запоминает аргуметы переданные ему в момент инициализации и при вызове любого метода (super().__init__(self) в примере выше) проходит по списку линеаризации класса второго аргумента (self.__class__.__mro__), пытаясь вызвать этот метод по очереди для всех классов, следующих за классом в первом аргументе (класс C), передавая в качестве параметра первый аргумент (self). Т.е. для нашего случая:

self.__class__.__mro__ = [C, B, A, P1, P2, …]
super(C, self).__init__() => B.__init__(self)
super(B, self).__init__() => A.__init__(self)
super(A, self).__init__() => P1.__init__(self)

* Изменяя атрибут __class__, можно перемещать объект вверх или вниз по иерархии наследования (впрочем, как и к любому другому типу)

```
>>>c = child()
 >>c.val = 10
 >>c.who()
'child'
>>> c.__class__ = parent
>>> c.who()
'parent'
>>> c.val
10
```

Полиморфизм

В ООП программировании этим термином обозначают возможность использования одного и того же имени операции или метода к объектам разных классов, при этом действия, совершаемые с объектами, могут существенно различаться.

* В компилируемых языках программирования полиморфизм достигается за счет создания виртуальных методов, которые в отличие от невиртуальных можно перегрузить в потомке. В Питоне все методы являются виртуальными, что является естественным следствием разрешения доступа на этапе исполнения.

```
class Parent(object):
        def isParOrPChild(self) : return True
        def who(self) : return 'parent'
class Child(Parent):
        def who(self): return 'child'
>>> x = Parent()
>>> x.who(), x.isParOrPChild()
('parent', True)
>>> x = Child()
>>> x.who(), x.isParOrPChild()
('child', True)
```

* Два разных класса могут содержать метод (например total) , однако инструкции в методах могут предусматривать совершенно разные операции.

```
class T1:
    n=10
    def total(self,N):
        self.total = int(self.n) + int(N)

class T2:
    def total(self,s):
        self.total = len(str(s))

t1 = T1()
t2 = T2()
t1.total(45)
t2.total(45)
print (t1.total) # Вывод: 55
print (t2.total) # Вывод: 2
```

Переопределение методов

Использование полиморфизма при наследовании классов позволяет переопределять методы суперклассов их подклассами. Например, может возникнуть ситуация, когда все подклассы реализуют определенный метод из суперкласса, и лишь один подкласс должен иметь его другую реализацию. В таком случае метод переопределяется в подклассе.

*Пример:

```
class Base:
    def __init__(self,n):
        self.numb = n
    def out(self):
        print (self.numb)
class One(Base):
    def multi(self,m):
        self.numb *= m
class Two(Base):
    def inlist(self):
        self.inlist = list(str(self.numb))
    def out(self):
        i = 0
        while i < len(self.inlist):
            print (self.inlist[i])
            i += 1
```

Продолжение программы:

```
obj1 = One(45)
obj2 = Two('abc')

obj1.multi(2)
obj1.out() # Вывод числа 90

obj2.inlist()
obj2.out() # Вывод в столбик букв a, b, c
```

Инкапсуляция и доступ к свойствам

Инкапсуляция — свойство языка программирования, позволяющее объединить и защитить данные и код в объекте и скрыть реализацию объекта от пользователя. При этом пользователю предоставляется только спецификация (интерфейс) объекта.

Пользователь может взаимодействовать с объектом только через этот интерфейс.

Пользователь не может использовать закрытые данные и методы.

* Одиночное подчеркивание в начале имени атрибута говорит о том, что метод не предназначен для использования вне методов класса (или вне функций и классов модуля), однако, атрибут все-таки доступен по этому имени. Два подчеркивания в начале имени дают несколько большую защиту: атрибут перестает быть доступен по этому имени. Последнее используется достаточно редко.

* Есть существенное отличие между такими атрибутами и личными (private) членами класса в таких языках как C++ или Java: атрибут остается доступным, но под именем вида _ИмяКласса__ИмяАтрибута, а при каждом обращении Python будет модифицировать имя в зависимости от того, через экземпляр какого класса происходит обращение к атрибуту. Таким образом, родительский и дочерний классы могут иметь атрибут с именем, например, «__f», но не будут мешать друг другу.

* Пример:

```
class parent(object):
      def __init__(self):
          self.__f = 2
      def get(self):return self.__f
      ....
class child(parent):
      def __init__(self):
        self.__f = 1
        parent.__init__(self)
      def cget(self):return self.__f
      ....
 c = child()
print  c.get()
     2
print  c.cget()
     1
print c.__dict__
     {'_child__f': 1, '_parent__f': 2}
 # на самом деле у объекта "c" два разных атрибута
```

Агрегация. Контейнеры. Итераторы

* Агрегация, когда один объект входит в состав другого, или отношение «HAS-A» («имеет»), реализуется в Питоне с помощью ссылок. Питон имеет несколько встроенных типов контейнеров: список, словарь, множество. Можно определить собственные классы контейнеров со своей логикой доступа к хранимым объектам. (Следует заметить, что в Питон агрегацию можно считать разновидностью ассоциации, так реально объекты не вложены друг в друга в памяти и, более того, время жизни элемента может не зависеть от времени жизни контейнера.)

* Следующий класс из модуля utils.py среды web.py является примером контейнера-словаря, дополненного возможностью доступа ко значениям при помощи синтаксиса доступа к атрибутам:

```
* class Storage(dict):
*     def __getattr__(self, key):
*         try:
*             return self[key]
*         except KeyError, k:
*             raise AttributeError, k
*
*     def __setattr__(self, key, value):
*         self[key] = value
*
*     def __delattr__(self, key):
*         try:
*             del self[key]
*         except KeyError, k:
*             raise AttributeError, k
*
*     def __repr__(self):
```

Вот как он работает:

```
>>> v = Storage(a=5)
>>> v.a
5
>>> v['a']
5
>>> v.a = 12
>>> v['a']
12
>>> del v.a
```

* Атрибут __dict__ служит для предоставления пользователю информации о классе, экземляре класса, методах класса

*Пример:

```
class falt:
  #Hi People of this cruel world
    people = ‘oxygen'
     math =  ‘’
   def star(self):
       return self.math + ‘ is not ’  + self.people
print falt.__dict__
>> {'__module__': '__main__', '__doc__': None, '__str__':  ‘star': ‘<function star at 0x01FE57B0>}
a = falt()
print a. __dict__
>> {}
a.people = 10
print falt.__dict__
>> {‘people’: 10}
```

* Атрибут __doc__

Служит для показа комментариев в библиотеках, классах, функциях, файлах:

```
Пример: class falt:
        '''Hi, people of this world'''
          people = 'oxygen'
           math =  ''
         def star(self):
         '''It's me again'''
            return self.math + ' is not '  + self.people
print falt.__doc__
print falt.star. __doc__
>> Hi people of this cruel world
     It's me again
```

```
import urllib
…
print urllib.__doc__ #Выведет документацию urllib
```

*Метод __str__:

*Служит для задания фомативного вывода при вызове print и подобных случаях:

Пример:

```
class joker:
    pass
    def __str__(self):
        return "This is me %s" % self.__dict__
frag1 = joker()
frag1. name = ‘Tommy’
print frag1
>> This is me {‘name’: ‘Tommy’}
```

V sobake kot 0_o?!