**Объектно-ориентированное программирование**

**Объектно-ориентированное программирование (ООП)** — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

**Принципы ООП**

Объектно-ориентированный язык работает по следующим принципам:

* Все данные представляются объектами
* Программа является набором взаимодействующих объектов, посылающих друг другу сообщения
* Каждый объект имеет собственную часть памяти и может иметь в составе другие объекты
* Каждый объект имеет тип
* Объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия)

**Идеи/принципы объектно-ориентированного программирования:**

**Наследование.** Возможность выделять общие свойства и методы классов в один класс верхнего уровня (родительский). Классы, имеющие общего родителя, различаются между собой за счет включения в них различных дополнительных свойств и методов.

**Инкапсуляция.** Свойства и методы класса делятся на доступные из вне (опубликованные) и недоступные (защищенные). Защищенные атрибуты нельзя изменить, находясь вне класса. Опубликованные же атрибуты также называют интерфейсом объекта, т. к. с их помощью с объектом можно взаимодействовать. По идеи, инкапсуляция призвана обеспечить надежность программы, т.к. изменить существенные для существования объекта атрибуты становится невозможно.

**Полиморфизм.** Полиморфизм подразумевает замещение атрибутов, описанных ранее в других классах: имя атрибута остается прежним, а реализация уже другой. Полиморфизм позволяет специализировать (адаптировать) классы, оставляя при этом единый интерфейс взаимодействия.

**Преимущества ООП**

* Использование одного и того же программного кода с разными данными. Классы позволяют создавать множество объектов, каждый из которых имеет собственные значения атрибутов. Нет потребности вводить множество переменных, т.к объекты получают в свое распоряжение индивидуальные так называемые пространства имен. Пространство имен конкретного объекта формируется на основе класса, от которого он был создан, а также от всех родительских классов данного класса. Объект можно представить, как некую упаковку данных.
* Наследование и полиморфизм позволяют не писать новый код, а настраивать уже существующий, за счет добавления и переопределения атрибутов. Это ведет к сокращению объема исходного кода.

**Особенность ООП**

ООП позволяет сократить время на написание исходного кода, однако ООП всегда предполагает большую роль предварительного анализа предметной области, предварительного проектирования. От правильности решений на этом предварительном этапе зависит куда больше, чем от непосредственного написания исходного кода.

**Особенности ООП в Python**

По сравнению с другими распространенными языками программирования у Python можно выделить следующие особенности, связанные с объектно-ориентированным программированием:

Любое данное (значение) — это объект. Число, строка, список, массив и др. — все является объектом. Бываю объекты встроенных классов (как те,что перечисленные в предыдущем предложении), а бывают объекты пользовательских классов (тех, что создает программист). Для единого механизма взаимодействия предусмотрены методы перегрузки операторов.

**Класс** — это тоже объект с собственным пространством имен. Это нигде не было указано в данном цикле уроков. Однако это так. Поэтому правильнее было употреблять вместо слова «объект», слово «экземпляр». И говорить «экземпляр объекта», подразумевая под этим созданный на основе класса именно объект, и «экземпляр класса», имея ввиду сам класс как объект.

Инкапсуляции в Python не уделяется особого внимания. В других языках программирования обычно нельзя получить напрямую доступ к свойству, описанному в классе. Для его изменения может быть предусмотрен специальный метод. В Python же это легко сделать, просто обратившись к свойству класса из вне. Несмотря на это в Python все-таки предусмотрены специальные способы ограничения доступа к переменным в классе.

**Обзор терминологии ООП**

**Класс** — определенный пользователем прототип для объекта, который определяет набор атрибутов, которые характеризуют любой объект класса. Атрибутами являются члены данных (переменные класса и переменные экземпляра) и методы, доступ к которым осуществляется через точечную запись.

**Переменная класса** — переменная, которая используется всеми экземплярами класса. Переменные класса определены внутри класса, но вне любого из методов класса. Переменные класса используются не так часто, как переменные экземпляра.

**Член данных** — переменная класса или переменная экземпляра, которая содержит данные, связанные с классом и его объектами.

**Перегрузка функций** — назначение более чем одного поведения определенной функции. Выполняемая операция варьируется в зависимости от типов объектов или аргументов.

**Переменная экземпляра** — переменная, которая определена внутри метода и принадлежит только текущему экземпляру класса.

**Наследование** — передача характеристик класса другим классам, которые являются его производными.

**Экземпляр** — индивидуальный объект определенного класса. Например, объект obj, принадлежащий классу Circle, является экземпляром класса Circle.

Instantiation — создание экземпляра класса.

**Метод** — особый вид функции, который определен в определении класса.

**Объект** — уникальный экземпляр структуры данных, который определяется его классом. Объект включает в себя как члены данных (переменные класса и переменные экземпляра), так и методы.

**Перегрузка оператора** — назначение более чем одной функции определенному оператору.

Python соответствует принципам объектно-ориентированного программирования. В python всё является объектами - и строки, и списки, и словари, и всё остальное.

Для создания классов предусмотрена инструкция class. Тело класса состоит из блока различных инструкций.

class ИмяКласса:

ПеременнаяКласса = Значение

…

def ИмяМетода(self, ...):

self.ПеременнаяКласса = Значение

…

…

**Методы в классах** — это те же функции, которые принимают один обязательный параметр — self (с англ. можно перевести как «собственная личность»). Он нужен для связи с конкретным объектом.

**Атрибуты класса** — это имена переменных вне функций и имена функций. Эти атрибуты наследуются всеми объектами, созданными на основе данного класса. Атрибуты обеспечивают свойства и поведение объекта. Объекты могут иметь атрибуты, которые создаются в теле метода, если данный метод будет вызван для конкретного объекта.

Пример класса (ООП) на Python 3:

class Second:

color = "red"

form = "circle"

def changecolor(self,newcolor):

self.color = newcolor

def changeform(self,newform):

self.form = newform

obj1 = Second()

obj2 = Second()

print (obj1.color, obj1.form) # вывод на экран "red circle"

print (obj2.color, obj2.form) # вывод на экран "red circle"

obj1.changecolor("green") # изменение цвета первого объекта

obj2.changecolor("blue") # изменение цвет второго объекта

obj2.changeform("oval") # изменение формы второго объекта

print (obj1.color, obj1.form) # вывод на экран "green circle"

print (obj2.color, obj2.form) # вывод на экран "blue oval"

Результат выполнения скрипта Python 3:

red circle

red circle

green circle

blue oval

Конструктор класса — метод \_\_init\_\_

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 и в языке программирования Python носит имя \_\_init\_\_. (В начале и конце по два знака подчеркивания.)

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.

**Примеры.**

Есть два класса: в одном будет использоваться конструктор, а в другом нет. Требуется создать два атрибута объекта.

Пример 1:

class YesInit:

def \_\_init\_\_(self,one,two):

self.fname = one

self.sname = two

obj1 = YesInit("Peter","Ok")

print (obj1.fname, obj1.sname)

Пример 2:

class NoInit:

def names(self,one,two):

self.fname = one

self.sname = two

obj2 = NoInit()

obj2.names("Peter","Ok")

print (obj2.fname, obj2.sname)

Результат выполнения двух скриптов:

Peter Ok

Что значит аргумент self в Python 3 в методе класса

Аргумент self — это ссылка на создаваемый в памяти компьютера объект.

**Методы класса** — это небольшие программки, предназначенные для работы с объектами. Методы могут создавать новые свойства (данные) объекта, изменять существующие, выполнять другие действия над объектами.

Методу необходимо «знать», данные какого объекта ему предстоит обрабатывать. Для этого ему в качестве первого (а иногда и единственного) аргумента передается имя переменной, связанной с объектом (можно сказать, передается сам объект). Чтобы в описании класса указать передаваемый в дальнейшем объект, используется параметр self.

С другой стороны, вызов метода для конкретного объекта в основном блоке программы выглядит следующим образом:

Объект.ИмяМетода(…)

Здесь под словом Объект имеется в виду переменная, связанная с ним. Это выражение преобразуется в классе, к которому относится объект, в

ИмяМетода(Объект, …)

Т.е. конкретный объект подставляется вместо параметра self

**Еще примеры**

**Создание собственного класса**

... class A:

... pass

**Создание нескольких экземпляров этого класса:**

>>>

>>> a = A()

>>> b = A()

>>> a.arg = 1 # у экземпляра a появился атрибут arg, равный 1

>>> b.arg = 2 # а у экземпляра b - атрибут arg, равный 2

>>> print(a.arg)

1

>>> print(b.arg)

2

>>> c = A()

>>> print(c.arg) # а у этого экземпляра нет arg

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'A' object has no attribute 'arg'

Классу возможно задать собственные методы:

>>>

>>> class A:

... def g(self): # self - обязательный аргумент, содержащий в себе экземпляр

... # класса, передающийся при вызове метода,

... # поэтому этот аргумент должен присутствовать

... # во всех методах класса.

... return 'hello world'

...

>>> a = A()

>>> a.g()

'hello world'

Пример:

>>>

>>> class B:

... arg = 'Python' # Все экземпляры этого класса будут иметь атрибут arg,

... # равный "Python"

... # Но впоследствии мы его можем изменить

... def g(self):

... return self.arg

...

>>> b = B()

>>> b.g()

'Python'

>>> B.g(b)

'Python'

>>> b.arg = 'spam'

>>> b.g()

'spam'

Вместо использования обычных операторов для доступа к атрибутам, можное использовать следующие функции:

GetAttr (объект, имя [, по умолчанию]) — для доступа к атрибуту объекта.

Hasattr (объект, имя) — проверить , если атрибут существует или нет.

SetAttr (объект, имя, значение) — установить атрибут. Если атрибут не существует, он будет создан.

Delattr (объект, имя) — для удаления атрибута.

**Встроенные атрибуты класса**

Каждый класс Python поддерживает следующие встроенные атрибуты, и к ним можно получить доступ, используя оператор точки, как и любой другой атрибут —

\_\_dict\_\_ — словарь, содержащий пространство имен класса.

\_\_doc\_\_ — Строка документации класса или нет, если она не определена.

\_\_name\_\_ — Имя класса.

\_\_module\_\_ — Имя модуля, в котором определяется класс. Этот атрибут «\_\_main\_\_» в интерактивном режиме.

\_\_bases\_\_ — возможно пустой кортеж, содержащий базовые классы, в порядке их появления в списке базовых классов.

**Наследование классов**

Вместо того, чтобы начинать с нуля, вы можете создать класс, выведя его из ранее существовавшего класса, перечислив родительский класс в скобках после имени нового класса.

Дочерний класс наследует атрибуты своего родительского класса, и вы можете использовать эти атрибуты, как если бы они были определены в дочернем классе. Дочерний класс также может переопределять элементы данных и методы родительского класса.

**Синтаксис**

Производные классы объявляются так же, как их родительский класс; однако список базовых классов для наследования дается после имени класса

class SubClassName (ParentClass1[, ParentClass2, ...]):

'Optional class documentation string'

class\_suite

Пример

#!/usr/bin/python

class Parent: # define parent class

parentAttr = 100

def \_\_init\_\_(self):

print "Calling parent constructor"

def parentMethod(self):

print 'Calling parent method'

def setAttr(self, attr):

Parent.parentAttr = attr

def getAttr(self):

print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class

def \_\_init\_\_(self):

print "Calling child constructor"

def childMethod(self):

print 'Calling child method'

c = Child() # instance of child

c.childMethod() # child calls its method

c.parentMethod() # calls parent's method

c.setAttr(200) # again call parent's method

c.getAttr() # again call parent's method

Когда приведенный выше код выполняется, он дает следующий результат

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

Аналогичным образом можно управлять классом из нескольких родительских классов следующим образом:

class A: # define your class A

.....

class B: # define your class B

.....

class C(A, B): # subclass of A and B

.....

Можно использовать функции issubclass () или isinstance (), чтобы проверить отношения двух классов и экземпляров.

Issubclass ( к югу, вир) функция булева возвращает истину , если данный подкласс суб действительно подкласс суперкласса вир .

Isinstance (объект, класс) Функция булева возвращает истину, если OBJ является экземпляром класса Class или является экземпляром подкласса класса

**Переопределяющие методы**

Можно переопределить родительские методы класса. Одна из причин переопределения родительских методов заключается в том, что может потребоваться особая или другая функциональность в подклассе.

Пример

#!/usr/bin/python

class Parent: # define parent class

def myMethod(self):

print 'Calling parent method'

class Child(Parent): # define child class

def myMethod(self):

print 'Calling child method'

c = Child() # instance of child

c.myMethod() # child calls overridden method

Когда приведенный выше код выполняется, он дает следующий результат

Calling child method

**Базовые методы перегрузки**

В следующей таблице перечислены некоторые общие функции, которые вы можете переопределить в своих собственных классах.

|  |  |
| --- | --- |
| **Sr.No.** | **Метод, описание и пример вызова** |
| 1 | **\_\_init\_\_ (self [, args …])**  Конструктор (с любыми необязательными аргументами)  Пример вызова: *obj = className (args)* |
| 2 | **\_\_del \_\_ (самостоятельно)**  Деструктор, удаляет объект  Образец звонка: *del obj* |
| 3 | **\_\_repr \_\_ (самостоятельно)**  Оцениваемое строковое представление  Пример вызова: *repr (obj)* |
| 4 | **\_\_str \_\_ (самостоятельно)**  Печатное представление строки  Пример вызова: *str (obj)* |
| 5 | **\_\_cmp\_\_ (self, x)**  Сравнение объектов  Пример вызова: *cmp (obj, x)* |

**Операторы перегрузки**

Если создать класс Vector для представления двумерных векторов и добавить оператор «плюс», то, скорее всего, Python выдаст ошибку.

Но если определить метод \_\_add\_\_ в классе для выполнения сложения векторов, то оператор плюс будет вести себя так, как ожидалось:

пример

#!/usr/bin/python

class Vector:

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def \_\_str\_\_(self):

return 'Vector (%d, %d)' % (self.a, self.b)

def \_\_add\_\_(self,other):

return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)

v2 = Vector(5,-2)

print v1 + v2

Когда приведенный выше код выполняется, он дает следующий результат

Vector(7,8)

**Скрытие данных**

Атрибуты объекта могут или не могут быть видны вне определения класса. Необходимо присвоить имена атрибутам с двойным префиксом подчеркивания, и тогда эти атрибуты не будут напрямую видны посторонним.

пример

#!/usr/bin/python

class JustCounter:

\_\_secretCount = 0

def count(self):

self.\_\_secretCount += 1

print self.\_\_secretCount

counter = JustCounter()

counter.count()

counter.count()

print counter.\_\_secretCount

Когда приведенный выше код выполняется, он дает следующий результат

1

2

Traceback (most recent call last):

File "test.py", line 12, in <module>

print counter.\_\_secretCount

AttributeError: JustCounter instance has no attribute '\_\_secretCount'

Python защищает этих членов, внутренне изменяя имя, чтобы включить имя класса. Можно получить доступ к таким атрибутам как object.\_className\_\_attrName . Если заменить последнюю строку следующим образом, то она заработает

.........................

print counter.\_JustCounter\_\_secretCount

Когда приведенный выше код выполняется, он дает следующий результат

1

2

2