Python

Урок 5. Модули, пакеты, классы и объекты

Модули

Как организована программа

Как правило, программа на языке Python состоит из множества текстовых файлов, содержащих инструкции. Программа организована как один главный файл, к которому могут подключаться дополнительные файлы, известные как модули.

Главный файл определяет, как будет двигаться основной поток выполнения программы, – это тот файл, который необходимо запустить, чтобы начать работу приложения. Файлы модулей – это библиотеки инструментальных средств, где содержатся компоненты, используемые главным файлом (и, возможно, где-то еще). Главный файл использует инструменты, определенные в файлах модулей, а модули используют инструменты, определенные в других модулях.

Чтобы получить доступ к определенным в модуле инструментам, именующимся *атрибутами* модуля (имена переменных, связанные с такими объектами, как функции), в языке Python необходимо *импортировать* этот модуль.

Импорт

```
import module
```

Фактически имя модуля, используемое в инструкции import, во-первых, идентифицирует внешний файл и, во-вторых, становится именем переменной, которая будет представлять загруженный модуль.

Объекты, определяемые модулем, также создаются во время выполнения, когда производится импорт модуля: инструкция import, в действительности,

последовательно выполняет инструкции в указанном файле, чтобы воссоздать его содержимое.

Любой файл может импортировать функциональные возможности из любого другого файла.

Импорт выполяет следующие операции:

1. Отыскивает файл модуля.
2. Компилирует его в байт-код (если это необходимо).
3. Запускает программный код модуля, чтобы создать объекты, которые он определяет.

Пути поиска модулей

В общих чертах пути поиска модулей в языке Python выбираются из объединенных данных следующих основных источников:

1. Домашний каталог программы.
2. Содержимое переменной окружения PYTHONPATH (если таковая определена).
3. Каталоги стандартной библиотеки.
4. Содержимое любых файлов с расширением .pht (если таковые имеются).

В конечном итоге объединение этих четырех компонентов составляет sys.path – список строк с именами каталогов.

Если вам потребуется узнать, как выглядит путь поиска на вашей машине, вы всегда сможете сделать это, просмотрев содержимое встроенного списка sys. path:

```
>>> import sys
>>> sys.path
['', 'C:\\users', 'C:\\Windows\\system32\\python33.zip', 'c:\
 \Python33\\DLLs', 'c:\\Python33\\lib', 'c:\\Python33\\lib\\plat-
 win', 'c:\\Python33', 'C:\\Users\\Mark', 'c:\\Python33\\lib\
 \site-packages']
```

import и from

Для использования файла модуля нужно выполнить инструкцию import или from. Обе инструкции отыскивают, компилируют и запускают программный код модуля, если он еще не был загружен. Главное различие этих инструкций

заключается в том, что инструкция import загружает модуль целиком, поэтому при обращении к именам в модуле их необходимо дополнять именем модуля. Инструкция from, напротив, загружает (или копирует) из модуля отдельные имена.

```
import re
re.findall(...)
```

```
from re import findall
findall(...)
```

После выполнения инструкции import создается переменная, указывающая на созданный в памяти объект модуля.

```
from re import *
findall(...)
```

from * копирует все имена в импортируемом модуле в область видимости, из которой производится импорт.

Инструкции import и from являются выполняемыми инструкциями, а не объявлениями времени компиляции. Они могут вкладываться в условные инструкции if, присутствовать в объявлениях функций def и так далее, и они не имеют никакого эффекта, пока интерпретатор не достигнет их в ходе выполнения программы.

Пространство имен модуля

Модули – это всего лишь пространства имен (места, где создаются имена), и имена, находящиеся в модуле, называются его атрибутами.

Доступ к пространствам имен модулей можно получить через атрибут __dict__ или dir(M).

Операция импортирования никогда не изменяет область видимости для программного кода в импортируемом файле – из импортируемого файла нельзя получить доступ к именам в импортирующем файле.

Двойное импортирование

В действительности, когда mod1 импортирует mod2, он создает двухуровневое вложение пространств имен. Используя полный путь к имени mod2.mod3.X, он

может погрузиться в модуль mod3, который вложен в импортированный модуль mod2. Суть в том, что модуль mod1 может обращаться к переменным X во всех трех файлах и, следовательно, имеет доступ ко всем трем глобальным областям видимости.

Однако обратное утверждение неверно: модуль mod3 не имеет доступа к именам в mod2, а модуль mod2 не имеет доступа к именам в mod1.

Атрибуты модулей

У каждого модуля есть набор специфических атрибутов:

Атрибут	Описание
__name__	Полное имя модуля. Путь от начала с точками как разделителями. Например, 'xml.dom' или 'xml.dom.minidom'
__doc__	описание (так называемый docstring)
__file__	полный путь к файлу, из которого модуль был создан (загружен). До версии Python 3.2 это путь к .py или .pyc (записанный на диск кеш, результат автоматической компиляции кода модуля, используется для ускорения загрузки). Начиная с 3.2 __file__ всегдауказывает на исходный .py файл
__path__	список файловых путей, в которых находится пакет. Существует только для пакетов.
__package__	имя пакета, в котором лежит модуль (пустая строка для модулей верхнего уровня). Появился для поддержки относительного импорта from . import a.

Пакеты

Помимо возможности импортировать имя модуля существует возможность импортировать имена каталогов. Каталог на языке Python называется пакетом, поэтому такая операция импортирования называется импортированием пакетов. В действительности, операция импортирования пакета превращает имя каталога в еще одну разновидность пространства имен, в котором атрибутам соответствуют подкаталоги и файлы модулей, находящиеся в этих каталогах.

Если вы решили использовать импортирование пакетов, существует еще одно условие: каждый каталог в пути, указанном в инструкции импортирования пакета, должен содержать файл с именем __init__.py, в противном случае операция импорта пакета будет терпеть неудачу.

Для такой структуры каталогов:
dir0\dir1\dir2\mod.py

и инструкции импортирования, имеющей следующий вид:
import dir1.dir2.mod

применяются следующие правила:
- dir1 и dir2 должны содержать файл __init__.py.
- dir0, каталог-контейнер, может не содержать файл __init__.py – этот файл будет проигнорирован, если он присутствует.
- dir0, но не dir0\dir1, должен присутствовать в пути поиска модулей (то есть он должен быть домашним каталогом или присутствовать в переменной окружения PYTHONPATH и так далее).

Таким образом, структура каталогов в этом примере должна иметь следующий вид:

```
dir0\ # Каталог-контейнер в пути поиска модулей
   dir1\
      __init__.py
      dir2\
         __init__.py
         mod.py
```

Файлы __init__.py могут содержать программный код на языке Python, как любые другие файлы модулей. Отчасти они являются объявлениями для интерпретатора и могут вообще ничего не содержать.

Когда интерпретатор Python импортирует каталог в первый раз, он автоматически запускает программный код файла __init__.py этого каталога. По этой причине обычно в эти файлы помещается программный код, выполняющий действия по инициализации, необходимые для файлов в пакете. Например, этот файл инициализации в пакете может использоваться для создания файлов с данными, открытия соединения с базой данных и так далее. Обычно файлы __init__.py не предназначены для непосредственного выполнения – они запускаются автоматически, когда выполняется первое обращение к пакету.

Классы

В объектно-ориентированной модели языка Python существует две разновидности объектов: объекты классов и объекты экземпляров. Объекты классов реализуют поведение по умолчанию и играют роль фабрик по производству объектов экземпляров.

То есть, в действительности, классы – это фабрики, способные производить множество экземпляров объектов.

Классы — фабрики объектов

При выполнении инструкция class создается объект класса и ему присваивается имя, указанное в заголовке инструкции. Как и инструкции def, инструкции class обычно выполняются при первом импортировании содержащих их файлов. Согласно общепринятым соглашениям, имена классов в языке Python должны начинаться с заглавной буквы, чтобы обеспечить визуальное отличие. Таким образом, в именах классов нет ничего необычного, это просто переменные, которым присваиваются объекты-классы.

```
class <name>(superclass,...): # Присваивание имени
    data = value              # Данные класса
    def method(self,...):     # Методы
        self.member = value   # Данные экземпляров
```

Операции присваивания внутри инструкции class создают атрибуты класса. После выполнения инструкции class атрибуты класса становятся доступны по их составным (полным) именам: object.name.

Инструкции def, вложенные в инструкцию class, создают методы класса.

Создание объктов — экземпляров класса

Всякий раз, когда вызывается класс, создается и возвращается новый объект экземпляра. Экземпляры представляют собой конкретные элементы данных в вашей программе.

Каждый объект экземпляра наследует атрибуты класса и приобретает свое собственное пространство имен. Объекты экземпляров создаются из классов и представляют собой новые пространства имен.

Операции присваивания значений атрибутам через ссылку self в методах создают атрибуты в каждом отдельном экземпляре. Методы класса

получают в первом аргументе (с именем self в соответствии с соглашениями) ссылку на обрабатываемый объект экземпляра – *присваивание атрибутам через ссылку self создает или изменяет данные экземпляра, а не класса.*

На самом деле, изначально создаваетмые объекты экземпляры класса пустые, но они связаны с классом, из которого были созданы. Если через имя объекта экземпляра обратиться к атрибуту объекта класса, то в результате поиска по дереву наследования интерпретатор вернет значение атрибута класса (при условии, что в экземпляре отсутствует одноименный атрибут).

Таким образом, механизм наследования привлекается лишь в момент разрешения имени атрибута, и вся его работа заключается лишь в поиске имен в связанных объектах. В этом заключается суть наследования в языке Python.

Наследование

Объекты экземпляров, созданные из класса, наследуют атрибуты класса. В языке Python классы также могут наследовать другие классы, что дает возможность создавать иерархию классов, поведение которых специализируется за счет переопределения более обобщенных атрибутов, находящихся выше в дереве иерархии, в подклассах, находящихся ниже в иерархии. *В результате, чем ниже мы опускаемся в дереве иерархии, тем более специализированными становятся классы.*

В языке Python экземпляры наследуют классы, а классы наследуют супер-классы.

Суперклассы перечисляются в круглых скобках в заголовке инструкции class. Чтобы унаследовать атрибуты другого класса, достаточно указать этот класс в круглых скобках в заголовке инструкции class. Наследующий класс называется подклассом, а наследуемый класс называется его супер-классом.

Классы наследуют атрибуты своих суперклассов. Как экземпляры наследуют имена атрибутов, определяемых их классами, так же и классы наследуют все имена атрибутов, определяемые в их суперклассах, – интерпретатор автоматически отыскивает их, когда к ним выполняется обращение, если эти атрибуты отсутствуют в подклассах.
Экземпляры наследуют атрибуты всех доступных классов. Каждый экземпляр наследует имена из своего класса, а также из всех его суперклассов. Во время поиска имен интерпретатор проверяет сначала экземпляр, потом его класс, а потом все суперклассы.

Каждое обращение object.attribute вызывает новый независимый поиск. Интерпретатор выполняет отдельную процедуру поиска в дереве классов для каждого атрибута, который ему встречается в выражении запроса. Сюда входят ссылки на экземпляры и классы из инструкции class (например, X.attr), а также

ссылки на атрибуты аргумента экземпляра self в методах класса. Каждое выражение self.attr в методе вызывает поиск attr в self и выше.

Изменения в подклассах не затрагивают суперклассы. Замещение имен суперкласса в подклассах ниже в иерархии (в дереве классов) изменяет подклассы и тем самым изменяет унаследованное поведение.

Метод-конструктор __init__

Метод **__init__** вызывается, когда создается новый объект экземпляра.

Он позволяет классам заполнять атрибуты вновь созданных экземпляров. Конструктор полезно использовать практически во всех разновидностях ваших классов.

```
class NewObject(SuperObject):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __str__(self):
        return ('[Объект класса NewObject: a=%s, b=%s, c=%s]'
                % (self.a, self.b, self.c))
```

Метод **__str__** вызывается при выводе объекта (точнее, когда он преобразуется в строку для вывода вызовом встроенной функции str или ее эквивалентом внутри интерпретатора).

Инкапсуляция

Идея инкапсуляции в Python заключается в том, чтобы **спрятать логику операций за интерфейсами** и тем самым добиться, чтобы каждая операция имела единственную реализацию в программе. Благодаря такому подходу, если в дальнейшем потребуется вносить какие-либо изменения, модифицировать программный код придется только в одном месте.

Кроме того, мы сможем изменять внутреннюю реализацию операции практически как угодно, не рискуя нарушить работоспособность программного кода, использующего ее.

В терминах языка Python это означает, что мы должны реализовать операции над объектами в виде методов класса, а не разбрасывать их по всей программе. Фактически возможность сосредоточить программный код в одном месте,

устранить избыточность и тем самым упростить его сопровождение является одной из самых сильных сторон классов.

Вызов методов предков

```
class Person:
    def raise_pay(self, percent):
        self.pay = int(self.pay * (1 + percent/100))

class Manager(Person):
    def raise_pay(self, percent, bonus=10):
        Person.raise_pay(self, percent+bonus)
```

Напомним, что вызов метода объекта:

```
instance.method(args...)
```

автоматически транслируется интерпретатором в эквивалентную форму:

```
class.method(instance, args...)
```

Вызов конструктора суперкласса

Если в вашей программе вы переопределяете метод __init__ вашего класса, часто бывает нужно вызвать __init__ суперкласса. Это делается точно также, как и с любым другим методом:

```
class Super:
    def __init__(self, x):
        ...программный код по умолчанию...

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)  # Вызов __init__ суперкласса
        ...адаптированный код... # дополнительные действия

I = Sub(1, 2)
```

Преимущества ООП

При объектно-ориентированном подходе к созданию программ мы адаптируем имеющийся программный код, а не копируем и не изменяем его. Это преимущество не всегда очевидно для начинающих программистов, особенно на

фоне дополнительных требований, предъявляемых при создании классов. Но в целом применение объектно-ориентированного стиля программирования способно существенно сократить время разработки, по сравнению с другими подходами.

Внутренние методы классов

Чтобы минимизировать вероятность конфликта имен, программисты часто добавляют **символ подчеркивания** в начало имени метода, не предназначенного для использования за пределами класса. Это общепринятое соглашение об именовании внутренних методов классов в языке Python.

Также возможно использование псевдочастных атрибутов класса. Они обозначаются при помощи **двойного символа подчеркивания**. Интерпретатор автоматически дополняет такие имена, включая в них имя вмещающего класса, что обеспечивает им уникальность.

```
>>> class Klass:
        _a = 10
        __b = 100

>>> dir(BB)
['_BB__b', '__class__', ...,  '_a']
```