Питон

Курс лекций

Лекция девятая

© Зайцев Вадим Валерьевич, 2002–2010,
spider.vz@gmail.com

Сегодня мы наконец перейдём ко второй части курса. Если в первой части вы получили некоторые знания о языках программирования, о методах работы в питоне, о его стиле, об основных типах данных и управляющих конструкциях, то во второй вы ознакомитесь с современными принципами работы программиста и проектирования больших программных комплексов. Эти принципы, конечно, пригодны, и для малых программ, но в этом случае не настолько эффективны.

Когда компьютеры были большими, программы были маленькими, писались на перфокартах и на было необходимости в каких-то серьёзных научных методах их проектирования, потому что всё равно разобраться в сколь-нибудь крупной программе (а такие всё же писались) без программиста или команды программистов, её создавших, было невозможно. Сейчас быстро развиваются как вычислительные мощности компьютеров, так и технология хранения информации и объёмы носителей. Это даёт возможность писать настолько большие программы, что в их создании задействованы тысячи людей и понадобилась технология, обеспечивающая, во-первых, взаимодействие этих людей, озабоченных каждый только решением своей маленькой задачи, а во-вторых, безболезненную замену одного программиста другим без переработки всего проекта.

Сначала для этого использовался структурный подход и теория спецификаций и абстракций. При этом большая задача решалась абстрактно, и её решение выражалось через решения ряда более простых задач. Это называется алгоритмическая декомпозиция, то есть разделение алгоритма на части. Если для решения какой-то мелкой подзадачи писалась процедура, к ней прилагалась так называемая спецификация, объясняющая её смысл и формат запуска так, чтобы обращение к ней могло происходить без знания внутренней структуры этой процедуры. Этот метод работал хорошо до тех пор, пока не были предприняты первые попытки создания систем, где число таких подпрограмм исчислялось тысячами. Никакие спецификации не помогут даже самому одарённому программисту запомнить тысячи

подпрограмм. Кроме того, в этом случае нарушается второе требование — безболезненно заменить опытного программиста нельзя, потому что новый, пришедший на его место, не будет столь же компетентен, пока не запомнит первую пару сотен названий процедур.

На смену структурному проектированию пришла...

5 Объектно-ориентированная технология

5.1 Объектная модель и связанные с ней термины

Объектно-ориентированная технология основывается на так называемой объектной модели. Основными её принципами (мы рассмотрим их подробно) являются: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость. Каждый из этих принципов сам по себе не нов, но в объектной модели они впервые применены в совокупности.

Объектно-ориентированный анализ и проектирование принципиально отличаются от традиционных подходов структурного проектирования: здесь нужно по-другому представлять себе процесс декомпозиции, а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для структурного программирования. Отличия обусловлены тем, что структурное проектирование основано на структурном программировании, тогда как в основе объектно-ориентированного проектирования лежит методология объектно-ориентированного программирования. К сожалению, для разных людей термин «объектно-ориентированное программирование» означает разное. Ренч, один из видных теоретиков, верно предсказал: *объектно-ориентированное программирование будет занимать такое же место, какое занимало структурное программирование. Оно всем будет нравиться. Каждая фирма будет рекламировать свой продукт как созданный по этой технологии. Все программисты будут писать в этом стиле, причём все по-разному. Все менеджеры будут рассуждать о нём. И никто не будет знать, что же это такое.*

Речь шла о восьмидесятых годах прошлого столетия, хотя предсказание продолжает сбываться и в наши дни. Мы попытаемся внести хоть небольшую ясность в осознание этого термина нашими слушателями (читателями).

Объектом называется некая сущность, которой можно оперировать. На самом деле такого определения уже достаточно, дальнейшая конкретизация может только погубить всё дело. В разных программах объектом может быть источник звука, устройство ввода/вывода, буква, книга, самолёт, планета, дыхание или полёт стрелы. Объект, как мы увидим позже, имеет состояние, поведение и идентичность. Состояние описывает текущее положение объекта, поведение — то, как он будет реагировать на различные ситуации, идентичность отличает объект от других.

Класс — это множество очень схожих объектов. Англичанин, наверное, объяснил бы различие между понятиями *класс* и *объект* через определённые и неопределённые артикли своего родного языка. Лишённые такой возможности, мы поясним на примерах: кошка — это класс, а трёхцветная кошка по имени Мурка, лежащая на подоконнике — объект. Карта PCI — класс, а внутренний модем Acorp/PCI/56k/V90 или звуковая карта SBLive! — разные объекты этого класса. Литературное произведение — класс, «Слово о полку Игореве» — объект. Вообще, что называть классом, а что — объектом, определяется программистом на этапе проектирования системы. Кроме того, класс можно понимать как `тип объекта`.

Объектно-ориентированное программирование, или object-oriented programming (OOP), определяется его следующим образом:

Объектно-ориентированное программирование — это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённом класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части:

- OOP использует в качестве базовых элементов объекты, а не алгоритмы
- Каждый объект является экземпляром какого-либо определённого класса
- Классы организованы иерархически

Конечно, программа будет объектно-ориентированной только при соблюдении всех трёх указанных требований. В частности, программирование, не основанное на иерархических отношениях, называется программированием на основе *абстрактных типов данных*.

Объектно-ориентированное проектирование, или object-oriented design (OOD) — термин хоть и близкий к OOP, но не идентичный ему. Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование же, напротив, основное внимание уделяет правильному и эффективному структурированию сложных систем. Мы определяем объектно-ориентированное проектирование следующим образом:

Объектно-ориентированное проектирование — это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приёмы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части: объектно-ориентированное проектирование

- основывается на объектно-ориентированной декомпозиции;
- использует многообразие приёмов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и про-

цессы) структуру системы, а также её статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором — алгоритмами. Иногда мы будем использовать аббревиатуру OOD для обозначения метода объектно-ориентированного проектирования, потому что иначе при использовании русских аббревиатур мы быстро перепутаем программирование с проектированием.

Объектно-ориентированный анализ, или object-oriented analysis (OOA), направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ — это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

А как же соотносятся между собой OOP, OOD и OOA? На результатах, полученных OOA, формируются модели, на которых основывается OOD, которое, в свою очередь, создаёт фундамент для окончательной реализации системы с использованием методологии OOP.

5.2 Объекты и классы в питоне

В соответствии с определением OOP, не все языки программирования являются объектно-ориентированными. Страуструп, автор C++, одного из первых объекто-ориентированных языков, считал, что объектно-ориентированный язык — это язык, имеющий средства хорошей поддержки объектно-ориентированного стиля программирования... Обеспечение такого стиля в свою очередь означает, что в языке удобно пользоваться этим стилем. Если написание программ в стиле OOP требует специальных усилий или невозможно, то этот язык не отвечает требованиям OOP. А отвечает ли требованиям объектно-ориентированного языка питон, выбранный нами? Это легко проверить. Попробуем создать какой-нибудь класс и его объект. Если мы этого сделать не сможем или нам будет очень трудно, придётся сложить оружие и признать, что питон не объектно-ориентирован.

Напишем класс `Dog`, описывающий собаку, которая может лаять. Для этого внутри описания класса (то есть после отступа) зададим новую функцию `hawl()`, от которой в нашем простом примере много не требуется. Затем создадим объект, то есть совершенно определённую собаку по имени Шарик. Попросим её полаять.

```
class Dog:
 def hawl(self):
  print "Гав!"
Sharik=Dog('Шарик')
Sharik.hawl()
```

Она действительно лает (можете проверить). Итак, всё было сделано быстро и заключилось в пяти строках кода. Вывод: питон — объектно-ориентированный язык!

Подведём итоги правил задания классов и объектов в питоне: класс задаётся ключевым словом `class` с указанием имени класса. При этом создаётся новая область действия имён переменных — можно создавать внутренние переменные класса и внутренние подпрограммы (они называются *методами*). Для создания объекта используется оператор присваивания, в правой части которого стоит имя класса. (Таким образом, каждый объект должен принадлежать какому-то классу). Обращение к методам объекта происходит через имя объекта и имя метода, разделённые точкой.

Есть только одно отличие метода от обычной процедуры или функции: первым параметром он обязан брать сам объект, частью которого является. При вызове метода с именем объекта оно подставляется автоматически, мы ведь написали `Sharik.hawl()`, а не `Sharik.hawl(Sharik)`. А вот при вызове метода с именем класса (и такое можно в питоне!) объект нужно передавать в явном виде: `Dog.hawl(Sharik)`. Таким образом, нельзя пользоваться методами класса, не создав ни одного экземпляра.

Следуя традиции, мы называем первый параметр `self` (сам). Это не более чем соглашение, и любой программист может в целях экономии места или по любой другой причине переименовать его.

В некоторых классах есть необходимость произведения некоторых действий при создании и/или при уничтожении объекта. Для этого используются *конструкторы* и *деструкторы*. Хорошим стилем объектно-ориентированного программирования считается записывание *всего*, что нужно выполнить при создании объекта, в конструктор, а всего, что нужно при уничтожении — в деструктор. Записываются они следующим образом:

```
class Dog:
 def __init__(self,newname='Бобик'):
  self.name=newname
 def __del__(self):
  print self.name,':',
  self.hawl()
 def hawl(self):
  print "Гав!"

Sharik=Dog('Шарик')
Bobik=Dog();
```

Конструктор — это та подпрограмма, которая вызывается при создании экземпляра класса со всеми параметрами, написанными между скобками после имени класса. Как видно из примера, никто не запрещает использовать в написании конструктора особых приёмов работы с подпрограммами, в частности, параметров со значениями по умолчанию. Вообще, конструктор — такой же метод класса, как и любой другой (в качестве лю-

бого другого можно рассмотреть `hawl`), только имеющий особое имя, распознаваемое интерпретатором как служебное. Подробнее о именах свойств и методов мы поговорим позже.

Деструктор вызывается неявно при удалении объекта (то есть, в том случае, когда мощность объекта станет нулевой), поэтому не может иметь никаких параметров (опять же, кроме самого объекта). Деструктор вызывается **до** действительного удаления, так что все данные ещё живы и могут быть спасены от уничтожения путём копирования в безопасное место.