Питон
Курс лекций
Лекция восьмая

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

4.5 Лямбда-исчисление

В первой половине XX века американский математик Алонзо Чёрч предложил использовать для описания частично рекурсивных функций достаточно простой формализм, названный им лямбда-исчислением. Он же сформулировал так называемый **тезис Чёрча** (на котором базируются тезисы Тьюринга и Маркова) о том, что любая функция, вычислимая в интуитивном смысле эквивалентна некоей частично рекурсивной функции. Этот тезис содержит в себе нестрогое определение, поэтому, с одной стороны, не может быть доказан, а с другой, позволяет упростить некоторые теоретические выкладки. Частично рекурсивные функции суть функции, которые могут зависеть от собственного значения при других входных данных и могут быть определены не для всех входных данных.

Впервые лямбда-выражения появились в языке Лисп в конце 1950-х годов. Позаимствовав термин у Чёрча, его создатели, безусловно, внесли множество изменений. Позже было создано несколько языков чисто функционального типа, как на базе Лиспа (Схема, Лупс, КЛОС, Миранда, Хаскелл), так и сильно отличающихся от него (ФП, МЛ, Хоуп, Эрланг). Функциональное программирование — это отдельная парадигма программирования, где программист задаёт зависимость функций друг от друга, определяя таким образом их свойства и значения. В языках, наиболее точно соответствующих этой концепции, нет переменных, которые могут влиять на контекстную независимость, как мы видели на прошлой лекции. Элементы функционального программирования есть и в питоне.

Лямбда-функция в питоне — это функция без имени, о которой известно только количество аргументов и формула для вычисления итогового значения, причём формула должна записываться единым выражением. Вот пример лямбда-функции, складывающей три числа:

```
lambda x,y,z:x+y+z
```

Проще и не придумать. Понятно, что описывать огромную функцию, вызывающуюся много раз, лямбда-функцией, по меньше мере неразумно,

но в некоторых случаях (которые мы рассмотрим на этой лекции) лямбда-функции бывают полезны. Во-первых, их можно присваивать:

```
R=lambda x,y:pow(x*x+y*y,0.5)
print R(3,4)
```

На печать будет выдано `5.0`. Сравните ту же самую функцию, объявленную стандартным питоновским способом:

```
def R(x,y):
 return pow(x*x+y*y,0.5)
```

Длиннее, многословнее и, что более важно, менее очевидно. Когда же мы перейдём к применению лямбда-функций в приёмах функционального программирования, экономия места будет ещё больше.

Второе применение лямбда-функций следует из того, что определение обычной функции не может быть сгенерировано программой «на лету», а определение лямбда-функции — может, и очень просто:

```
def genincr(n):
 return lambda x,i=n:x+i
```

Эта функция возвратит функцию-инкрементатор, увеличивающую свой аргумент на `n`, где `n` даётся при создании функции. Для любителей экзотики сразу отвечаем утвердительно на возникший у них вопрос. Да, так тоже можно:

```
genincr=lambda n:lambda x,i=n:x+i
```

Это определение функции `genincr` полностью эквивалентно предыдущему.

Упражнение. Почему нельзя было задать возвращаемую функцию-инкрементатор как `lambda x:x+n`? (Возможно, если вы затрудняетесь ответить на этот вопрос, вам следует повторить материал прошлой лекции).

4.6 Элементы функционального программирования

Кроме всех этих очевидных применений лямбда-функций, существуют ещё четыре стандартных приёма:

1. **Вызов функций — `apply()`.**

   Выполнять функции можно, как мы знаем, пользуясь скобками: `func()`, где `func` — имя функции. Но в некоторых случаях бывает удобно сначала последовательно подготовить все аргументы, и только потом вызвать функцию. Функция `apply` принимает два или три аргумента (третий необязателен). Первый — функция: либо имя переменной, содержащей функцию, либо определение лямбда-функции. Второй — кортеж (или другая последовательность, которая внутри функции `apply` всё равно преобразуется в кортеж) с параметрами. Третий аргумент — словарь со всеми именованными параметрами. Так,

   ```
   A=1,2,[3,4],[5,6],7
   ```

   ```
   B=2,3
   ```

```
apply(lambda x,y,z:x[y].append(z),(A,2,B))
```

аналогично следующему:

```
A=1,2,[3,4],[5,6],7
B=2,3
def func(x,y,z):
 x[y].append(z)
func(A,2,B)
```

2. **Отображение списков — map().**

   Функция `map()` порождает новый список из значений функции, применённой к каждому элементу первоначального списка. Легко догадаться, что эта функция берёт не менее двух аргументов: функции для применения и последовательности (списка или кортежа) её параметров. В этом случае наша функция должна брать только один параметр. Можно использовать и многопараметрические функции, но в этом случае нужно давать столько списков или кортежей, сколько у неё параметров. Конечно же, они должны быть одинаковой длины. Вне зависимости от типов последовательностей, данных функции `map`, она вернёт список.

   Функция `map` является как бы обобщением предыдущей функции, `apply`, последовательно применяя данную функцию к элементам последовательности. Можно, например, вычислить значения синусов чисел от 1 до 10:

   ```
   from math import sin
   map(sin,range(1,10))
   ```

   Если же нужен не синус, а, скажем, возведение в третью степень, нам поможет лямбда-функция:

   ```
   map(lambda x:x*x*x,range(1,10))
   ```

   В употреблении функции `map` существует ещё одна хитрость: можно вместо функции подставить `None`, тогда будет использована функция по умолчанию — т.н. функция идентичности, возвращающая свои аргументы. Догадливые программисты могут использовать этот факт для краткой записи транспонирования матрицы. С использованием всех полученных нами знаний можно определить функцию транспонирования матрицы произвольного размера следующим образом:

   ```
   trans=lambda X:map(list,apply(map,[None]+X))
   ```

   Матрица должна быть представлена в виде списка списков. Это удобное представление не только для работы с элементами, но и для функции `apply`. Не хватает только первого (точнее, нулевого) элемента — имени функции, что мы и добавляем. Затем выполняется `apply`, а точнее, `map`, собственно транспонирующий наш список списков в список

кортежей, что исправляется (нехорошо изменять структуру представления при транспонировании) применением функции `list` к каждому элементу списка (к каждому кортежу). Запусками нашей функции с различными параметрами можно убедиться, что она работает как для квадратных, так и для прямоугольных матриц.

3. **Фильтрация списков — filter()**.

   Функция `filter()` генерирует новый список из тех элементов исходного списка, для которых проверочная функция истинна. Сами значения элементов при этом не изменяются. Первым аргументом даётся проверочная функция, а вторым следует список (или кортеж, или строка, ...). Если функция — `None`, то аналогично уже описанному используется функция идентичности, то есть из последовательности выбрасываются все нулевые или пустые элементы.

   Например, список нечётных чисел от 2 до 15 можно получить так:

   ```
   filter(lambda x:x%2,range(2,16)
   ```

   Как видно, лямбда-функции хорошо себя рекомендуют и тут. И только врождённая честность не позволяет нам утаить тот факт, что список нечётных чисел можно получить и проще, пользуясь третьим, необязательным параметром функции `range()`.

4. **Цепочечные вычисления — reduce()**.

   Функция `reduce()` производит цепочечные вычисления, многократно применяя данную функцию к каждому элементу, подставляя аккумулятор в качестве первого параметра, а сам элемент — в качестве второго. При этом она берёт от двух до трёх аргументов: функцию для вычисления и последовательность как обязательные и стартовое значение аккумулятора как необязательный. Например, факториал числа можно считать так:

   ```
   fact=lambda n:reduce(lambda a,N:a*N,range(1,n+1),1L)
   ```

   Стартовое значение в `1L` необходимо для того, чтобы результат был длинным целым, иначе нельзя будет посчитать даже факториал 10.

   Цепочечные вычисления идут слева направо. Например, при выполнении `lambda x,y:x+y,range(1,5)` порядок выполнения будет таков: `(((1+2)+3)+4)`.

4.7 Поиск простых чисел

Пришла пора нам попробовать применить полученные знания о функциональном программировании. Итак, не боясь длинных выражений, будем помнить все приёмы. Попробуем написать лямбда-функцию, которая будет находить все простые числа, не превосходящие данного. Что такое простое число? Согласно определению, простое число не делится без остатка ни на какое другое число. Понятно, что нет смысла проверять делимость на

числа, превышающие исходное. Напишем сначала функцию, составляющую список всех остатков от деления данного числа на другие:

```
Z=lambda n:map(lambda a,b=n:b%a,range(2,n))
```

Если в этом списке есть хотя бы один ноль, это означает, что число `n` делится без остатка на какое-то другое число, то есть, что `n` — не простое. Построим функцию, возвращающую 1, если в данном ей списке нет нулей и 0 в противном случае:

```
Y=lambda l:reduce(lambda c,d:c*d!=0,l,1)
```

Теперь `Y(Z(n))` возвращает 1, если `n` — простое и 0 в противном случае. Половина дела уже сделана. Осталось реализовать перебор всех чисел из какого-то промежутка, то есть построить отображение (map) списка последовательных чисел в список нулей и единиц. В таком случае единицы будут стоять на местах, обозначающих номер простого числа. Будет лучше, если вместо единиц мы будем выдавать само число, тогда, удалив все нули и списка, мы получим список простых чисел без лишних усилий.

```
X=lambda m:map(lambda e:e*Y(Z(e)),range(2,m))
```

Ну, удалить нули для нас проще простого:

```
W=lambda k:filter(None,X(k))
```

Если вспомнить арифметику, то станет понятно, что делитель числа не может превышать его корня. Поэтому из соображений оптимизации по скорости `Z` можем изменить следующим образом:

```
Z=lambda n:map(lambda a,b=n:b%a,range(2,1+pow(n,0.5)))
```

Всё, задание выполнено и, даже более того, выполнено оптимально. Если мы теперь подставим `Z` в `Y`, `Y` в `X`, а `X` в `W`, то сами ужаснёмся результату наших усилий:

```
V=lambda k:filter(None,map(lambda e:e*reduce(lambda
c,d:c*d!=0,map(lambda a,b=e:b%a,range(2,1+pow(e,0.5))),1),range(2,k)))
```

Для тех, кто любит создавать программы, которые невозможно прочесть кому-либо, кроме их создателя (да и ему это под силу только спустя день-два после написания), можно напомнить, что разные переменные из разных областей действия имён могут иметь одинаковые имена. Если это учесть (а у нас нигде не употребляется более двух имён параметров одновременно), то функция примет вид:

```
U=lambda x:filter(None,map(lambda x:x*reduce(lambda
x,y:x*y!=0,map(lambda y,x=x:x%y,range(2,1+pow(e,0.5))),1),range(2,x)))
```

Потрясающе! Пользуйтесь приёмами функционального программирования, и ваши программы будут мутны и нечитаемы! А вообще, рассмотренный нами пример есть курьёз, хоть и вполне жизнеспособный, как правило, элементы функционального программирования у программистов на питоне так далеко не идут. Но игнорировать этот аппарат нельзя — слишком велика выгода от его применения, как в визуальном представлении алгоритмов, так и в скорости выполнения программ.

4.8 Подпрограммы как средство поднятия уровня абстракции

Рассмотрев различные виды функций и их применения, мы можем подвести итог. Подпрограммы являются чрезвычайно полезным инструментом, без которых невозможна реализация сколь-нибудь крупных проектов. Подпрограммы позволяют скрывать от проектировщика подробности реализации тех или иных мелких подзадач и дают сконцентрироваться на их композиции и решении больших задач путём собирания их из готовых решений подзадач. Подпрограммы дают возможность смены терминологии, её укрупнения. Например, при реализации межпрограммного взаимодействия по сети одни подпрограммы будут решать задачу пересылки серии байт в порт компьютера, другие будут пересылать целые массивы сложных строковых данных с контролем целостности, пользуясь при этом первыми, третьи будут управлять работой удалённого компьютера и приложений на нём, посылая команды путём запуска других функций, четвёртые подпрограммы будут позволять запросто работать на одном компьютере, при этом оперируя окном приложения с другого, пользуясь третьими подпрограммами. Всё это пришлось бы делать одновременно, если бы не было этого аппарата.

На практике дело обстоит куда лучше, для многих элементарных задач существуют уже написанные решения, поэтому программисту-разработчику достаточно только приспособить их для своих нужд, то есть использовать их в своих подпрограммах. Это называется *повторным использованием кода* и применяется очень широко. И этого не было бы без аппарата создания подпрограмм.

Подпрограммы позволяют создавать некий алгоритм решения проблемы и называть его одним именем, пользуясь этим именем позже, при составлении более сложных алгоритмов, и так далее. Сегодня мы заканчиваем первую часть курса, посвящённую процедурному программированию. Следующая лекция и все следующие за ней будут рассказывать уже об объектном подходе, модели, позволяющей писать ещё б*о*льшие проекты, и делать это проще, чем поддерживание спецификации сотен тысяч мелких подпрограмм.