**Функциональное программирование на языке Python**

Автор: David Mertz, Ph.D., Applied Metaphysician, Gnosis Software, Inc.

Хотя пользователи обычно думают о Python как о процедурном и объектно-ориентированном языке, он содержит все необходимое для поддержки полностью функционального подхода к программированию.

В этой статье рассматриваются общие концепции функционального программирования и иллюстрируются способы реализации функционального подхода на Python.

**Что такое Python?**

Python - свободно распространяемый, очень высокоуровневый интерпретируемый язык, разработанный Гвидо ван Россумом (Guido van Rossum). Он сочетает прозрачный синтаксис с мощной (но необязательной) объектно-ориентированной семантикой. Python доступен почти на всех существующих ныне платформах и обладает очень высокой переносимостью между платформами.

Что такое функциональное программирование?

Лучше всего начать с труднейшего вопроса - а что, собственно, такое "функциональное программирование (FP)"? Один из возможных ответов - "это когда вы пишете на языке наподобие Lisp, Scheme, Haskell, ML, OCAML, Clean, Mercury или Erlang (или еще на некоторых других)". Этот ответ, безусловно, верен, но не сильно проясняет суть. К сожалению, получить четкое мнение о том, что же такое FP, оказывается очень трудно даже среди собственно функциональных программистов. Вспоминается притча о трех слепцах и слоне. Возможно также определить FP, противопоставив его "императивному программированию" (тому, что вы делаете на языках наподобие C, Pascal, C++, Java, Perl, Awk, TCL и на многих других - по крайнее мере, большей частью).

Хотя автор всеми силами приветствует советы со стороны тех, кто лучше него знает предмет, он мог бы приблизительно охарактеризовать функциональное программирование как обладающее как минимум несколькими из следующих свойств. В языках, называемых функциональными, хорошо поддерживаются нижеперечисленные подходы, а все прочие подходы поддерживаются плохо или не поддерживаются вовсе:

Функции - объекты первого класса. Т.е., все, что можно делать с "данными", можно делать и с функциями (вроде передачи функции другой функции в качестве параметра).

Использование рекурсии в качестве основной структуры контроля потока управления. В некоторых языках не существует иной конструкции цикла, кроме рекурсии.

Акцент на обработке списков (lists, отсюда название Lisp - LISt Processing). Списки с рекурсивным обходом подсписков часто используются в качестве замены циклов.

"Чистые" функциональные языки избегают побочных эффектов. Это исключает почти повсеместно распространенный в императивных языках подход, при котором одной и той же переменной последовательно присваиваются различные значения для отслеживания состояния программы.

FP не одобряет или совершенно запрещает утверждения (statements), используя вместо этого вычисление выражений (т.е. функций с аргументами). В предельном случае, одна программа есть одно выражение (плюс дополнительные определения).

FP акцентируется на том, что должно быть вычислено, а не как.

Большая часть FP использует функции "высокого порядка" (функции, оперирующие функциями, оперирующими функциями).

Защитники функционального программирования доказывают, что все эти характеристики приводят к более быстрой разработке более короткого и безошибочного кода. Более того, высокие теоретики от компьютерной науки, логики и математики находят, что процесс доказательства формальных свойств для функциональных языков и программ много проще, чем для императивных.

**Функциональные возможности, присущие Python**

Python поддерживает большую часть характеристик функционального программирования, начиная с версии Python 1.0. Но, как большинство возможностей Python, они присутствуют в очень смешанном языке. Так же как и с объектно-ориентированными возможностями Python, вы можете использовать то, что вам нужно, и игнорировать все остальное (пока оно вам не понадобится). В Python 2.0 было добавлено очень удачное "синтаксическое украшение" - списочные встраивания (list comprehensions). Хотя и не добавляя принципиально новых возможностей, списочные встраивания делают использование многих старых возможностей значительно приятнее.

Базовые элементы FP в Python - функции map(), reduce(), filter() и оператор lambda. В Python 1.x введена также функция apply(), удобная для прямого применения функции к списку, возвращаемому другой. Python 2.0 предоставляет для этого улучшенный синтаксис. Несколько неожиданно, но этих функций и всего нескольких базовых операторов почти достаточно для написания любой программы на Python; в частности, все управляющие утверждения ('if', 'elif', 'else', 'assert', 'try', 'except', 'finally', 'for', 'break', 'continue', 'while', 'def') можно представить в функциональном стиле, используя исключительно функции и операторы. Несмотря на то, что задача реального удаления всех команд управления потоком, возможно, полезна только для представления на конкурс "невразумительный Python" (с кодом, выглядящим как программа на Lisp'е), стоит уяснить, как FP выражает управляющие структуры через вызовы функций и рекурсию.

**Исключение команд управления потоком**

Первое, о чем стоит вспомнить в нашем упражнении - то, что Python "замыкает накоротко" вычисление логических выражений.1 Оказывается, это предоставляет эквивалент блока 'if'/'elif'/'else' в виде выражения. Итак:

#------ "Короткозамкнутые" условные вызовы в Python -----#

# Обычные управляющие конструкции

if <cond1>: func1()

elif <cond2>: func2()

else: func3()

# Эквивалентное "накоротко замкнутое" выражение

(<cond1> and func1()) or (<cond2> and func2()) or (func3())

# Пример "накоротко замкнутого" выражения

>>> x = 3

>>> def pr(s): return s

>>> (x==1 and pr('one')) or (x==2 and pr('two')) or (pr('other'))

'other'

>>> x = 2

>>> (x==1 and pr('one')) or (x==2 and pr('two')) or (pr('other'))

'two'

Казалось бы, наша версия условных вызовов с помощью выражений - не более, чем салонный фокус; однако все становится гораздо интересней, если учесть, что оператор lambda может содержать только выражения! Раз, как мы только что показали, выражения могут содержать условные блоки, используя короткое замыкание, выражение lambda позволяет в общей форме представить условные возвращаемые значения. Базируясь на предыдущем примере:

#--------- Lambda с короткозамкнутыми условными выражениями в Python -------#

>>> pr = lambda s:s

>>> namenum = lambda x: (x==1 and pr("one")) \

... or (x==2 and pr("two")) \

... or (pr("other"))

>>> namenum(1)

'one'

>>> namenum(2)

'two'

>>> namenum(3)

'other'

**Функции как объекты первого класса**

Приведенные примеры уже засвидетельствовали, хотя и неочевидным образом, статус функций как объектов первого класса в Python. Дело в том, что, создав объект функции оператором lambda, мы произвели чрезвычайно общее действие. Мы имели возможность привязать наш объект к именам pr и namenum в точности тем же способом, как могли бы привязать к этим именам число 23 или строку "spam". Но точно так же, как число 23 можно использовать, не привязывая ни к какому имени (например, как аргумент функции), мы можем использовать объект функции, созданный lambda, не привязывая ни к какому имени. Функция в Python - всего лишь еще одно значение, с которым можно что-то сделать.

Главное, что мы делаем с нашими объектами первого класса - передаем их во встроенные функции map(), reduce() и filter(). Каждая из этих функций принимает объект функции в качестве первого аргумента. map() применяет переданную функцию к каждому элементу в переданном списке (списках) и возвращает список результатов. reduce() применяет переданную функцию к каждому значению в списке и ко внутреннему накопителю результата; например, reduce(lambda n,m:n\*m, range(1,10)) означает 10! (факториал 10 - умножить каждый элемент на результат предыдущего умножения). filter() применяет переданную функцию к каждому элементу списка и возвращает список тех элементов исходного списка, для которых переданная функция вернула значение истинности. Мы также часто передаем функциональные объекты нашим собственным функциям, но чаще некоторым комбинациям вышеупомянутых встроенных функций.

Комбинируя три этих встроенных FP-функции, можно реализовать неожиданно широкий диапазон операций потока управления, не прибегая к утверждениям (statements), а используя лишь выражения.

**Функциональные циклы в Python**

Замена циклов на выражения так же проста, как и замена условных блоков. 'for' может быть впрямую переведено в map(). Так же, как и с условным выполнением, нам понадобится упростить блок утверждений до одного вызова функции (мы близки к тому, чтобы научиться делать это в общем случае):

#---------- Функциональный цикл 'for' в Python ----------#

for e in lst: func(e) # цикл на утверждении 'for'

map(func,lst) # цикл, основанный на map()

Кстати, похожая техника применяется для реализации последовательного выполнения программы, используя функциональный подход. Т.е., императивное программирование по большей части состоит из утверждений, требующих "сделать это, затем сделать то, затем сделать что-то еще". 'map()' позволяет это выразить так:

#----- Функциональное последовательное выполнение в Python ----------#

# создадим вспомогательную функцию вызова функции

do\_it = lambda f: f()

# Пусть f1, f2, f3 (etc) - функции, выполняющие полезные действия

map(do\_it, [f1,f2,f3]) # последовательное выполнение, реализованное на map()

В общем случае, вся главная программа может быть вызовом 'map()' со списком функций, которые надо последовательно вызвать, чтобы выполнить программу. Еще одно удобное свойство функций как объектов - то, что вы можете поместить их в список.

Перевести 'while' впрямую немного сложнее, но вполне получается :

#-------- Функциональный цикл 'while' в Python ----------#

# Обычный (основаный на утверждении 'while') цикл

while <cond>:

<pre-suite>

if <break\_condition>:

break

else:

<suite>

# Рекурсивный цикл в функциональном стиле

def while\_block():

<pre-suite>

if <break\_condition>:

return 1

else:

<suite>

return 0

while\_FP = lambda: (<cond> and while\_block()) or while\_FP()

while\_FP()

Наш вариант 'while' все еще требует функцию while\_block(), которая сама по себе может содержать не только выражения, но и утверждения (statements). Но мы могли бы продолжить дальнейшее исключение утверждений в этой функции (как, например, замену блока 'if/else' в вышеописанном шаблоне на короткозамкнутое выражение). К тому же, обычная проверка на месте <cond> (наподобие 'while myvar==7') вряд ли окажется полезной, поскольку тело цикла (в представленном виде) не может изменить какие-либо переменные (хотя глобальные переменные могут быть изменены в while\_block()). Один из способов применить более полезное условие - заставить while\_block() возвращать более осмысленное значение и сравнивать его с условием завершения. Стоит взглянуть на реальный пример исключения утверждений:

#---------- Функциональный цикл 'echo' в Python ------------#

# Императивная версия "echo()"

def echo\_IMP():

while 1:

x = raw\_input("IMP -- ")

if x == 'quit':

break

else

print x

echo\_IMP()

# Служебная функция, реализующая "тождество с побочным эффектом"

def monadic\_print(x):

print x

return x

# FP версия "echo()"

echo\_FP = lambda: monadic\_print(raw\_input("FP -- "))=='quit' or echo\_FP()

echo\_FP()

Мы достигли того, что выразили небольшую программу, включающую ввод/вывод, циклы и условия в виде чистого выражения с рекурсией (фактически - в виде функционального объекта, который при необходимости может быть передан куда угодно). Мы все еще используем служебную функцию monadic\_print(), но эта функция совершенно общая и может использоваться в любых функциональных выражениях , которые мы создадим позже (это однократные затраты).2 3 Заметьте, что любое выражение, содержащее monadic\_print(x) вычисляется так же, как если бы оно содержало просто x. В FP (в частности, в Haskell) есть понятие "монады" для функции, которая "не делает ничего, и вызывает побочный эффект при выполнении".

**Исключение побочных эффектов**

После всей проделанной работы по избавлению от совершенно осмысленных конструкций и замене их на невразумительные вложенные выражения, возникает естественный вопрос - "Зачем?!". Перечитывая мои описания характеристик FP, мы можем видеть, что все они достигнуты в Python. Но важнейшая (и, скорее всего, в наибольшей степени реально используемая) характеристика - исключение побочных эффектов или, по крайней мере, ограничение их применения специальными областями наподобие монад. Огромный процент программных ошибок и главная проблема, требующая применения отладчиков, случается из-за того, что переменные получают неверные значения в процессе выполнения программы. Функциональное программирование обходит эту проблему, просто вовсе не присваивая значения переменным.

Взглянем на совершенно обычный участок императивного кода. Его цель - распечатать список пар чисел, чье произведение больше 25. Числа, составляющие пары, сами берутся из двух других списков. Все это весьма напоминает то, что программисты реально делают во многих участках своих программ. Императивный подход к этой задаче мог бы выглядеть так:

#--- Императивный код для "печати произведений" ----#

# Процедурный стиль - поиск больших произведений с помощью вложенных циклов

xs = (1,2,3,4)

ys = (10,15,3,22)

bigmuls = []

#...прочий код...

for x in xs:

for y in ys:

#...прочий код...

if x\*y > 25:

bigmuls.append((x,y))

#...прочий код...

#...прочий код...

print bigmuls

Этот проект слишком мал для того, чтобы что-нибудь пошло не так. Но, возможно, он встроен в код, предназначенный для достижения множества других целей в то же самое время. Секции, комментированные как "#...прочий код..." - места, где побочные эффекты с наибольшей вероятностью могут привести к ошибкам. В любой из этих точек переменные xs, ys, bigmuls, x, y могут приобрести неожиданные значения в гипотетическом коде. Далее, после завершения этого куска кода все переменные могут иметь значения, которые могут ожидаются, а могут и не ожидаться посдедующим кодом. Очевидно, что инкапсуляция в функциях/объектах и тщательное управление областью видимости могут использоваться, чтобы защититься от этого рода проблем. Вы также можете всегда удалять ('del') ваши переменные после использования. Но, на практике, указанный тип ошибок весьма обычен.

Функциональный подход к нашей задаче полностью исключает ошибки, связанные с побочными эффектами. Возможное решение могло бы быть таким:

#--- Функциональный код для поиска/печати больших произведений на Python ----#

bigmuls = lambda xs,ys: filter(lambda (x,y):x\*y > 25, combine(xs,ys))

combine = lambda xs,ys: map(None, xs\*len(ys), dupelms(ys,len(xs)))

dupelms = lambda lst,n: reduce(lambda s,t:s+t, map(lambda l,n=n: [l]\*n, lst))

print bigmuls((1,2,3,4),(10,15,3,22))

Мы связываем в примере анонимные ('lambda') функции с именами, но это не необходимо. Вместо этого мы могли просто вложить определения. Мы использовали имена как ради большей читаемости, так и потому, что combine() - в любом случае отличная служебная функция (генерирует список всех возможных пар элементов из двух списков). В свою очередь, dupelms() в основном лишь вспомогательная часть combine(). Хотя этот функциональный пример более многословен, чем императивный, при повторном использовании служебных функций код в собственно bigmuls() окажется, вероятно, более лаконичным, чем в императивном варианте.

Реальное преимущество этого функционального примера в том, что в нем абсолютно ни одна переменная не меняет своего значения. Какое-либо неожиданное побочное влияние на последующий код (или со стороны предыдущего кода) просто невозможно. Конечно, само по себе отсутствие побочных эффектов не гарантирует безошибочность кода, но в любом случае это преимущество. Однако заметьте, что Python, в отличие от многих функциональных языков, не предотвращает повторное привязывание имен bigmuls, combine и dupelms. Если дальше в процессе выполнения программы combine() начнет значить что-нибудь другое - увы! Можно было бы разработать класс-одиночку (Singleton) для поддержки однократного связывания такого типа (напр. 's.bigmuls', etc.), но это выходит за рамки настоящей статьи.

Еще стоит отметить, что задача, которую мы только что решили, скроена в точности под новые возможности Python 2.0. Вместо вышеприведенных примеров - императивного или функционального - наилучшая (и функциональная) техника выглядит следующим образом:

#----- Код Python для "bigmuls" с использованием списочных встраиваний (list comprehensions) -----#

print [(x,y) for x in (1,2,3,4) for y in (10,15,3,22) if x\*y > 25]

Заключение

Эта статья продемонстрировала способы замены практически любой конструкции управления потоком в Python на функциональный эквивалент (избавляясь при этом от побочных эффектов). Эффективный перевод конкретной программы требует дополнительного обдумывания, но мы увидели, что встроенные функциональные примитивы являются полными и общими. В последующих статьях мы рассмотрим более мощные подходы к функциональному программированию; и, я надеюсь, сможем подробнее рассмотреть "pro" и "contra" функционального подхода.