**И опять о функциональном программировании на Python**

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

Предыдущие статьи коснулись основных понятий функционального программирования (ФП). Эта статья продолжит обсуждение, иллюстрируя дополнительные возможности, главным образом реализованные в библиотеке Xoltar Toolkit: частичное вычисление функций (Currying, карринг), функции высшего порядка (higher-order functions) и другие концепции.

**Что такое python?**

Python - свободно доступный, интерпретируемый язык программирования высокого уровня, разработанный Гвидо ван Россумом (Guido van Rossum). Он объединяет ясный синтаксис с мощной (но необязательно) объектно-ориентированной семантикой. Python может быть установлен на любой платформе и обеспечивает прекрасную совместимость при переходе с одной платформы на другую.

**Связывание выражений**

Недовольный полурешениями, один из читателей - Ричарда Дейвис (Richard Davies) - поднял вопрос, можем ли мы целиком переместить связывания в отдельные выражения. Давайте попытаемся понять, зачем нам может этого захотеться, а также продемонстрируем замечательно элегантный способ этого добиться, предоставленный участником comp.lang.python.

Давайте сначала вспомним о классе Bindings, определенном в модуле functional. Используя свойства этого класса, мы смогли гарантировать, что отдельное имя имеет единственное значение в пределах области данного блока:

#------- Python FP session with guarded rebinding -------#

>>> from functional import \*

>>> let = Bindings()

>>> let.car = lambda lst: lst[0]

>>> let.car = lambda lst: lst[2]

Traceback (innermost last):

File "", line 1, in ?

File "d:\tools\functional.py", line 976, in \_\_setattr\_\_

raise BindingError, "Binding '%s' cannot be modified." % name

functional.BindingError: Binding 'car' cannot be modified.

>>> let.car(range(10))

0

С помощью класса Bindings нам удалось достичь желаемого результата в пределах модуля или функции, но в отношении отдельного выражения мы бессильны. Тем не менее, для семейства ML-языков вполне естественно создавать связывания в пределах отдельного выражения:

#-------- Haskell expression-level name bindings --------#

-- car (x:xs) = x -- \*could\* create module-level binding

list\_of\_list = [[1,2,3],[4,5,6],[7,8,9]]

-- 'where' clause for expression-level binding

firsts1 = [car x | x <- list\_of\_list] where car (x:xs) = x

-- 'let' clause for expression-level binding

firsts2 = let car (x:xs) = x in [car x | x <- list\_of\_list]

-- more idiomatic higher-order 'map' technique

firsts3 = map car list\_of\_list where car (x:xs) = x

-- Result: firsts1 == firsts2 == firsts3 == [1,4,7]

Грэг Эвинг (Greg Ewing) заметил, что мы можем достичь того же эффекта, воспользовавшись списочными встраиваниями Python (list comprehensions); мы даже можем сделать это почти столь же ясным способом, как в Haskell:

#------ Python 2.0+ expression-level name bindings ------#

>>> list\_of\_list = [[1,2,3],[4,5,6],[7,8,9]]

>>> [car\_x for x in list\_of\_list for car\_x in (x[0],)]

[1, 4, 7]

Этот прием - размещение выражения внутри одноэлементного кортежа в списочном встраивании - не позволяет использовать связывание на уровне выражений с функциями высшего порядка. Для использования таких функций мы все еще должны использовать область блока:

#------- Python block-level bindings with 'map()' -------#

>>> list\_of\_list = [[1,2,3],[4,5,6],[7,8,9]]

>>> let = Bindings()

>>> let.car = lambda l: l[0]

>>> map(let.car,list\_of\_list)

[1, 4, 7]

Неплохо, хотя если мы хотим использовать map(), область связывания остается несколько шире, чем мы того хотели. Тем не менее, можно уговорить списочное встраивание делать для нас связывание имен, даже если список - не то, что нам нужно в конечном счете:

#---- "Stepping down" from Python list-comprehension ----#

# Compare Haskell expression:

# result = func car\_car

# where

# car (x:xs) = x

# car\_car = car (car list\_of\_list)

# func x = x + x^2

>>> [func for x in list\_of\_list

... for car in (x[0],)

... for func in (car+car\*\*2,)][0]

2

В этом примере мы произвели арифметическое действие над первым элементом первого элемента списка list\_of\_list и одновременно поименовали это действие (но только в области объемлющего выражения). В качестве "оптимизации" можно посоветовать создавать список длиной не более одного элемента, поскольку с помощью индекса [0] в конце выражения выбираем только первый элемент:

#---- Efficient stepping down from list-comprehension ---#

>>> [func for x in list\_of\_list[:1]

... for car in (x[0],)

... for func in (car+car\*\*2,)][0] 2

Функции высшего порядка: частичное вычисление функций - карринг[1] (currying)

Три наиболее общих функций высшего порядка встроены в Python: map(), reduce() и filter(). Эти функции используют в качестве (некоторых) своих параметров другие функции - вот почему мы называем их функциями высшего порядка. Другие функции высшего порядка (но не эти три) возвращают объекты-функции (function objects).

Python всегда предоставлял программистам возможность создавать свои собственные функции высшего порядка благодаря полноправному статусу функций как объектов. Ниже в качестве иллюстрации приведен простой пример:

#----------- Trivial Python function factory ------------#

>>> def foo\_factory():

... def foo():

... print "Foo function from factory"

... return foo

...

>>> f = foo\_factory()

>>> f()

Foo function from factory

Программа Xoltar Toolkit, о которой я упоминал в предыдущих статьях, содержит замечательный набор функций высшего порядка. Большинство этих функций, предоставляемых модулем functional, имеются во множестве традиционных функциональных языках программирования, и их полезность проверена многолетним использованием. Пожалуй, наиболее известная и важная функция высшего порядка традиционно называется curry(). Она названа в честь логика Хаскелла Карри (Haskell Curry), чьим именем назван уже упоминавшийся язык программирования. В основе карринга лежит допущение о том, что (почти) любую функцию можно рассматривать как частично вычисляемую функцию одного аргумента. Для того, чтобы эта идея работала, необходимо чтобы значение, возвращаемое функцией, само могло быть функцией, но возвращаемые функции должны быть уже или ближе к завершению . Этот механизм подобен замыканию, о котором я рассказывал в предыдущей статье - каждый вызов каррированой функции добавляет больше данных, необходимых для окончательного вычисления (данные прикрепляются к процедуре). Проиллюстрируем сначала карринг очень простым примером на Haskell, а затем повторим тот же пример на Python с помощью модуля functional:

#------------- Currying a Haskell computation -----------#

computation a b c d = (a + b^2+ c^3 + d^4)

check = 1 + 2^2 + 3^3 + 5^4

fillOne = computation 1 -- specify "a"

fillTwo = fillOne 2 -- specify "b"

fillThree = fillTwo 3 -- specify "c"

answer = fillThree 5 -- specify "d"

-- Result: check == answer == 657

А теперь на Python:

#------------- Currying a Python computation ------------#

>>> from functional import curry

>>> computation = lambda a,b,c,d: (a + b\*\*2 + c\*\*3 + d\*\*4)

>>> computation(1,2,3,5)

657

>>> fillZero = curry(computation)

>>> fillOne = fillZero(1) # specify "a"

>>> fillTwo = fillOne(2) # specify "b"

>>> fillThree = fillTwo(3) # specify "c"

>>> answer = fillThree(5) # specify "d"

>>> answer

657

Приведем еще один пример, подтверждающий, что между каррингом и замыканием много общего. Для этого, используя curry(), перепишем простую программу расчета налога, код которой можно найти в предыдущей статье:

#------------ Python curried tax calculations -----------#

from functional import \*

taxcalc = lambda income,rate,deduct: (income-(deduct))\*rate

taxCurry = curry(taxcalc)

taxCurry = taxCurry(50000)

taxCurry = taxCurry(0.30)

taxCurry = taxCurry(10000)

print "Curried taxes due =",taxCurry

print "Curried expression taxes due =", \

curry(taxcalc)(50000)(0.30)(10000)

В отличие от замыкания, при использовании curry( ) необходимо заполнять параметры в определенном порядке (слева направо). Но заметьте, в модуль functional также включен класс rcurry(), для которого отсчет начинается с другого конца (справа налево).

Обратите внимание на второй оператор print в этом примере - с одной стороны, это всего лишь тривиальное синтаксическое изменение - можно было бы просто вызвать taxcalc(50000,0.30,10000). Но с другой стороны, благодаря этому становится понятным идея о том, что каждая функция может быть функцией всего одного аргумента - весьма неожиданная идея для тех, кто с эти незнаком.

**Другие функции высшего порядка**

Помимо фундаментальной функции curry(), в модуль functional включает ряд интереснейших функций высшего порядка. Более того, не составляет большого труда самому написать свои собственные функции высшего порядка - с помощью или без этого модуля. Во всяком случае, можно воспользоваться идеями, представленными в нем.

По большей части функции высшего порядка ведут себя как "усовершенствованные" версии стандартных функций map(), filter() и reduce(). В большинстве случаев они действуют согласно следующему правилу: "принять функцию или функции и некоторые списки в качестве параметров, затем применить функцию (функции) к списку параметров". Это правило предоставляет потрясающие возможности для программирования. Другой принцип "принять набор функций и создать функцию, комбинирующую их функциональность". И опять-таки, возможны многочисленные вариации. Давайте взглянем, что предоставляет functional.

Функции sequential() и also() создают функцию, основанную на последовательность других функций. Функции-компоненты затем могут быть вызваны с одинаковым аргументом (аргументами). Главное различие между этими двумя функциями заключается в том, что в sequential() список функций принимается в качестве первого аргумента, а also() принимает список аргументов (каждый из которых должен быть функцией) переменной длины. В большинстве случаев их используют ради побочных эффектов составляющих функций, однако sequential() позволяет опционально задать, результат какой функции вернуть как комбинированное значение:

#---- Sequential calls to functions (with same args) ----#

>>> def a(x):

... print x,

... return "a"

...

>>> def b(x):

... print x\*2,

... return "b"

...

>>> def c(x):

... print x\*3,

... return "c"

...

>>> r = also(a,b,c)

>>> r

>>> r(5)

5 10 15

'a'

>>> sequential([a,b,c],main=c)('x')

x xx xxx

'c'

Функции disjoin() и conjoin() схожи с sequential() и also() в том смысле, что они также создают новые функции, которые применяют параметр(ы) к нескольким составляющим функциям. Но disjoin() выясняет, возвращает ли хотя бы одна из составляющих функций "истину" (true), а conjoin() выясняет, возвращают ли все функции "истину". При этом, когда это возможно, логика "короткого замыкания"[2], поэтому при их вызове часть побочных эффектов может не проявиться. joinfuncs() похожа на also(), но, в отличие от нее, возвращает кортеж результатов составляющих функций, а не выбирает одно значение.

В то время как вышеуказанные функции вызывают много функций с одинаковыми параметрами, функции any(), all() и none\_of() позволяют вызывать одну и ту же функцию для каждого значения из списка. В общем случае они подобны встроенным функциям map(), filter() и reduce(). Но, в отличие от последних, эти функции задают булевы (логические) вопросы касательно набора возвращаемых величин. Например,

#--------- Ask about collections of return values -------#

>>> from functional import \*

>>> isEven = lambda n: (n%2 == 0)

>>> any([1,3,5,8], isEven)

1

>>> any([1,3,5,7], isEven)

0

>>> none\_of([1,3,5,7], isEven)

1

>>> all([2,4,6,8], isEven)

1

>>> all([2,4,6,7], isEven)

0

Особый интерес для тех, кто неравнодушен к математике, представляет функция высшего порядка compose(). Композиция из нескольких функций формирует цепочку, направляя возврат одной функции на вход следующей. Программист, комбинирующий несколько функций, должен следить за тем, чтобы выход и вход соответствовали друг другу - но ведь это так в любом случае, если программист использует возвращаемое значение. Ниже приведен пример, поясняющий сказанное:

#----------- Creating compositional functions -----------#

>>> def minus7(n): return n-7

...

>>> def times3(n): return n\*3

...

>>> minus7(10)

3

>>> minustimes = compose(times3,minus7)

>>> minustimes(10)

9

>>> times3(minus7(10))

9

>>> timesminus = compose(minus7,times3)

>>> timesminus(10)

23

>>> minus7(times3(10))

23

**Некоторые рекомендации**

Надеюсь, этот обзор функций высшего порядка пробудит у читателя интерес к определенному стилю мышления. Во всяком случае, поэкспериментируйте с ними. Попытайтесь создать собственные функции высшего порядка; некоторые из них могут оказаться весьма полезными и мощными. Поделитесь со мной своими достижениями, и, возможно, в будущей статье мы обсудим оригинальные и замечательные идеи, которыми непрерывно снабжают меня читатели.