**Еще о функциональном программировании на Python**

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

Эта статья продолжает серию статей о функциональном программирования (ФП) на Python. В ней демонстрируется несколько достаточно сложных концепций ФП. Читателю окажется полезным введение в различные подходы программного решения задач.

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.

**Переходим на функциональное программирование?**

В предыдущей статье "Функциональное программирование на Python" были освещены основные понятия функционального программирования (ФП). В этой статье мы попытаемся немного углубиться в эту богатейшую концептуальную область. Библиотека Xoltar Toolkit Брина Келлера (Bryn Keller) окажет нам в этом неоценимую помощь. Основные возможности ФП Келлер представил в виде небольшого эффективного модуля на чистом Python. Помимо модуля functional, в Xoltar Toolkit входит модуль lazy, поддерживающий структуры, вычисляемые "только когда это необходимо". Множество функциональных языков программирования поддерживают отложенное вычисление, поэтому эти компоненты Xoltar Toolkit предоставят вам многое из того, что вы можете найти в функциональном языке наподобие Haskell.

**Присвоение значений**

Внимательный читатель помнит об упоминаемом мной ограничении функциональной техники, описанной в предыдущей статье. Речь шла о том, что ничто в Python не запрещает переприсваивания другого значения имени, ссылающемуся на функциональное выражение. В ФП под именами понимается всего лишь буквенное сокращение более длинных выражений, при этом подразумевается, что одно и то же выражение всегда приводит к одному и тому же результату . Если же уже определенному имени присваивается новое значение, это допущение нарушается. Предположим, мы определяем неколько сокращений для использования в нашей функциональной программе, как в следующем примере:

#---- FP-сессия Python с переприсваиванием приводит к неприятностям ---#

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

>>> cdr = lambda lst: lst[1:]

>>> sum2 = lambda lst: car(lst)+car(cdr(lst))

>>> sum2(range(10)) 1

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

>>> sum2(range(10)) 5

К несчастью, одно и то же выражение sum2(range(10)) вычисляется к разным результатам в двух местах программы, несмотря на то, что аргументы выражении не являются изменяемыми переменными.

К счастью, модуль functional предоставляет класс Bindings (предложенный Келлеру автором), предотвращающий такое переприсваивание (по крайней мере, случайное; Python не препятствует решительному программисту намеренно нарушать правила). Хотя использование Bindings требует немного дополнительного кода, это оградит вас от случайностей. Келлер обозначает экземпляр класса Bindings как let (я полагаю, из-за зарезервированного слова let в ML-языках программирования).

Например, мы могли бы сделать следующее:

#------- FP-сессия Python с защитой от переприсваивания -------#

>>> 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

Разумеется, реальная программа должна перехватить и обработать исключение BindingError, однако сам факт его возбуждения позволяет избежать целого класса проблем.

Помимо класса Bindings, functional содержит функцию namespace, предоставлюющую доступ к пространству имен (на самом деле, к словарю) из экземпляра класса Bindings. Это очень удобно, если вам нужно вычислить выражение в (неизменяемом) пространстве имен, определенном в Bindings. Функция eval() в Python позволяет проводить вычисление в пространстве имен. Следующий пример поясняет сказанное:

#----- FP-сессия Python, использующая неизменяемые пространства имен -----#

>>> let = Bindings() # "Real world" function names

>>> let.r10 = range(10)

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

>>> let.cdr = lambda lst: lst[1:]

>>> eval('car(r10)+car(cdr(r10))', namespace(let))

>>> inv = Bindings() # "Inverted list" function names

>>> inv.r10 = let.r10

>>> inv.car = lambda lst: lst[-1]

>>> inv.cdr = lambda lst: lst[:-1]

>>> eval('car(r10)+car(cdr(r10))', namespace(inv))

17

**Замыкание**

В ФП существует очень интересное понятие - замыкание (closure). На самом деле, эта идея оказалась настолько заманчивой для многих разработчиков, что реализована даже в нефункциональных языках программирования, таких как Perl и Ruby. Кроме того, похоже, что в Python 2.1 неизбежно будет включен лексический контекст[1], что на 99% приблизит нас к замыканиям.

Так что же такое замыкание? Стив Маджевски (Steve Majewski) замечательно охарактеризовал это понятие в одной из посвященных Python сетевых конференций:

Объект - это совокупность данных вместе с привязанными к ним процедурами...

Замыкание - это процедура вместе с привязанноой к ней совокупностью данных.

Другими словами, замыкание есть что-то наподобие функционального доктора Джекилла по отношению к объектно-ориентированному мистеру Хайду (или, возможно, наоборот). Замыкание, так же как и экземпляр объекта, есть способ представления функциональности и данных, связанных и упакованных вместе.

Давайте вернемся немного назад, чтобы понять, какую проблему решают как объекты, так и замыкания, и как эта проблема решается без них. Обычно результат, возвращаемый функцией, определяется контекстом во время ее вычисления. Самый общий - и, возможно, самый очевидный - способ определить этот контекст - это передать в функцию параметры, указывающие, какие значения должны обрабатываться. Но иногда имеется вполне очевидное различие между "фоновыми" и "приоритетными" параметрами - между тем, что делает функция в данный момент, и тем, как она сконфигурирована для выполнения многократных потенциальных вызовов.

Существует ряд способов поддержки фоновых аргументах, акцентируя внимание на приоритетных. Например, можно просто передавать каждый необходимый аргумент в функцию при каждом ее вызове. Часто это сводится к передаче ряда значений (или структуры со множеством членов) на протяжении всей последовательности вызовов, чтобы передать данные туда, где они могут потребоваться. Это иллюстрируется простым примером:

#--------- Python session showing cargo variable --------#

>>> def a(n):

... add7 = b(n)

... return add7

...

>>> def b(n):

... i = 7

... j = c(i,n)

... return j

...

>>> def c(i,n):

... return i+n

...

>>> a(10) # Pass cargo value for use downstream

17

В этом примере, параметр n в пределах функции b() нужен только для того, чтобы быть доступным для передачи в c().

Другое возможное решение - использование глобальных переменных:

#--- Сессия Python, показывающая использование глобальной переменной ---#

>>> N = 10

>>> def addN(i):

... global N

... return i+N

...

>>> addN(7) # Добавить глобальную переменную N к аргументу

17

>>> N = 20

>>> addN(6) # Добавить глобальную переменную N к аргументу

26

Глобальная переменная доступна в любой момент, где бы вы ни вызывали addN(), при этом вовсе не обязательно явно передавать фоновый контекст .

Несколько более "питоновская" техника - "заморозить" переменную в функции, используя для этого значение параметра по умолчанию во время определения функции:

#-------- Сессия Python, иллюстрирующая замороженную переменную --------#

>>> N = 10

>>> def addN(i, n=N):

... return i+n

...

>>> addN(5) # Добавить 10

15

>>> N = 20

>>> addN(6) # Добавить 10 (текущее значение N не играет роли)

16

Замороженная нами переменная, в сущности, замыкание. Некие данные прикреплены к функции addN(). В случае полного замыкания, все данные, присутствовавшие в момент описания этой функции, были бы доступны при ее вызове. Однако в данном примере (и во многих более серьезных) можно просто обеспечить доступ к достаточному количеству данных с помощью параметров по умолчанию. Ведь переменные, которые не используются функцией addN(), не играют никакой роли при ее вычислении.

Давайте рассмотрим объектный подход на примере несколько более насущной проблемы. В это время года мысли обычно заняты интерактивными программами, используемыми для вычисления налога. Они собирают различные данные - необязательно в определенном порядке - а затем в какие-то момент используют их при вычислении. Давайте создадим упрощенный вариант такой программы:

#----- Класс в стиле Python для вычисления налога ------#

class TaxCalc:

def taxdue(self):

return (self.income-self.deduct)\*self.rate

taxclass = TaxCalc()

taxclass.income = 50000

taxclass.rate = 0.30

taxclass.deduct = 10000

print "Pythonic OOP taxes due =", taxclass.taxdue()

В нашем классе TaxCals (точнее, в его экземпляре) мы можем собрать данные - в любом порядке - и как только у нас будут все необходимые элементы, можно будет вызвать метод объекта, чтобы выполнить вычисление над собранными данными. Все собрано в пределах экземпляра и, тем самым, разные экземпляры могут задавать разные наборы данных. Создание множества экземпляров, отличающихся друг от друга только своими данными, невозможно при использовании глобальных или "замороженных" переменных. Подход "грубой силы" может решить эту задачу, но можно видеть, что в реальном примере может понадобиться передача множества значений.

Теперь давайте посмотрим, как можно решить эту задачу с помощью ООП с использованием передачи сообщений (это похоже на Smalltalk или Self, так же как и на некоторые объектно-ориентированные варианты xBase, которыми я пользовался):

#------- Smalltalk-style (Python) tax calculation -------#

class TaxCalc:

def taxdue(self):

return (self.income-self.deduct)\*self.rate

def setIncome(self,income):

self.income = income

return self

def setDeduct(self,deduct):

self.deduct = deduct

return self

def setRate(self,rate):

self.rate = rate

return self

print "Smalltalk-style taxes due =", \

TaxCalc().setIncome(50000).setRate(0.30).setDeduct(10000).taxdue()

Возвращение self каждым установщиком позволяет нам рассматривать текущий экземпляр как результат вызова каждого метода. Как видно в дальнейшем, этот подход имеет интересные общие черты с использованием замыкания в ФП.

Работая с Xoltar toolkit, можно создавать полные замыкания, имеющие требуемое свойство объединения данные с функцией, а также множественные замыкания, несущие в себе различные наборы данных:

#------- Python Functional-Style tax calculations -------#

from functional import \*

taxdue = lambda: (income-deduct)\*rate

incomeClosure = lambda income,taxdue: closure(taxdue)

deductClosure = lambda deduct,taxdue: closure(taxdue)

rateClosure = lambda rate,taxdue: closure(taxdue)

taxFP = taxdue

taxFP = incomeClosure(50000,taxFP)

taxFP = rateClosure(0.30,taxFP)

taxFP = deductClosure(10000,taxFP)

print "Functional taxes due =",taxFP()

print "Lisp-style taxes due =", \

incomeClosure(50000,

rateClosure(0.30,

deductClosure(10000, taxdue)))()

Каждая описанная нами функция замыкания берет любые значения, определенные в ее области видимости, и привязывает эти величины к глобальной области видимости функционального объекта. Однако то, что кажется глобальной областью данной функции, необязательно совпадает как с "настоящей" глобальной областью действия модуля, так и с глобальной областью другого замыкания. Замыкание просто "несет данные с собой".

В нашем примере, чтобы поместить определенные значения в область действия замыкания, мы используем несколько частных функций (income, deduct, rate). Было бы достаточно просто изменить дизайн так, чтобы было можно присваивать произвольные значения. Кроме того, ради развлечения, мы используем в этом примере два слегка различных функциональных стиля. Первый последовательно привязывает дополнительные значения к области замыкания; сделав taxFP изменяемой, мы позволяем строкам добавить в замыкание появляться в любом порядке. Однако, если бы мы использовали неизменяемые имена наподобие tax\_with\_Income, нам пришлось бы расположить связывания в определенном порядке и передавать более ранние последующим. В любом случае, как только все необходимое привязано к замыканию, мы можем вызывать выращенную функцию.

Второй стиль, на мой взгляд, несколько похож на Lisp (исключительно из-за присутствия круглых скобок). Помимо эстетики, во втором стиле присутствуют два интересных момента.

Во-первых, полностью отсутствует привязывание имен. Второй стиль - одиночное выражение без использования директив[2]. (см. предыдущую статье, где объясняется, почему это играет роль).

Другая интересная деталь Lisp-стиля в том, насколько сильно его использование замыканий напоминает методы передачи сообщений a la Smalltalk, о которых говорилось выше. В обоих случаях значения накопливаются до вызова функции/метода taxdue() (оба в этих упрощенных версиях возбуждают исключения, если требуемые данные недоступны). Smalltalk-стиль на каждом шаге передает объект, в то время как Lisp-стиль - продолжение (continuation). Но если смотреть в корень, то функциональное и объектно-ориентированное программирование приводят почти к одному и тому же.

**Хвостовая рекурсия**

В этой статье мы еще немного вгрызлись в гранит функционального программирования. То, что осталось, меньше (и, возможно, проще) того, что сделано (название этого раздела - небольшая шутка; к сожалению, ее суть пока не разъяснена). Отличный способ дальнейшего освоения множества концепций ФП - просмотреть исходники модуля functional. Модуль прекрасно откомментирован и содержит примеры для большинства функций/классов. В этой статье не были рассмотрены некоторые мета-функции, упрощающие комбинацию и взаимодействие других функций. Они определенно стоят изучения для программиста на Python, стремящегося продолжить изучение подходов функционального программирования.