Python. Лекция 3. Элементы функционального программирования.

Что такое функциональное программирование?

Функции являются **абстракциями**, в которых детали реализации некоторого действия скрываются за отдельным именем. Хорошо написанный набор функций позволяет использовать их много раз. Стандартная библиотека Python содержит множество готовых и отлаженных функций, многие из которых достаточно универсальны, чтобы работать с широким спектром входных данных. Даже если некоторый участок кода не используется несколько раз, но по входным и выходным данным он достаточно автономен, его смело можно выделить в отдельную функцию.

Эта лекция более ориентирована на практические соображения, а не на теорию функционального программирования. Однако там, где нужно, будут употребляться и поясняться соответствующие термины.

Далее будут подробно рассмотрены описание и использование функций в Python, рекурсия, передача и возврат функций в качестве параметров, обработка последовательностей и итераторы, а также такое понятие как генератор. Будет продемонстрировано, что в Python функции являются объектами (и, значит, могут быть переданы в качестве параметров и возвращены в результате выполнения функций). Кроме того, речь пойдет о том, как можно реализовать некоторые механизмы функционального программирования, не имеющие в Python прямой синтаксической поддержки, но широко распространенные в языках функционального программирования.

Функциональное программирование - это стиль программирования, использующий только композиции функций. Другими словами, это программирование в выражениях, а не в императивных командах.

Как отмечает Дэвид Мертц (David Mertz) в своей статье о функциональном программировании на Python, "функциональное программирование - программирование на функциональных языках (LISP, ML, OCAML, Haskell, ...)", основными атрибутами которых являются:

- "Наличие функций первого класса (функции наравне с другими объектами можно передавать внутрь функций).
- Рекурсия является основной управляющей структурой в программе.
- Обработка списков (последовательностей).
- Запрещение побочных эффектов у функций, что в первую очередь означает отсутствие присваивания (в "чистых" функциональных языках)
- Запрещение операторов, основной упор делается на выражения. Вместо операторов вся программа в идеале - одно выражение с сопутствующими определениями.
- Ключевой вопрос: **что** нужно вычислить, а не **как**.
- Использование функций более высоких порядков (функции над функциями над функциями)".

Функциональная программа

В математике функция отображает объекты из одного множества (**множества определения функции**) в другое (**множество значений функции**). Математические функции (их называют **чистыми**) "механически", однозначно вычисляют результат по заданным аргументам. Чистые функции не должны хранить в себе какие-либо данные между двумя вызовами. Их можно представлять себе черными ящиками, о которых известно только то, что они делают, но совсем не важно, как.

Программы в функциональном стиле конструируются как **композиция** функций. При этом функции понимаются почти так же, как и в математике: они отображают одни объекты в другие. В программировании "чистые" функции - идеал, не всегда достижимый на практике. Практически полезные функции обычно имеют **побочный эффект**: сохраняют состояние между вызовами или меняют состояние других объектов. Например, без побочных эффектов невозможно представить себе функции ввода-вывода. Собственно, такие функции ради этих "эффектов" и используются. Кроме того, математические функции легко работают с объектами, требующими бесконечного объема информации (например, вещественные числа). В общем случае компьютерная программа может выполнить лишь приближенные вычисления.

Кстати, бинарные операции " `+` ", " `-` ", " `*` ", " `/` ", которые записываются в выражениях, являются "математическими" функциями над двумя аргументами -- **операндами**. Их используют настолько часто, что синтаксис языка программирования имеет для них более короткую запись. Модуль `operator` позволяет представлять эти операции в функциональном стиле:

```
>>> from operator import add, mul
>>> print add(2, mul(3, 4))
14
```

Функция: определение и вызов

Как уже говорилось, определить функцию в Python можно двумя способами: с помощью оператора `def` и `lambda` -выражения. Первый способ позволяет использовать операторы. При втором - определение функции может быть только выражением.

Забегая вперед, можно заметить, что методы классов определяются так же, как и функции. Отличие состоит в специальном смысле первого аргумента `self` (в нем передается экземпляр класса).

Лучше всего рассмотреть синтаксис определения функции на нескольких примерах. После определения соответствующей функции показан один или несколько вариантов ее вызова (некоторые примеры взяты из стандартной библиотеки).

Определение функции должно содержать список **формальных параметров** и **тело определения функции**. В случае с оператором `def` функции также задается некоторое имя. Формальные параметры являются локальными именами внутри тела определения функции, а при вызове функции они оказываются связанными с объектами, переданными как фактические параметры. Значения по умолчанию вычисляются в момент выполнения оператора `def`, и потому в них можно использовать видимые на момент определения имена.

Вызов функции синтаксически выглядит как `объект-функция(фактические параметры)`. Обычно объект-функция - это просто имя функции, хотя это может быть и любое выражение, которое в результате вычисления дает исполняемый объект.

Функция одного аргумента:

```
def swapcase(s):
    return s.swapcase()

print swapcase("ABC")
```

Функция двух аргументов, один из которых необязателен и имеет значение по умолчанию:

```
def inc(n, delta=1):
    return n+delta

print inc(12)
print inc(12, 2)
```

Функция с одним обязательным аргументом, с одним, имеющим значение по умолчанию и неопределенным числом именованных аргументов:

```
def wrap(text, width=70, **kwargs):
   from textwrap import TextWrapper
   # kwargs  - словарь с именами и значениями аргументов
   w = TextWrapper(width=width, **kwargs)
   return w.wrap(text)

print wrap("my long text ...", width=4)
```

Функция произвольного числа аргументов:

```
def max_min(*args):
  # args - список аргументов в порядке их указания при вызове
  return max(args), min(args)

print max_min(1, 2, -1, 5, 3)
```

Функция с обычными (позиционными) и именованными аргументами:

```
def swiss_knife(arg1, *args, **kwargs):
  print arg1
  print args
  print kwargs
  return None

print swiss_knife(1)
print swiss_knife(1, 2, 3, 4, 5)
print swiss_knife(1, 2, 3, a='abc', b='sdf')
# print swiss_knife(1, a='abc', 3, 4)  # !!! ошибка

lst = [2, 3, 4, 5]
dct = {'a': 'abc', 'b': 'sdf'}
print swiss_knife(1, *lst, **dct)
```

Пример определения функции с помощью `lambda` -выражения дан ниже:

```
func = lambda x, y: x + y
```

В результате `lambda` -выражения получается безымянный объект-функция, которая затем используется, например, для того, чтобы связать с ней некоторое имя. Однако, как правило, определяемые `lambda` -выражением функции, применяются в качестве параметров функций.

В языке Python функция может возвратить только одно значение, которое может быть кортежем. В следующем примере видно, как стандартная функция `divmod()` возвращает частное и остаток от деления двух чисел:

```
def bin(n):
    """Цифры двоичного представления натурального числа """
    digits = []
    while n > 0:
```

```
            n, d = divmod(n, 2)
            digits = [d] + digits
        return digits

print bin(69)
```

Примечание:

Важно понять, что за именем функции стоит объект. Этот объект можно связать с другим именем:

```
def add(x, y):
  return x + y
addition = add  # теперь addition и add - разные имена одного и того же объекта
```

Пример, в котором в качестве значения по умолчанию аргумента функции используется изменчивый объект (список). Этот объект - один и тот же для всех вызовов функций, что может привести к казусам:

```
def mylist(val, lst=[]):
  lst.append(val)
  return lst

print mylist(1),
print mylist(2)
```

Вместо ожидаемого `[1] [2]` получается `[1] [1, 2]`, так как добавляются элементы к "значению по умолчанию".

Правильный вариант решения будет, например, таким:

```
def mylist(val, lst=None):
  lst = lst or []
  lst.append(val)
  return lst
```

Конечно, приведенная выше форма может использоваться для хранения в функции некоторого состояния между ее вызовами, однако, практически всегда вместо функции с таким побочным эффектом лучше написать класс и использовать его экземпляр.

Рекурсия

В некоторых случаях описание функции элегантнее всего выглядит с применением вызова этой же функции. Такой прием, когда функция вызывает саму себя, называется **рекурсией**. В функциональных языках рекурсия обычно используется много чаще, чем итерация (циклы).

В следующем примере переписывается функция `bin()` в рекурсивном варианте:

```
def bin(n):
    """Цифры двоичного представления натурального числа """
    if n == 0:
      return []
    n, d = divmod(n, 2)
    return bin(n) + [d]

print bin(69)
```

Здесь видно, что цикл `while` больше не используется, а вместо него появилось условие окончания рекурсии: условие, при выполнении которого функция не вызывает себя.

Конечно, в погоне за красивым рекурсивным решением не следует упускать из виду эффективность реализации. В частности, пример реализации функции для вычисления `n` -го числа Фибоначчи это демонстрирует:

```
def Fib(n):
   if n < 2:
       return n
   else:
       return Fib(n-1) + Fib(n-2)
```

В данном случае количество рекурсивных вызовов растет экспоненциально от числа `n`, что совсем не соответствует временной сложности решаемой задачи.

В качестве упражнения предлагается написать итеративный и рекурсивный варианты этой функции, которые бы требовали линейного времени для вычисления результата.

Предупреждение:

При работе с рекурсивными функциями можно легко превысить глубину допустимой в Python рекурсии. Для настройки глубины рекурсии следует использовать функцию `setrecursionlimit(N)` из модуля `sys`, установив требуемое значение `N`.

Функции как параметры и результат

Как уже не раз говорилось, функции являются такими же объектами Python как числа, строки или списки. Это означает, что их можно передавать в качестве параметров функций или возвращать из функций.

Функции, принимающие в качестве аргументов или возвращающие другие функции в результате, называют **функциями высшего порядка**. В Python функции высшего порядка применяются программистами достаточно часто. В большинстве случаев таким образом строится механизм обратных вызовов (callbacks), но встречаются и другие варианты. Например, алгоритм поиска может вызывать переданную ему функцию для каждого найденного объекта.

Функция apply()

Функция `apply()` применяет функцию, переданную в качестве первого аргумента, к параметрам, которые переданы вторым и третьим аргументом. Эта функция в Python устарела, так как вызвать функцию можно с помощью обычного синтаксиса вызова функции. Позиционные и именованные параметры можно передать с использованием звездочек:

```
>>> lst = [1, 2, 3]
>>> dct = {'a': 4, 'b': 5}
>>> apply(max, lst)
3
>>> max(*lst)
3
>>> apply(dict, [], dct)
{'a': 4, 'b': 5}
>>> dict(**dct)
{'a': 4, 'b': 5}
```

Обработка последовательностей

Многие алгоритмы сводятся к обработке массивов данных и получению новых массивов данных в результате. Среди встроенных функций Python есть несколько для работы с последовательностями.

Под **последовательностью** в Python понимается любой тип данных, который поддерживает интерфейс последовательности (это несколько специальных методов, реализующих операции над последовательностями, которые в данном курсе обсуждаться не будут).

Следует заметить, что тип, основной задачей которого является хранение, манипулирование и обеспечение доступа к самостоятельным данным называется **контейнерным типом** или просто **контейнером**. Примеры контейнеров в Python - списки, кортежи, словари.

Функции range() и xrange()

Функция `range()` уже упоминалась при рассмотрении цикла `for`. Эта функция принимает от одного до трех аргументов. Если аргумент всего один, она генерирует список чисел от 0 (включительно) до заданного числа (исключительно). Если аргументов два, то список начинается с числа, указанного первым аргументом. Если аргументов три - третий аргумент задает шаг

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1, 10, 3)
[1, 4, 7]
```

Функция `xrange()` - аналог `range()`, более предпочтительный для использования при последовательном доступе, например, в цикле `for` или с итераторами. Она возвращает специальный `xrange` -объект, который ведет себя почти как список, порождаемый `range()`, но не хранит в памяти все выдаваемые элементы.

Функция map()

Для применения некоторой функции ко всем элементам последовательности применяется функция `map(f, *args)`. Первый параметр этой функции - функция, которая будет применяться ко всем элементам последовательности. Каждый следующий `n+1` -й параметр должен быть последовательностью, так как каждый его элемент будет использован в качестве `n` -го параметра при вызове функции `f()`. Результатом будет список, составленный из результатов выполнения этой функции.

В следующем примере складываются значения из двух списков:

```
>>> l1 = [2, 7, 5, 3]
>>> l2 = [-2, 1, 0, 4]
>>> print map(lambda x, y: x+y, l1, l2)
[0, 8, 5, 7]
```

В этом примере применена безымянная функция для получения суммы двух операндов ко всем элементам `l1` и `l2`. В случае если одна из последовательностей короче другой, вместо соответствующего операнда будет `None`, что, конечно, собьет операцию сложения. В зависимости от решаемой задачи, можно либо видоизменить функцию, либо считать разные по длине последовательности ошибкой, которую нужно обрабатывать как отдельную ветвь алгоритма.

Частный случай применения `map()` - использование `None` в качестве первого аргумента. В этом случае просто формируется список кортежей из элементов исходных последовательностей:

```
>>> l1 = [2, 7, 5, 3]
>>> l2 = [-2, 1, 0, 4]
>>> print map(None, l1, l2)
[(2, -2), (7, 1), (5, 0), (3, 4)]
```

Функция filter()

Другой часто встречающейся операцией является фильтрование исходной последовательности в соответствии с некоторым предикатом (условием). Функция `filter(f, seq)` принимает два аргумента: функцию с условием и последовательность, из которой берутся значения. В результирующую последовательность попадут только те значения из исходной, для которой `f()` возвратит истину. Если в качестве `f` задано значение `None`, результирующая последовательность будет состоять из тех значений исходной, которые имеют истинностное значение `True`.

Например, в следующем фрагменте кода можно избавится от символов, которые не являются буквами:

```
>>> filter(lambda x: x.isalpha(), 'Hi, there! I am eating an apple.')
'HithereIameatinganapple'
```

Списковые включения

Для более естественной записи обработки списков в Python 2 была внесена новинка: списковые включения. Фактически это специальный сокращенный синтаксис для вложенных циклов `for` и условий `if`, на самом низком уровне которых определенное выражение добавляется к списку, например:

```
all_pairs = []
for i in range(5):
  for j in range(5):
    if i <= j:
      all_pairs.append((i, j))
```

Все это можно записать в виде спискового включения так:

```
all_pairs = [(i, j) for i in range(5) for j in range(5) if i <= j]
```

Как легко заметить, списковые включения позволяют заменить `map()` и `filter()` на более удобные для прочтения конструкции.

В следующей таблице приведены эквивалентные выражения в разных формах:

В форме функции	В форме спискового включения
`filter(f, lst)`	`[x for x in lst if f(x)]`
`filter(None, lst)`	`[x for x in lst if x]`
`map(f, lst)`	`[f(x) for x in lst]`

Функция sum()

Получить сумму элементов можно с помощью функции `sum()`:

```
>>> sum(range(10))
45
```

Эта функция работает только для числовых типов, она не может конкатенировать строки. Для конкатенации списка строк следует использовать метод `join()`.

Функция reduce()

Для организации цепочечных вычислений (вычислений с накоплением результата) можно применять функцию `reduce()`, которая принимает три аргумента: функцию двух аргументов, последовательность и начальное значение. С помощью этой функции можно, в частности, реализовать функцию `sum()`:

```
def sum(lst, start):
  return reduce(lambda x, y: x + y, lst, start)
```

Совет:

Следует помнить, что в качестве передаваемого объекта может оказаться список, который позволит накапливать промежуточные результаты. Тем самым, `reduce()` может использоваться для генерации последовательностей.

В следующем примере накапливаются промежуточные результаты суммирования:

```
lst = range(10)
f = lambda x, y: (x[0] + y, x[1]+[x[0] + y])
print reduce(f, lst, (0, []))
```

В итоге получается:

```
(45, [0, 1, 3, 6, 10, 15, 21, 28, 36, 45])
```

Функция zip()

Эта функция возвращает список кортежей, в котором `i` -й кортеж содержит `i` -е элементы аргументов-последовательностей. Длина результирующей последовательности равна длине самой короткой из последовательностей-аргументов:

```
>>> print zip(range(5), "abcde")
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

Итераторы

Применять для обработки данных явные последовательности не всегда эффективно, так как на хранение временных данных может тратиться много оперативной памяти. Более эффективным решением представляется использование **итераторов** - специальных объектов, обеспечивающих последовательный доступ к данным контейнера. Если в выражении есть операции с итераторами вместо контейнеров, промежуточные данные не будут требовать много места для хранения - ведь они запрашиваются по мере необходимости для вычислений. При обработке данных с использованием итераторов память будет требоваться только для исходных данных и результата, да и то необязательно вся сразу - ведь данные могут читаться и записываться в файл на диске.

Итераторы можно применять вместо последовательности в операторе `for`. Более того, внутренне оператор `for` запрашивает от последовательности ее итератор. Объект файлового типа тоже (построчный) итератор, что позволяет обрабатывать большие файлы, не считывая их целиком в память.

Там, где требуется итератор, можно использовать последовательность.

Работа с итераторами рассматривается в разделе, посвященном функциональному программированию, так как итераторами удобно манипулировать именно в функциональном стиле.

Использовать итератор можно и "вручную". Любой объект, поддерживающий интерфейс итератора, имеет метод `next()`, который при каждом вызове выдает очередное значение итератора. Если больше значений нет, возбуждается исключение `StopIteration`. Для получения итератора по некоторому объекту необходимо прежде применить к этому объекту функцию `iter()` (цикл `for` делает это автоматически).

В Python имеется модуль `itertools`, который содержит набор функций, комбинируя которые, можно составлять достаточно сложные схемы обработки данных с помощью итераторов. Далее рассматриваются некоторые функции этого модуля.

Функция iter()

Эта функция имеет два варианта использования. В первом она принимает всего один аргумент, который должен "уметь" предоставлять свой итератор. Во втором один из аргументов - функция без аргументов, другой - стоповое значение. Итератор вызывает указанную функцию до тех пор, пока та не возвратит стоповое значение. Второй вариант встречается много реже первого и обычно внутри метода класса, так как сложно порождать значения "на пустом месте":

```
it1 = iter([1, 2, 3, 4, 5])

def forit(mystate=[]):
  if len(mystate) < 3:
    mystate.append(" ")
    return " "

it2 = iter(forit, None)

print [x for x in it1]
print [x for x in it2]
```

Примечание:

Если функция не возвращает значения явно, она возвращает `None`, что и использовано в примере выше.

Функция enumerate()

Эта функция создает итератор, нумерующий элементы другого итератора. Результирующий итератор выдает кортежи, в которых первый элемент - номер (начиная с нуля), а второй - элемент исходной последовательности:

```
>>> print [x for x in enumerate("abcd")]
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

Функция sorted()

Эта функция, появившаяся в Python 2.4, позволяет создавать итератор, выполняющий сортировку:

```
>>> sorted('avdsdf')
['a', 'd', 'd', 'f', 's', 'v']
```

Далее рассматриваются функции модуля `itertools`.

Функция itertools.chain()

Функция `chain()` позволяет сделать итератор, состоящий из нескольких соединенных последовательно итераторов. Итераторы задаются в виде отдельных аргументов. Пример:

```
from itertools import chain
it1 = iter([1,2,3])
it2 = iter([8,9,0])
for i in chain(it1, it2):
    print i,
```

даст в результате

```
1 2 3 8 9 0
```

Функция itertools.repeat()

Функция `repeat()` строит итератор, повторяющий некоторый объект заданное количество раз:

```
for i in itertools.repeat(1, 4):
    print i,
```

```
1 1 1 1
```

Функция itertools.count()

Бесконечный итератор, дающий целые числа, начиная с заданного:

```
for i in itertools.count(1):
    print i,
    if i > 100:
        break
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100 101
```

Функция itertools.cycle()

Можно бесконечно повторять и некоторую последовательность (или значения другого итератора) с помощью функции `cycle()`:

```
tango = [1, 2, 3]
for i in itertools.cycle(tango):
    print i,
```

```
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2
3 1 2 3 1
2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2
3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 . . .
```

Функции itertools.imap(), itertools.starmap() и itertools.ifilter()

Аналогами `map()` и `filter()` в модуле `itertools` являются `imap()` и `ifilter()`. Отличие `imap()` от `map()` в том, что вместо значения от преждевременно завершившихся итераторов объект `None` не подставляется. Пример:

```
for i in map(lambda x, y: (x,y), [1,2], [1,2,3]):
    print i,
```

```
(1, 1) (2, 2) (None, 3)
```

```
from itertools import imap
for i in imap(lambda x, y: (x,y), [1,2], [1,2,3]):
  print i,
```

```
(1, 1) (2, 2)
```

Здесь следует заметить, что обычная функция `map()` нормально воспринимает итераторы в любом сочетании с итерабельными (поддающимися итерациям) объектами:

```
for i in map(lambda x, y: (x,y), iter([1,2]), [1,2,3]):
    print i,
```

```
(1, 1) (2, 2) (None, 3)
```

Функция `itertools.starmap()` подобна `itertools.imap()`, но имеет всего два аргумента. Второй аргумент - последовательность кортежей, каждый кортеж которой задает набор параметров для функции (первого аргумента):

```
>>> from itertools import starmap
>>> for i in starmap(lambda x, y: str(x) + y, [(1,'a'), (2,'b')]):
...   print i,
...
1a 2b
```

Функция `ifilter()` работает как `filter()`. Кроме того, в модуле `itertools` есть функция `ifilterfalse()`, которая как бы добавляет отрицание к значению функции:

```
for i in ifilterfalse(lambda x: x > 0, [1, -2, 3, -3]):
    print i,
```

```
-2 -3
```

Функции itertools.takewhile() и itertools.dropwhile()

Некоторую новизну вносит другой вид фильтра: `takewhile()` и его "отрицательный" аналог `dropwhile()`. Следующий пример поясняет их принцип действия:

```
for i in takewhile(lambda x: x > 0, [1, -2, 3, -3]):
    print i,

print
for i in dropwhile(lambda x: x > 0, [1, -2, 3, -3]):
  print i,

1
-2 3 -3
```

Таким образом, `takewhile()` дает значения, пока условие истинно, а остальные значения даже не берет из итератора (именно не берет, а не высасывает все до конца!). И, наоборот, `dropwhile()` ничего не выдает, пока выполняется условие, зато потом выдает все без остатка.

Функция itertools.izip()

Функция `izip()` аналогична встроенной `zip()`, но не тратит много памяти на построение списка кортежей, так как итератор выдает их строго по требованию.

Функция itertools.groupby()

Эта функция дебютировала в Python 2.4. Функция принимает два аргумента: итератор (обязательный) и необязательный аргумент - функцию, дающую значение ключа: `groupby(iterable[, func])`. Результатом является итератор, который возвращает двухэлементный кортеж: ключ и итератор по идущим подряд элементам с этим ключом. Если второй аргумент опущен, элемент итератора сам является ключом. В следующем примере группируются идущие подряд положительные и отрицательные элементы:

```
import itertools, math
lst = map(lambda x: math.sin(x*.4), range(30))
for k, i in itertools.groupby(lst, lambda x: x > 0):
  print k, lst(i)
```

Функция itertools.tee()

Эта функция тоже появилась в Python 2.4. Она позволяет клонировать итераторы. Первый аргумент - итератор, подлежащий клонированию. Второй (`N`) -- количество необходимых копий. Функция возвращает кортеж из `N` итераторов. По умолчанию `N=2`. Функция имеет смысл, только если итераторы задействованы более или менее параллельно. В противном случае выгоднее превратить исходный итератор в список.

Собственный итератор

Для полноты описания здесь представлен пример итератора, определенного пользователем. Если пример не очень понятен, можно вернуться к нему после изучения объектно-ориентированного программирования:

```
class Fibonacci:
  """Итератор последовательности Фибоначчи до N"""

  def __init__(self, N):
    self.n, self.a, self.b, self.max = 0, 0, 1, N

  def __iter__(self):
    # сами себе итератор: в классе есть метод next()
    return self

  def next(self):
    if self.n < self.max:
      a, self.n, self.a, self.b = self.a, self.n+1, self.b, self.a+self.b
      return a
    else:
      raise StopIteration

# Использование:
for i in Fibonacci(100):
  print i,
```

Простые генераторы

Разработчики языка не остановились на итераторах. Как оказалось, в интерпретаторе Python достаточно просто реализовать **простые генераторы**. Под этим термином фактически понимается специальный объект, вычисления в котором продолжаются до выработки очередного значения, а затем приостанавливаются до возникновения необходимости в выдаче следующего значения. Простой генератор формируется функцией-генератором, которая синтаксически похожа на обычную функцию, но использует специальный оператор `yield` для выдачи следующего значения. При вызове такая функция ничего не вычисляет, а создает объект с интерфейсом итератора для получения значений. Другими словами, если функция должна возвращать последовательность, из нее довольно просто сделать генератор, который будет функционально эквивалентной "ленивой" реализацией. **Ленивыми** называются вычисления, которые откладываются до самого последнего момента, когда получаемое в результате значение сразу используется в другом вычислении.

Для примера с последовательностью Фибоначчи можно построить такой вот генератор:

```
def Fib(N):
  a, b = 0, 1
  for i in xrange(N):
    yield a
    a, b = b, a + b
```

Использовать его не сложнее, чем любой другой итератор:

```
for i in Fib(100):
  print i,
```

Однако следует заметить, что программа в значительной степени упростилась.

Генераторное выражение

В Python 2.4 по аналогии со списковым включением появилось **генераторное выражение**. По синтаксису оно аналогично списковому, но вместо квадратных скобок используются круглые. Списковое включение порождает список, а, значит, можно ненароком занять очень много памяти. Генератор же, получающийся в результате применения генераторного выражения, списка не создает, он вычисляет каждое следующее значение строго по требованию (при вызове метода `next()`).

В следующем примере можно прочесть из файла строки, в которых производятся некоторые замены:

```
for line in (l.replace("- ", " - ") for l in open("input.dat")):
  print line
```

Ничто не мешает использовать итераторы и для записи в файл:

```
open("output.dat", "w").writelines(
        l.replace("- ", " - ") for l in open("input.dat"))
```

Здесь для генераторного выражения не потребовалось дополнительных скобок, так как оно расположено внутри скобок вызова функции.

Карринг

Библиотека Xoltar toolkit (автор Bryn Keller) включает модуль `functional`, который позволяет упростить использование возможностей функционального программирования. Модуль `functional` применяет "чистый" Python. Библиотеку можно найти по этому адресу.

При **карринге** (частичном применении) функции создается новая функция, задавая некоторые аргументы исходной. Следующий пример иллюстрирует частичное применение вычитания:

```
from functional import curry
def subtract(x, y):
    return x - y

print subtract(3, 2)
subtract_from_3 = curry(subtract, 3)
print subtract_from_3(2)
print curry(subtract, 3)(2)
```

Во всех трех случаях будет выведено `1`. В следующем примере получается новая функция, подставляя второй аргумент. Вместо другого аргумента вставляется специальное значение `Blank`:

```
from functional import curry, Blank
def subtract(x, y):
    return x - y

print subtract(3, 2)
subtract_2 = curry(subtract, Blank, 2)
print subtract_2(3)
print curry(subtract, Blank, 2)(3)
```

Заключение

В этой лекции рассматривался принцип построения функциональных программ. Кроме того, было показано, что в Python и его стандартных модулях имеются достаточно мощные выразительные средства для создания функциональных программ. В случае, когда требуются дополнительные возможности, например, карринг, их можно легко реализовать или взять готовую реализацию.

Следует отметить, что итераторы - это практичное продолжение функционального начала в языке Python. Итераторы по сути позволяют организовать так называемые **ленивые вычисления** (lazy computations), при которых значения вычисляются только когда они непосредственно требуются.