List Comprehensions

Python provides an elegant mechanism for building a list by embedding a **`for`** within list brackets. This a termed a *List Comprehension.*

The general form is an expression, followed by a **`for`** to generate values, optionally followed by **`if`**s (to select or reject values) of additional **`for`**s.

In essence this is a procedural version of a **`map`**, without the need to actually provide a function to be mapped.

To begin with a simple example,

```
>>> [2*i for i in [1,2,3]]
[2, 4, 6]
```

This is the same as mapping the doubling function on the list `[1,2,3]`, but without an explicit function.

With an `if` to filter values, we might have:

```
>>> [2*i for i in [3,2,1,0,-1] if i != 0]
[6, 4, 2, -2]
```

We can also (in effect) nest `for`'s:

```
[(x,y) for x in [1,2,3] for y in [-1,0] ]

[(1, -1), (1, 0), (2, -1), (2, 0),
(3, -1), (3, 0)]
```

Function Definitions

Function definitions are of the form

```
def  name(args):
    body
```

The symbol **def** tells Python that a function is to be defined. The function is called **name** and **args** is a tuple defining the names of the function's arguments. The **body** of the function is delimited using indentation. For example,

```
def fact(n):
  if n<=1:
    return 1
  else:
    return n*fact(n-1)
>>> fact(5)
120
>>> fact(20L)
```

```
24329020081766400000L
>>> fact(2.5)
3.75
>>> fact(2+1J)
(1+3j)
```

Scalar parameters are passed by value; mutable objects are allocated in the heap and hence are passed (in effect) by reference:

```
>>> def asg(ar):
...     a[1]=0
...     print ar
...
>>> a=[1,2,3,4.5]
>>> asg(a)
[1, 0, 3, 4.5]
```

Arguments may be given a *default* value, making them *optional* in a call. Optional parameters must follow required parameters in definitions. For example,

```
 >>> def expo(val,exp=2):
...    return val**exp
...
>>> expo(3,3)
27
>>> expo(3)
9
>>> expo()
TypeError: not enough arguments;
expected 1, got 0
```

A *variable* number of arguments is allowed; you prefix the last formal parameter with a `*`; this parameter is bound to a *tuple* containing all the actual parameters provided by the caller:

```
>>> def sum(*args):
...     sum=0
...     for i in args:
...         sum=sum+i
...     return sum
...
>>> sum(1,2,3)
6
>>> sum(2)
2
>>> sum()
0
```

You may also use the name of formal parameters in a call, making the order of parameters less important:

```
>>> def cat(left="[",body="",
    right="]"):
...     return left+body+right
...
>>> cat(body='xyz');
'[xyz]'
>>> cat(body='hi there!'
   ,left='--[')
'--[hi there!]'
```

Scoping Rules in Functions

Each function body has its own local namespace during execution. An identifier is resolved (if possible) in the local namespace, then (if necessary) in the global namespace.

Thus

```
>>> def f():
...     a=11
...     return a+b
...
>>> b=2;f()
13
>>> a=22;f()
13
>>> b=33;f()
44
```

Assignments are to local variables, even if a global exists. To *force* an assignment to refer to a global identifier, you use the declaration

```
global id
```

which tells Python that in this function **id** should be considered global rather than local. For example,

```
>>> a=1;b=2
>>> def f():
...     global a
...     a=111;b=222
...
>>> f();print a,b
111 2
```

Other Operations on Functions

Since Python is interpreted, you can dynamically create and execute Python code.

The function **`eval(string)`** interprets **`string`** as a Python expression (in the current execution environment) and returns what is computed. For example,

```
>>> a=1;b=2
>>> eval('a+b')
3
```

`exec(string)` executes **`string`** as arbitrary Python code (in the current environment):

```
>>> a=1;b=2
>>> exec('for op in "+-*/":
print(eval("a"+op+"b"))')
3
-1
2
0
```

`execfile(string)` executes the contents of the file whose pathname is specified by **`string`**. This can be useful in loading an existing set of Python definitions.

The expression

```
lambda args: expression
```

creates an anonymous function with **`args`** as its argument list and **`expression`** as it body. For example,

```
>>> (lambda a:a+1)(2)
3
```

And there are definitions of **`map`**, **`reduce`** and **`filter`** to map a function to a list of values, to reduce a list (using a binary function) and to select values from a list (using a predicate):

```
>>> def double(a):
...   return 2*a;
...
>>> map(double,[1,2,3,4])
[2, 4, 6, 8]
```

```
>>> def sum(a,b):
...   return a+b
...
>>> reduce(sum,[1,2,3,4,5])
15
>>> def even(a):
...   return not(a%2)
...
>>> filter(even,[1,2,3,4,5])
[2, 4]
```

Generators

Many languages, including Java, C# and Python provide iterators to index through a collection of values. Typically, a **`next`** function is provided to generate the next value and **`hasNext`** is used to test for termination.

Python provides *generators*, a variety of function (in effect a co-routine) to easily and cleanly generate the sequence of values required of an iterator.

In any function a **`yield`** (rather than a **`return`**) can provide a value and suspend execution. When the next value is needed (by an invisible call to **`next`**) the function is resumed at the point of the **`yield`**. Further yields generate successive values. Normal

termination indicates that **`hasNext`** is no longer true.

As a very simple example, the following function generates all the values in a list **`L`** except the initial value:

```
>>> def allButFirst(L):
...     for i in L[1:]:
...         yield i

>>> for j in allButFirst([1,2,3,4]):
...     print j,

2 3 4
```

The power of generators is their ability to create non-standard traversals of a data structure in a clean and compact manner.

As an example, assume we wish to visit the elements of a list not in left-to-right or right-to-left order, but in an order that visits even positions first, then odd positions. That is we will first see `L[0]`, then `L[2]`, then `L[4]`, ..., then `L[1]`, `L[3]`, ...

We just write a generator that takes a list and produces the correct visit order:

```
>>> def even_odd(L):
...     ind = range(0,len(L),2)
...     ind = ind + range(1,len(L),2)
...     for i in ind:
...         yield L[i]
```

Then we can use this generator wherever an iterator is needed:

```
>>> for j in even_odd([10,11,12,13,14]):
...     print j,
...
10 12 14 11 13
```

Generators work in list comprehensions too:

```
>>> [j for j in even_odd([11,12,13])]
[11, 13, 12]
```

I/O in Python

The easiest way to print information in Python is the **`print`** statement. You supply a list of values separated by commas. Values are converted to strings (using the **`str()`** function) and printed to standard out, with a terminating new line automatically included. For example,

```
>>> print "1+1=",1+1
1+1= 2
```

If you don't want the automatic end of line, add a comma to the end of the print list:

```
>>> for i in range(1,11):
...     print i,
...
1 2 3 4 5 6 7 8 9 10
```

For those who love C's **printf**, Python provides a nice formatting capability using a printf-like notation. The expression

```
format % tuple
```

formats a tuple of values using a format string. The detailed formatting rules are those of C's **printf**. Thus

```
>>> print "%d+%d=%d" % (10,20,10+20)
10+20=30
```

File-oriented I/O

You open a file using

`open(name,mode)`

which returns a "file object."

`name` is a string representing the file's path name; **`mode`** is a string representing the desired access mode(**`'r'`** for read, **`'w'`** for write, etc.).

Thus

```
>>> f=open("/tmp/f1","w");
>>> f
<open file '/tmp/f1', mode 'w' at decd8>
```

opens a temp file for writing.

The command

```
 f.read(n)
```

reads **`n`** bytes (as a string).

`f.read()` reads the *whole file* into a string. At end-of-file, **`f.read`** returns the null string:

```
>>> f = open("/tmp/ttt","r")
>>> f.read(3)
'aaa'
>>> f.read(5)
' bbb '
>>> f.read()
'ccc\012ddd eee fff\012g h i\012'
>>> f.read()
''
```

`f.readline()` reads a whole line of input, and **`f.readlines()`** reads the whole input file into a list of strings:

```
>>> f = open("/tmp/ttt","r")
>>> f.readline()
'aaa bbb ccc\012'
>>> f.readline()
```

```
'ddd eee fff\012'
>>> f.readline()
'g h i\012'
>>> f.readline()
''
>>> f = open("/tmp/ttt","r")
>>> f.readlines()
['aaa bbb ccc\012', 'ddd eee
fff\012', 'g h i\012']
```

f.write(string) writes a **string** to file object **f**; **f.close()** closes a file object:

```
>>> f = open("/tmp/ttt","w")
>>> f.write("abcd")
>>> f.write("%d    %d"%(1,-1))
>>> f.close()
>>> f = open("/tmp/ttt","r")
>>> f.readlines()
['abcd1    -1']
```

Classes in Python

Python contains a class creation mechanism that's fairly similar to what's found in C++ or Java.

There are significant differences though:

- All class members are public.
- Instance fields aren't declared. Rather, you just create fields as needed by assignment (often in constructors).
- There are class fields (shared by all class instances), but there are no class methods. That is, all methods are instance methods.

- All instance methods (including constructors) must explicitly provide an initial parameter that represents the object instance. This parameter is typically called **`self`**. It's roughly the equivalent of **`this`** in C++ or Java.

Defining Classes

You define a class by executing a class definition of the form

```
class name:
   statement(s)
```

A class definition creates a class object from which class instances may be created (just like in Java). The statements within a class definition may be data members (to be shared among all class instances) as well as function definitions (prefixed by a **`def`** command). Each function must take (at least) an initial parameter that represents the class instance within which the function (instance method) will operate. For example,

```
class Example:
  cnt=1
  def msg(self):
    print "Bo"+"o"*Example.cnt+
     "!"*self.n
>>> Example.cnt
1
>>> Example.msg
<unbound method Example.msg>
```

Example.msg is unbound because we haven't created any instances of the **Example** class yet.

We create class instances by using the class name as a function:

```
>>> e=Example()
>>> e.msg()
AttributeError: n
```

We get the **`AttributeError`** message regarding **`n`** because we haven't defined **`n`** yet! One way to do this is to just assign to it, using the usual field notation:

```
>>> e.n=1
>>> e.msg()
Boo!
>>> e.n=2;Example.cnt=2
>>> e.msg()
Booo!!
```

We can also call an instance method by making the class object an explicit parameter:

```
>>> Example.msg(e)
Booo!!
```

It's nice to have data members initialized when an object is created. This is usually done with a constructor, and Python allows this too.

A special method named **`__init__`** is called whenever an object is created. This method takes **`self`** as its first parameter; other parameters (possibly made optional) are allowed.

We can therefore extend our **`Example`** class with a constructor:

```
class Example:
  cnt=1
  def __init__(self,nval=1):
    self.n=nval
  def msg(self):
    print "Bo"+"o"*Example.cnt+
          "!"*self.n
>>> e=Example()
>>> e.n
1
>>> f=Example(2)
>>> f.n
2
```

You can also define the equivalent of Java's **`toString`** method by defining a member function named **`__str__(self)`.**

For example, if we add

```
def __str__(self):
    return "<%d>"%self.n
```

to **`Example`**,

then we can include **`Example`** objects in **`print`** statements:

```
>>> e=Example(2)
>>> print e
<2>
```

Inheritance

Like any language that supports classes, Python allows inheritance from a parent (or base) class. In fact, Python allows *multiple inheritance* in which a class inherits definitions from more than one parent.

When defining a class you specify parents classes as follows:

```
class name(parent classes):
   statement(s)
```

The subclass has access to its own definitions as well as those available to its parents. All methods are virtual, so the most recent definition of a method is always used.

```
class C:
 def DoIt(self):
     self.PrintIt()
 def PrintIt(self):
     print "C rules!"

class D(C):
 def PrintIt(self):
     print "D rules!"
 def TestIt(self):
     self.DoIt()

 dvar = D()
 dvar.TestIt()
```

```
D rules!
```

If you specify more than one parent for a class, lookup is depth-first, left to right, in the list of parents provided. For example, given

```
class A(B,C): ...
```

we first look for a non-local definition in **`B`** (and its parents), then in **`C`** (and its parents).

Operator Overloading

You can overload definitions of all of Python's operators to apply to newly defined classes. Each operator has a corresponding method name assigned to it. For example, `+` uses **`__add__`**, `-` uses **`__sub__`**, etc.

Given

```
class Triple:
 def __init__(self,A=0,B=0,C=0):
   self.a=A
   self.b=B
   self.c=C
 def __str__(self):
   return "(%d,%d,%d)"%
    (self.a,self.b,self.c)
 def __add__(self,other):
   return  Triple(self.a+other.a,
    self.b+other.b,
    self.c+other.c)
```

the following code

```
t1=Triple(1,2,3)
t2=Triple(4,5,6)
print t1+t2
```

produces

```
(5,7,9)
```

Exceptions

Python provides an exception mechanism that's quite similar to the one used by Java.

You "throw" an exception by using a **`raise`** statement:

```
raise exceptionValue
```

There are numerous predefined exceptions, including **`OverflowError`** (arithmetic overflow), **`EOFError`** (when end-of-file is hit), **`NameError`** (when an undeclared identifier is referenced), etc.

You may define your own exceptions as subclasses of the predefined class **`Exception`**:

```
class badValue(Exception):
  def __init__(self,val):
     self.value=val
```

You catch exceptions in Python's version of a **`try`** statement:

```
try:
  statement(s)
except exceptionName_1, id_1:
  statement(s)
...
except exceptionName_n, id_n:
  statement(s)
```

As was the case in Java, an exception raised within the **`try`** body is handled by an **`except`** clause if the raised exception matches the class named in the

except clause. If the raised exception is not matched by any **except** clause, the next enclosing **try** is considered, or the exception is reraised at the point of call.

For example, using our **badValue** exception class,

```
 def sqrt(val):
  if  val < 0.0:
      raise badValue(val)
  else:
      return cmath.sqrt(val)


try:
  print "Ans =",sqrt(-123.0)
except badValue,b:
  print "Can't take sqrt of",
        b.value
```

When executed, we get

```
Ans = Can't take sqrt of -123.0
```

Modules

Python contains a module feature that allows you to access Python code stored in files or libraries. If you have a source file **`mydefs.py`** the command

```
import mydefs
```

will read in all the definitions stored in the file. What's read in can be seen by executing

```
dir(mydefs)
```

To access an imported definition, you qualify it with the name of the module. For example,

```
mydefs.fct
```

accesses **`fct`** which is defined in module **`mydefs`**.

To avoid explicit qualification you can use the command

from modulename import id_1, id_2, ...

This makes **id_1, id_2, ...** available without qualification. For example,

```
>>> from test import sqrt
>>> sqrt(123)
(11.0905365064+0j)
```

You can use the command

from modulename import *

to import (without qualification) *all* the definitions in modulename.

The Python Library

One of the great strengths of Python is that it contains a vast number of modules (at least several hundred) known collectively as the *Python Library*. What makes Python really useful is the range of prewritten modules you can access. Included are network access modules, multimedia utilities, data base access, and much more.

See

`www.python.org/doc/lib`

for an up-to-date listing of what's available.