IAP Python - Lecture 2

Evan Broder, Andrew Farrell

MIT SIPB

January 6, 2011

Higher Order Functions

A function is a first-class object, so it can be returned from a function.

```
def maketunk(n):
    def thunk():
        return n
    return thunk

>>> t = makethunk(5)
>>> t()
5
>>> ts = [makethunk(i) for i in range(6)]
>>> for t in ts:
...     print t()
0
1
2
3
4
5
```

Higher Order Functions

It can also be passed to a function.

```
def make_double(f):
    def f2(n):
        return f(n)+f(n)
    return thunk
def plus3(n):
    return n+3
>>> plus3(4)
7
>>> twoxplus6 = make_double(plus3)
>>> twoxplus6(4)
14
```

lambdas

for creating small throwaway functions, you can just use lambdas

```
>>> make_double = lambda f: (lambda n: f(n)+ f(n))
>>> plus3 = lambda n: n+3
>>> twoxplus6 = make_double(plus3)
>>> txplus6(4)
14
```

But don't do it too much. for complex functions, def is more readable.

Map and Filter

```
def map(func,seq):
    return [func(i) for i in seq)
>>> map(plus3,[1,2,3])
[4,5,6]

def filter(func,seq):
    return [i for i in seq if func(i)]
>>> iseven = lambda x: x%2 == 0
>>> filter(iseven,[1,2,3,4])
[2,4]

def reduce(func,seq):
    if len(seq) == 2:
        return func(seq[0],seq[1])
    return func(seq[0],map(func,seq[1:]))
# what is my error with the last one
>>> add = lambda x,y : x+y
>>> reduce(add,[1,2,3,4])
10
```

Default Values

You can specify default values for arguments to take on if nothing is passed in.

```
>>> def dinner(food='spam'):
...     print "What's for dinner?"
...     print food
...
>>> dinner('eggs')
What's for dinner?
eggs
>>> dinner()
What's for dinner?
spam
```

A function can have multiple arguments with default values, or some arguments with default values and some without. Arguments without default values are required, and if you pass in fewer positional arguments than are specified, values are assigned from left-to-right.

Keyword arguments

Instead of referring to arguments in order, you can also refer to them by name.
If a function's definition starts:

```
def login(username, password, date):
    ...
```

Then you can call the function using

```
login(password='sekkrit', date='tomorrow', username='brdoer')
```

It's an error to specify a keyword argument that's not in the function's argument list, or to specify a keyword more than once. Specify positional arguments before keyword arguments.

Accepting Arbitrary Arguments

If you want a function to take an arbitrary number of arguments, simply add a special argument to the function with a * at the beginning of its name. That argument will then contain a tuple of all extra positional arguments passed to the function (or an empty tuple if there are none).

```
>>> def sum(a, b, *args):
...   sum = a + b
...   print sum
...   for arg in args:
...     sum += arg
...   return sum
...
>>> sum(1, 2, 3, 4, 5)
3
15
```

Similarly, you can use ** to indicate a variable should contain all extra keyword arguments. The keyword arguments are put into a dictionary where `key=value` translates to {`"key": value`} (note that the key is automatically changed to a string).
The variables taking arbitrary arguments are frequently referred to as `*args` and `**kwarg s`.

Passing in Arbitrary Arguments

Just as *args can be used to receive all extra positional arguments, you can also use * to pass a list of arguments to a function, and ** to pass a dictionary of keyword arguments.

```
>>> def test(a, b, c):
...   print a, b, c
...
>>> args = (1, 2, 3)
>>> test(*args)
1 2 3
>>> args = (1, 2)
>>> test(5, *args)
5 1 2
```

Example: format()

New in 2.6 and 3.x is string.format() Any integer in braces is replaced with a corresponding positional argument.

```
>>> "I'm ordering {0}, {1}, {2}, and {0}".format('spam', 'eggs', 'sausage')
"I'm ordering spam, eggs, sausage, and spam"
```

any string in braces is replaced with a corresponding keyword argument

```
>>> "You don't frighten us,{plural_noun}. Go and {verb}, you sons of {place
"You don't frighten us,{nationality} pig dogs. Go and boil your bottoms, yo
```

Of course, you can nest format strings... but if you do it too much it will probably make your brain explode.

Break

Questions?

More String Processing

We looked at `print` last time as a way of viewing strings, but let's take a look at actually slicing and dicing them to do more interesting things. First, strings can be concatenated together with +. They can also be duplicated with *.

```
>>> 'spam' + 'eggs'
'spameggs'
>>> 'spam' * 5
'spamspamspamspamspam'
```

There are also several helper functions that you can use to manipulate strings, including `capitalize`, `title`, `upper`, `lower`, `swapcase`, and many others.

```
>>> 'spam and eggs'.capitalize()
'Spam and eggs'
>>> 'spam and eggs'.title()
'Spam And Eggs'
>>> 'spam and eggs'.upper()
'SPAM AND EGGS'
>>> 'spaM aNd eGGs'.swapcase()
'SPAm AnD EggS'
```

There are also features for combining lists of strings and breaking up strings into lists:

```
>>> 'spam and eggs'.split()
['spam', 'and', 'eggs']
>>> ' '.join(['spam', 'and', 'eggs'])
'spam and eggs'
```

(`split` can take one argument: the substring to split the string on.)

You can test for substrings with `'needle' in 'haystack'`, and test the beginning or end of a string with `startswith` and `endswith`. There's also `replace`.
`str` (the string type) has a bunch of other useful methods - to read more about them, consult `help(str)`.

The other way to format strings is using the % operator, which is similar to `printf` in other languages.

```
>>> "I'm ordering %s" % 'spam'
"I'm ordering spam"
```

If you want to pass in multiple values, use a tuple instead of a single string:

```
>>> "I'm ordering %s, %s, %s, and %s" % ('spam', 'eggs', 'sausage', 'spam')
"I'm ordering spam, eggs, sausage, and spam"
```

Using %s is almost always enough to get the formatting you want, but if you need something more complex, this construct does support the truly staggering set of type arguments of printf; it's documented at `http://docs.python.org/library/stdtypes.html#string-formatting`

Break

Questions?

Iterators

We...havent actually talked about `range` as a way to get a list of numbers. But really long lists of numbers can take a lot of memory to store. And if you try to create a big enough range of numbers, Python will give up.

```
>>> range(10000000000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: range() result has too many items
```

To work around this, Python has a concept of **iterators** - something that's more or less list-like, but pulls up elements one at a time. An iterator has one method that matters: `next`. Each time you call it, it spits out the next value. To take a sequence-like type and use it as an iterator, pass it to the `iter` function.

```
>>> lst = range(10)
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> iter(lst)
<listiterator object at 0x24ca70>
```

As an example of using iterators to save memory, instead of using `range` you can use `xrange`.

```
>>> xrange(10)
xrange(10)
>>> iter(xrange(10))
<rangeiterator object at 0x1e128>
>>> xr = iter(xrange(10))
>>> xr.next()
0
>>> xr.next()
1
>>> xr.next()
2
```

However, you don't usually interact directly with `iter` and `next`. When you specify what you're looping over in a `for` loop, Python actually loops over that thing passed to `iter`.

```
>>> for i in xrange(5):
...     print i
...
0
1
2
3
4
```

It's certainly possible to write your own objects that implement the "iterator protocol," but there's not really any need to, because Python provides several useful tools for generating iterators.

break

Break stops a loop right then and there.

```
>>> for i in xrange(100):
...     print i
...     if i > 3:
...         break
...
0
1
2
3
4
```

Generators

Many of the built-in types have some concept of iteration built in through iterables. You can also define your own classes which exhibit the qualities of iterables. Or instead, you can use this concept of generators to make an iterable out of a function.

Constructing Generators

To do this, use the `yield` keyword instead of returning. When you call the generator, it returns an iterable. Every time that *generator*`.next()` is called, the function picks up executing where it left off. If the function returns for any reasons, that's equivalent to ending the iteration. Because generators return an iterable object, then can be used in `for` loops as well.

Generators

is best shown by example:

```
>>> def myXrange(n):
...     a = 0
...     while a < n:
...         yield a
...         a = a + 1
...
>>> test = myXrange(3)
>>> test.next()
0
>>> test.next()
1
>>> test.next()
2
>>> test.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Constructing Generators

Every time you yield, the iterator returns another object, and when `next` is called again, the generator continues execution right where you left off. When the generator reaches the end of execution, it ends the sequence.

```
>>> for n in myXrange(10):
...     print n,
...
0 1 2 3 4 5 6 7 8 9
```

Generator Expressions

Another way to create iterators is using **generator expressions** (or genexps), which, whose only relation to generators is that they both return iterators.
A generator expression is basically a for loop rolled into a single line and wrapped in parentheses. Again, examples are the easiest explanation:

```
>>> (i * i for i in xrange(10))
<generator object at 0x24d760>
>>> gen = (i * i for i in xrange(10))
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
4
```

Generator statements are to Iterables what List Comprehensions are to Lists.

Generator Expressions

You can also include conditionals after the looping construct:

```
>>> gen = (i * i for i in xrange(10) if i % 2 == 0)
>>> gen.next()
0
>>> gen.next()
4
>>> gen.next()
16
```

Generator Expressions

Generator objects usually aren't assigned to variables like that, though. They're more typically passed to functions. When a genexp is the only argument to a function, you don't need the extra set of parentheses around it, so you can write:

```
>>> sum(i * i for i in xrange(10))
285
```

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

: