IAP Python - Lecture 2

Evan Broder

January 15, 2009

1 More on Functions

1.1 Returning Values

The one function we looked at last time didn't return anything, so let's look at a quick example that does. Returning a value is very straightforward:

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

1.2 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.

1.3 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.

1.4 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 `**kwargs`.

1.5 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
```

2 More on Sequences

2.1 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`

2.2 Iterators

2.2.1 xrange

We talked last time 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(1000000000)
Python(62322) malloc: *** mmap(size=4000002048) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
>>>
>>> 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.

2.2.2 Generators

You can use the `yield` keyword to write things called **generators**, which look almost but not exactly like functions. Instead of returning a value, you `yield` a value. This is best shown by example:

```
>>> def fibgen():
...   a, b = 0, 1
...   while True:
...     yield a
...     a, b = b, a + b
...
>>> fibgen()
<generator object at 0x24d760>
>>> fib = fibgen()
>>> fib.next()
0
>>> fib.next()
1
>>> fib.next()
1
>>> fib.next()
2
>>> fib.next()
3
```

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.

2.2.3 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
```

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 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
```

2.2.4 List Comprehensions

One specific case of genexps is the **list comprehension**. Put simply, if you use square brackets instead of parentheses, Python loops over the sequence completely and returns a list containing each element.

```
>>> [i * i for i in xrange(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2.2.5 A Diversion

A very quick diversion for people who have done functional programming. Using genexps and list comprehensions is the preferred way to do what in other languages would be known as `map` or `filter`.

Where in Scheme you would write `(map f lst)`, in Python you write

```
[f(i) for i in lst]
```

If in Scheme you would write `(filter f lst)`, in Python you write

```
[i for i in lst if f(i)]
```

3 When Something Goes Wrong

If you try to do something that Python can't deal with, it'll usually give you some information about where the problem is:

```
>>> 1 +
  File "<stdin>", line 1
    1 +
      ^
SyntaxError: invalid syntax
>>> 10 * (1 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

There are two classes of errors. The first is syntax errors (or `SyntaxError`). These are errors from Python attempting to parse the input, and can't be recovered from.

Any error that's not a `SyntaxError` is an exception. Python allows you to raise exceptions to represent something going wrong.

So what can you do about them? You can catch them using an `except` block.

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> try:
...   1/0
... except ZeroDivisionError:
...   print "Whoops! Can't divide by zero!"
...
Whoops! Can't divide by zero!
```

If you want to catch multiple types of exceptions, you can list them as a tuple, i.e. `except (KeyError, IndexError, TypeError)`. Occasionally, you may want to access the exception itself for more information. To do that, follow either the type of exception or the tuple of exceptions with a comma and then a variable to assign the exception to.

In this example, we open a file, which is discussed in the next section, but the concept is easy enough:

```
>>> open('laksdf')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'laksdf'
>>> try:
...   open('blahblahblah')
... except IOError, e:
...   print e.errno
...   print e.strerror
```

```
...
2
No such file or directory
```

You can catch any exception by specifying `except:` without a type of exception, but this can be dangerous - pressing Ctrl-C is a `KeyboardInterrupt` exception, so you may put yourself in a situation where it's hard to quit your program.

There are two other blocks you can use alongside `try` and `catch`. The `else:` block runs if no exceptions were raised. The `finally:` block runs whether exceptions were raised or not.

If you want to raise exceptions yourself you use `raise` and then an exception object. For example:

```
>>> raise TypeError("Can't do that!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't do that!
```

4 File I/O

To read or write from a file, you first need to open it using the `file` function. `file` takes a filename and an optional mode argment, and returns a "file handle", which you can then use to interact with the file.

The mode is either `'r'` ("read" - the default), `'w'` ("write"), `'a'` ("append"), or `'r+'` ("read and write"). Opening an existing file in `'w'` mode will delete its contents; `'a'` mode will not.

To read the contents of the file, you can use either the `read` method on the file handle. You can either pass `read` no arguments, in which case it reads the full contents of the file, or you can pass it an integer, in which case it reads that many bytes from the file.

```
>>> f = open('/usr/share/dict/words')
>>> print f.read(50)
A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
```

You can also loop over a file handle. Looping over a file handle returns one line (including the whitespace at the end) per iteration.

If you're writing to a file, you can use `write` instead of `read`.

File handles store a pointer to where they are in the file internally. If you want to read a part of a file again or move around within the file, you can use the `tell` and `seek` methods.

Finally, when you're done with a file, you can close it with `close`. It's not necessary for you to this—files are automatically closed by Python when they get garbage collected—but it may matter if you're opening a lot of files or if you're writing to files (sometimes write operations are delayed until the file is closed).

There's a lot of documentation on `file` types in Python under `help(file)`

5 Modules

Modules are a way to take a set of "things" (functions, classes when we learn them, or just variables) and use them in multiple files. They're also useful for organizing your code, or for using code that other people write.

5.1 `import`

To get at the functionality of a module, you `import` it. Then you can access the names in that file by prepending them with *module_name.*`name`.

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.e
2.7182818284590451
>>> math.cos
<built-in function cos>
```

You can also get rid of the need to type *module_name.* by using the `from...import` syntax:

```
>>> from sys import stderr
>>> stderr
<open file '<stderr>', mode 'w' at 0x230b0>
```

Writing a Python module is actually no different from writing a Python script! If you have a file ending in `.py`, it can be imported as a module.

To figure out where Python looks for modules, look at the `sys.path` list in the `sys` module. `sys.path` is the path Python searches for modules, starting with `''` for the current directory. Changing `sys.path` changes where Python looks for modules, e.g. adding an element causes Python to look in one more place. (Note: `sys.path` is unrelated to the `PATH` environment variable, but it can be altered by setting a `PYTHONPATH` environment variable before running Python)

Python modules should typically only contain definitions; they shouldn't do anything. If you want to write a Python module that does something when it's run directly, but can still be used as a module, use `if __name__ == '__main__':` to wrap the script portion of the module.

Now, let's look at some very common modules.

5.2 sys

The `sys` module contains things related to the Python runtime. This includes `sys.argv`, which is the list of arguments passed into your program from the command line; `sys.path`, which we already discussed earlier; `sys.stdin`, `sys.stdout`, and `sys.stderr`, which are file handles that you can read from and write to to interact with the terminal; and `sys.exit()`, which you can use to stop a script's execution.

5.3 math

We've already seen `e`, `pi`, and `cos` in the `math` module. It contains a bunch of other math functions as well. Try looking at `help(math)` for a full list of them.

5.4 os

The `os` module is very OS-specific, but this is how you would gain access to some lower level system calls. On UNIX, there are things like `os.symlink`, and `os.stat`. The `os.path` submodule is also very useful on UNIX systems.

5.5 this

Finally, as a less serious module, running `import this` will print out "The Zen of Python".