IAP Python - Lecture 1

Evan Broder, Andrew Farrell, and Karen Sittig

MIT SIPB

January 20, 2010

Why choose Python?

- Executable pseudocode—meaning that code is easy to read, easy to write, and easy to follow

```
def getLunch(food):
  if food == "spam":
    print "But I don't like spam!"
  else:
    print "That sounds lovely"
>>> getLunch("spam")
But I don't like spam!
```

- Automatic memory management
- Interpreted language—no compiling or linking; run your code immediately
- Multi-platform—move from Windows to OS X to Linux and not have to change or recompile your code
- Multi-paradigm programming—code in the style that works best for you, whether your style is functional, procedural, or object oriented

- Active community—lots of people and companies are using Python. In fact, Google found Python to be so important to how they did things that they hired Guido van Rossum, the creator of Python, to maintain it full-time.
  Some of the other major users of Python include YouTube, BitTorrent, NASA, Fedora (yum and anaconda), Gentoo (Portage), Ubuntu/Canonical
- With the New Curriculum, Course VI teaches using Python
- "Only one way to do it"—the tongue in cheek motto of Python. The point is that you'll be able to read your code in 6 months, unlike with certain other languages Cough...Perl...Cough
- Monty Python heritage—why use "foo" and "bar" when you can have "spam" and "eggs"?

Getting Python

For all platforms, you can download the latext version of Python at `http://www.python.org/download/`.
If you're using Windows, the Python installer also installs IDLE, a Python IDE and terminal.
If you're using Mac OS Leopard (10.5), your computer already has Python 2.5 installed.
If you're using Athena, your machine has Python, but it's an old version. SIPB maintains a `python` locker that has the current version of Python. Running `add -f python` will add the locker onto the beginning of your path so that the system finds it before the old version.
In this class, we will be primarily covering Python 2.6.

Interactive Python, or, Using Python as a Calculator

When you run `python` from a shell or open IDLE, you will get the primary prompt, ">>> ".
If you enter a multi-line command, you will get a different prompt: "... ".

```
Python 2.6.2 (release26-maint, Apr 19 2009, 01:56:41)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello, world!"
Hello, world!
```

How to Run Python Code

There are two main ways of running Python code. You can either run it from within the interactive Python prompt or as a script. On Mac OS X and Linux, to run code as a script, simply put it in a file with an appropriate shebang line (`#!/usr/bin/python`).
When you run code at the interactive prompt that simply returns a value, Python will print out that returned value. However, when you run code as a scrpit, nothing is printed unless you explicitly call `print`.

Arithmetic

You don't need to do anything special to represent numbers or do basic arithmetic in Python; the notation is the same as any other language. Python will automatically deal with negative numbers and parentheses. To do exponentiation, use the ** operator

```
>>> 1 + 1
2
>>> 6/3
2
>>> 4 - 9
-5
>>> (2-5) * 6
-18
>>> 2 ** 3
8
```

Note that whitespace in the middle of lines doesn't matter. Whitespace at the beginning of lines definitely matters, so be careful about that.

Arithmetic : Division

Normally, Python will floor the result of integer division operations, like any other language. If this is a problem, you can either explicitly specify floating point numbers or run `from __future__ import division`[1]

```
>>> 5 / 2
2
>>> 5 / 2.0
2.5
>>> from __future__ import division
>>> 5 / 2
2.5
```

[1] This is the default in Python 3.0

Arithmetic : Complex Numbers

You can even do complex numbers by specifying the imaginary part with "`j`" or "`J`" or by using calling "`complex(`*real*`, `*imag*`)`".

```
>>> (3+1j) * 3
(9+3j)
>>> (3+1j) * (-2+5j)
(-11+13j)
>>> (1j) * (1j)
(-1+0j)
```

Variables

Names of variables in Python are alphanumeric. They can contain underscores and can not start with a number.
To assign a value to a variable, use =, like most languages. You don't have to declare a variable before you can use it; assigning a value to a variable creates it.

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Types

There are lots of types that are built into Python with a rich set of supporting methods. Here are the types that you will actually be using. `None` is only sort of a real type. Rather, it represents the absense of a real value. When a function doesn't return something, it really returns `None` There are two booleans: `True` and `False`. In reality, they act like the integers 1 and 0, respectively, but their representations as strings are "`True`" and "`False`"

```
>>> 0 == False
True
>>> 0 == True
False
>>> 1 == True
True
>>> 2 == True
False
```

Types : Numbers

The normal integer representation is 32 bit and signed (so values can range between -2147483648 and 2147483647). However, Python also supports long integers, which can represent an unlimited range. Most operations will automatically return a long integer if the return value would otherwise overflow. Python appends the character "L" to the value

```
>>> 2147483647
2147483647
>>> 2147483647 + 1
2147483648L
```

They work like integers for most math, but the details of their implementation is dependent on the underlying code.
A complex number is essentially a pair of floating point numbers, one for the real and one for the imaginary. You can access the real and imaginary parts of a complex number `z` with `z.real` and `z.imag`. To get the magnitude, use `abs(z)`.

```
>>> z = (1+3j)
>>> z.imag
```

Sequences

Python has a number of data types that are made from a series of smaller items

Strings

Strings are made up of a series of characters. You can use single or double quotes. Escaping strings is similar to in C. There is also the "heredoc" notation for longer strings, where you use a triple-quote to mark the start and end. Heredoc strings can extend over multiple lines, but remember that everything between the first quotes and the last quotes is included.

```
>>> dead = """
... 'E's not pinin'! 'E's passed on! This parrot is no more! He
... has ceased to be! 'E's expired and gone to meet 'is maker!
... 'E's a stiff! Bereft of life, 'e rests in peace! If you
... hadn't nailed 'im to the perch 'e'd be pushing up the
... daisies! 'Is metabolic processes are now 'istory! 'E's off
... the twig! 'E's kicked the bucket, 'e's shuffled off 'is
... mortal coil, run down the curtain and joined the bleedin'
... choir invisibile!! THIS IS AN EX-PARROT!!
... """
>>> print dead

'E's not pinin'! 'E's passed on! This parrot is no more! He
has ceased to be! 'E's expired and gone to meet 'is maker!
'E's a stiff! Bereft of life, 'e rests in peace! If you
hadn't nailed 'im to the perch 'e'd be pushing up the
daisies! 'Is metabolic processes are now 'istory! 'E's off
the twig! 'E's kicked the bucket, 'e's shuffled off 'is
mortal coil, run down the curtain and joined the bleedin'
choir invisibile!! THIS IS AN EX-PARROT!!

>>>
```

Tuples

A tuple is simply a series of values separated by commas. They are frequently enclosed in parentheses, although it's not necessary. They're good for representing things like (x, y) coordinate pairs.
Tuples are immutable, so you can't change a single value in a tuple, but you can construct a new tuple to do the same thing.
To specify an empty tuple (i.e. 0 elements), just use a pair of parentheses.
For a tuple with a single element, put a single comma after it.

```
>>> fruitDefense = ()
>>> fruitDefense
()
>>> fruitDefense = ('gun',)
>>> fruitDefense
('gun',)
>>> fruitDefense = ('gun', '16-ton weight')
>>> fruitDefense
('gun', '16-ton weight')
>>> fruitDefense = ('gun', '16-ton weight', 'pointed sticks')
>>> fruitDefense
```

Tuples

Remember that even though all of my example tuples are strings, any type can go in a tuple, and all the elements of a tuple don't have to be the same type either.

One particular trick you can use tuples for is that if you have a tuple on both sides of an assignment, than each element on the right gets assigned to the corresponding element on the right. This is similar to the common Perl idiom `list($foo, $bar, $baz) = (1, 2, 3)`. It can also be used to swap two variables, if you ever have a need to do that.

```
>>> (a, b) = (1, 2)
>>> a
1
>>> b
2
>>> a, b = b, a
>>> a
2
>>> b
1
```

Lists

Lists are almost the same as tuples, but they're mutable, meaning that you can add remove and change values in a list. You can specify a list literally by using square brackets, and an empty list is just a pair of square brackets.

```
>>> weapons = []
>>> weapons = ['surprise']
>>> weapons.append("fear")
>>> weapons
['surprise', 'fear']
>>> weapons.append("ruthless efficiency")
>>> weapons.append("almost fanatical devotion to the Pope")
>>> weapons
['surprise', 'fear', 'ruthless efficiency', 'almost fanatical devotion
```

Dictionaries

A dictionary is also known as an associative array or a hash table in other languages. You can initialize a dictionary either by calling `dict()` or by using a pair of curly brackets.
Dictionaries are a type of mapping from some kind of key to some kind of value. Those keys can be any immutable type, which generally means either a string or a number.

```
>>> aussieWines = {}
>>> aussieWines['Black Stump Bordeaux'] = 'peppermint flavoured Burgund
>>> aussieWines['Sydney Syrup'] = 'sugary wine'
>>> aussieWines['Chateau Blue'] = 'taste and lingering afterburn'
>>> print aussieWines['Sydney Syrup']
sugary wine
>>> aussieWines
{'Black Stump Bordeaux': 'peppermint flavoured Burgundy', 'Chateau Blue
and lingering afterburn', 'Sydney Syrup': 'sugary wine'}
```

At the end you can also see the literal representation of a dictionary, which is {key1: value1, key2: value2, . . . }

Operations on sequences

Well, it's great that we can define these types, but how can we manipulate them?

Well, you can figure out how long they are:

```
>>> weapons = ['surprise', 'fear', 'ruthless efficiency']
>>> len(weapons)
3
```

Operations on sequences : Indexing

You can also use square brackets to grab a certain item in the list. Like all sensible computer list structures, lists and other sequences in Python are 0-indexed, meaning that the first item in the list is 0, the second is 1, etc.

```
>>> weapons[0]
'surprise'
>>> weapons[1]
'fear'
```

Operations on sequences : Indexing

Then there's slice notation. Slice notation is used for taking a subset of the whole list. The syntax is *listname*[start:end]

```
>>> menu = ['spam', 'eggs', 'bacon', 'sausage', 'spam']
>>> menu[3:5]
['sausage', 'spam']
```

Notice how the end is "1-indexed" while the beginning is "0-indexed." This takes a little getting used to, but imagine that the indices go between each item instead of being associated with a particular one. This is actually one of my favorite features of the language, because I tend to find that it always does what I mean.
Slice notation is amazingly flexible; you definitely have to play around with it some to get the hang of it. For some things to try, look at negative indices, not specifying one of the indices (*listname*[start:]), or assigning to slices.

Getting Help

This is a good point to stop and take a look at the help features that are built into Python. Python has a rich collection of documentation that's accessible from several different places.
Within an interactive shell, you can type `>>> help(`*type*`)` to get help.
You can also see what features it has by typing `>>> help(`*type*`)`
From a normal terminal, you can also run `pydoc` *type*
Or you can go to `http://www.pydoc.org/`, although that site has an unfortunate tendency to go down frequently.

Control Structures

Ok. So we have all of these data types to play with, and that's cool. But it's hard to do anything useful with a program without some kind of control structures. In fact, control structures are an essential part of this Turing-Complete breakfast.

If-Elif-Else

I can't explain this any better than an example can.

```
>>> x = int(raw_input("Please enter an integer: "))
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
```

For comparisons, Python uses == like most programming languages as an equality test. However, to make sure that you know what you're doing, it doesn't allow assignments within conditionals (so the idiom `if value = open('file')` won't work).

Comparators

Other than that, Python's logical operators are usually just words. For example, `and` and `or` are logical and and or. Numerical comparisons are <, >, <=, and >=. Python supports either `!=` or <>[2] for inequality, although `!=` is preferred.

You can also daisy chain comparisons, so `1 < x < 7` will Do The Right Thing.

Also, Python has a built-in test for list membership, so you can write things like

```
food = ["spam", "eggs", "sausage", "spam"]
if "spam" in food:
  print "I told you already. I don't like spam!"
```

Python does not have a `case` statement. Instead, use a series of `if`s and `elif`s or index into a tuple.

[2] Removed in Python 3.0

Defining Scope

This bit is a huge portion of why people either love or hate Python. Instead of using braces or brackets or parentheses as quotes, Python uses the indendation of a block of code. Note that in the last example, the `x = 0` line and the `print 'Negative changed to zero'` line were both executed because they were at the same indentation level.

It doesn't matter how far blocks are indented, but it must always be increasing. Also, mixing tabs and spaces at the beginning of a line is confusing, because in all these years people haven't agreed on just how wide a tab shold be. So, you should't use tabs, most people use four spaces.

Also, when you're in interactive Python, to let it know that you're done with a block, you just enter a blank line. It will issue the secondary prompt (`...`) if it thinks its still waiting for more input.

for

Python's `for` statement is much different from that of other languages. It's more akin to the `foreach` statement of other languages. Instead of iterating over a series of numbers, you iterate through list or tupe items. If you feed it a string, it ranges over characters, it ranges over items if youfeed it a list, and it ranges over keys if you feed it a dictionary.

```
>>> licenses = ['dog', 'cat', 'fish']
>>> for x in licenses:
...   print len(x), x
...
3 dog
3 cat
4 fish
```

range()

Most of the time, you can iterate over a list or a sequence to get the functionality that you want. But occasionally, you want to iterate over numbers like other programming languages. To do this, use the `range()` function. `range(`n`)` returns a list that goes from 0 to $n-1$.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(4,10)
[4, 5, 6, 7, 8, 9]
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...   print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Compound items

If your list contains, lets say, coordinate points, you can assign these to distinct variables

```
>>> for x,y in [(1,1),(1,2),(1,3)]:
...     print (y,x)
...
(1, 1)
(2, 1)
(3, 1)
```

If you want both the index and items in a list, use enumerate.

Enuemerate

If you want both the index and items in a list, use enumerate.

```
>>> for i,food in enumerate(['cantalopes','orangutans','breakfast cereals']):
...     print i
...     print food
...
0
cantalopes
1
orangutans
2
breakfast cereals
```

while

Python's `while` loops work approximately the same as any other programming language's loops. A `while` loop executes its body repeatedly as long as the condition is true.

```
>>> a, b = 0, 1
>>> while b < 10:
...   print b
...   a, b = b, a + b
...
1
1
2
3
5
8
```

Functions

Functions are defined with the `def` keyword. The syntax is fairly predictable once you've seen other control structures.

```
def find(shrubbery, forest):
  for plant in forest:
    if plant == "shrubbery":
      return plant
  reutrn "AAAAGH!"
```

Like most languages, variables that are defined within a function don't exist outside of the function by default. If you want to access a variable from outside the function use `global` *var* at the beginning of the function.

More functions

Here's another example function:

```
>>> def fib(n):
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Note that this function has a string at the top defined in the heredoc format. This is called the docstring, and pydoc uses these strings to generate help files. For example,

```
>>> help(fib)
Help on function fib in module __main__:

fib(n)
    Print a Fibonacci series up to n.
```

Recursion

Functions can call other functions, including themselves

```
>>> def factorial(num):
...     if num == 1:
...             return 1
...     else:
...             return num * factorial(num - 1)
...
>>> factorial(4)
24
```

More Information

If any of these concepts seem unclear, or if you want a little more information, check out the Official Python Tutorial at `http://docs.python.org/tut/tut.html`. This is how I learned Python, and it's the guideline for structuring this class. It's a great reference, especially if you already know how to program, and there are lots of simple examples to demonstrate concepts. The material covered in this session corresponds approximately to Chapters 1 through 5.