Posted on comp.lang.python:

Hi, Im a BSc4 Maths/Computer Science student. Unfortunately my curriculum did not include Python programming yet I see many vacancies for Python developers. I studied programming Pascal, C++ and Delphi. So I need to catch up quickly and master Python programming. How do you suggest that I achieve this goal? Is python platform independent? What is the best way? And how long would it take before I can develop applications using python? Can you recommend websites that feature a gentle introduction to Python?

Stop thinking about how to start and just start. Python is pretty intuitive, especially if you have other language background to relate to. Download the Python dist for your platform (Linux? probably already there - Windows? binary installers from python.org or activestate will install in a snap). Run through the first few pages of any of the dozen or more online tutorials to start getting your fingernails dirty. You'll need a text editor with integrated building - I find SciTE and PyScripter to be good for beginners (and I still use SciTE after 4 years of Python programming).

You know C++ and Pascal? You already know the basic if-then-else, while, and for control structure concepts.

Here are some C++-to- Python tips:

- There's no switch statement in Python. Make do with cascading if/elif/else until you come across the dict dispatch idiom.
- There's no '?' operator in Python. If you download the latest version (2.5), there is an equivalent x if y else z which would map to y ? x : zusing the ternary operator. But lean towards explict readability vs. one-liner obscurity at least for a few days.
- Forget about new/delete. To construct an object of type A, call A's constructor using newA = A()To delete A, let if fall out of scope, or explicitly unbind the object from the name "newA" with newA = NoneThis will allow newA to be cleaned up by the garbage collector at some future time.
- Forget about for (x = 0; x < 10; x++). Python loops iterate over collections, or anything with an __iter__ method or __getitem__ method. This is much more like C++'s for(listiter = mylist.first(); listiter != mylist.end(); ++listiter)

```
To force a for loop to iterate 'n' times, use
   for i in range(n):
```

The range built-in returns the sequence [0, 1, 2, ..., n-1].

Don't commit this beginner's blunder:

```
list1 = [ 1, 2, 3 ]
for i in range(len(list1)):
     # do something with list1[i]
```

```
for elem in list1:
    # do something with elem, which points to each element of
    # list1 each time through the loop

If you really need the list index, use enumerate, as in:
    for i,elem in enumerate(list1):
        print "The %d item of the list is %s" % (i,elem)
```

(Hey, check out those string formatting placeholders, they borrow heavily from C's printf notation. Oh, they don't teach that anymore, and you used iostreams in C++ instead? Bummer.)

- Forget about braces {}'s. For some reason, this is a big deal for some people, but give it a chance. Just indent code as you would normally, and leave out the braces. Personally I set my editor to replace tabs with spaces, this is a style choice but do NOT mix tabs and spaces. In the end, you will find this liberating, especially if you have ever been on a project that had to define a coding standard, and spent way too much time (more then 30 seconds) arguing about "where the braces should go."
- Don't forget the ()'s. To invoke a method on an object, you must include the parens. This wont do anything:

```
a = "some string"
a = a.lower
You need this:
a = a.lower()
```

• Stop thinking about variables as addresses and storage locations, and start thinking about them as values bound to names. Even so, I still find myself using words like "assignment" and "variable", when strictly I should be saying "binding" and "name".

What does Python have that C++ doesn't?

- The biggie: dynamic typing (sometimes called "duck typing"). Dynamic typing is a huge simplifier for development:
 - o no variable declarations
 - o no method type signatures
 - o no interface definitions needed
 - o no templating for collections
 - o no method overloading by differing argument type signatures

("Imagine there's no data types - I wonder if you can..."). What? No static type-checking at compile time? Nope, not really. If your method expects an object of type X, use it like an X. If it's not an X, you may be surprised how often this is not a problem. For instance, here's a simple debugging routine:

```
def printClassOf(x):
    print x.__class__.__name__
```

Every object has the attribute __class__ and every class has the attribute __name__. In C++,

I'd have to go through extra contortions *not* to type the variable x, probably call it something non-intuitive like "void*". Or look at this example:

```
def printLengthOf(x):
    print "Length of x is", len(x)
```

x could be any collection class, or user-defined class that is sufficiently like a collection to support len (such as implementing the __len__ method). This class doesn't even have to exist when you write printLengthOf, it may come along years later.

- An interactive interpreter. Awfully handy for just trying things out, without having to go through the compile/link/run cycle. Also good for getting at documentation on built-in and custom objects and methods type "help(blah)" to get help on method or class blah.
- Language built-in types for list, tuple (a type of list that is immutable), dict (akin to map<x,y> in the C++ STL), and set. Since Python does dynamic typing, no need to templatize these collection types, just iterate over them and use the objects in them.
 - o Lists look like [1, 2, 3, "ABC", [4,5]]
 - o Tuples look like ("Bob", "Smith", "12345 River St.", 52)
 - o Dicts look like { "Bob" : 52, "Joe" : 24 }
 - o Sets look like set("A", "B", "C")
- Language built-in types for string and unicode
- Multiple variable assignment you can unpack a list into individual variables using:

```
a,b,c = 1,2,3
list1 = [4,5,6]
a,b,c = list1 (will assign 4 to a, 5 to b, and 6 to c)
```

Forget about the classic C chestnut to swap a and b in-place:

```
Just do:
```

 $a \wedge = b$; $b \wedge = a$; $a \wedge = b$;

• Compound return types - need 3 or 4 values returned from a function? Just return them. No need for clunky make_pair<> templates, or ad hoc struct definitions just to handle some complex return data, or (ick!) out parameters. Multiple assignment will take care of this:

```
def func():
    return 4,5,6
a,b,c = func()
```

a,b = b,a

- Flexible and multiline quoting. Quoted string literals can be set off using ""s, "s, or triple quotes (""" """, or "" ""). The triple quote versions can extend to multiple lines.
- Built-in doc strings. If you have a function written like this: def func():

```
"A function that returns 3 consecutive ints, starting with 4" return 4,5,6
```

then typing "help(func)" at the interactive interpreter prompt will return the string "A function that

returns 3 etc.". This is called the function's docstring, and just about any object (class, function, module) can have one.

• A huge library of common application modules. The latest version includes support for the SQLite database.

And a part of the Python "getting it" that usually takes place in the first hour or two of *just starting* is encapsulated in the Zen of Python. Type "import this" at the interpreter command line, and you'll see a list of basic concepts behind the language and its design. It is true, there are some dorky in-jokes in there, but look past them and pick up the nuggets of Python wisdom.

Wow, are you still reading? Quit wasting time and go download a Python dist and get started already!

-- Paul McGuire March, 2007