Appendix 2. A brief introduction to Python

Python is an elegant programming language. Although it does not produce programs that execute as rapidly as those produced by Java or C++, it is very easy to write simple programs quickly, and when it comes to needing more speed it is not hard to interface to programs written in other languages. It has been around for more than a dozen years, but seems to have become popular only recently. It deserves its new-found popularity.

My own purpose in learning Python was to design a graphics tool to replace the programming in PostScript that I have been doing for many years. I believe that I have accomplished this, with what I call the PiScript modules. But since then I now use Python to write all kinds of simple programs in. For example, although I will not explain it here, the regular expression package for Python is far more convenient than than the one for PERL, and in fact I see no reason for anyone at any time from now on to write a program in PERL. Thank Heaven.

There are lots of sites in the Internet where one can find an endless amount of information about Python, so I am not going to write a full blown text here. What I am hoping is that this note will get you to the point where you can write simple programs and then be able to look around for information on how to write more advanced ones. I hope in particular that this will be just enough so that you you read my sample PiScript programs and then make your own. In learning Python, I myself used the book **Learning Python** written by Mark Lutz and published by O'Reilly, but gosh! it is awfully—and painfully—verbose.

Documents available on the Internet include

`http://docs.python.org/tut/tut.html` (official tutorial)

and a plethora of stuff listed at

`http://wiki.python.org/moin/BeginnersGuide/Programmers`

35. Starting

I believe in teaching programming by examples. Here is a program that calculates and displays the sum of the first 100 integers:

```
#!/usr/bin/python

# calculates sum of first 100 integers
s = 0
for i in range(1, 100):
    s += i
print "s = " , s
```

The name of the file it is in is `sum.py`.

It's pretty simple. Let's go through it line by line.

```
#!/usr/bin/python
```

This first line is for systems running some flavour of UNIX (such as Linux or MacOS). It is not necessary, but allows you to set up the file as an executable on these systems by setting its permissions flag to be 755. Otherwise, the normal way to execute the program is to type `python` plus the file name on a command line: `python sum.py`. It is not necessary that the file have extension `.py`, but it is a good idea.

```
# calculates sum of first 100 integers
```

This is a comment. The sign # makes the remainder of the line a comment. Longer multi-line comments can be enclosed between a matching pair of triple quotation marks `"""`.

```
s = 0
```

Like most other languages these days, `=` is variable assignment. One mildly eccentric but lovable feature of Python is that the types of variables don't have to be declared. Python knows here that `s` is an integer variable, and keeps track of that fact. This **dynamic typing** might occasionally cause trouble, but not often. Python will complain about a variable's type only when it is asked to perform some task with the variable that its type is not equipped for.

Statements may be ended with a semi-colon or not. Separate statements on a single line must be separated by semi-colons.

```
for i in range(1, 100):
```

This is one of the two common loops in Python. As before, `i` is not declared, and in fact its type is somewhat indeterminate in a `for` loop. The `range(1,100)` is actually Python list (basically, an array) of the integers 1, 2, ... , 99. You can see this by

```
print range(10)
```

What the loop does is just run through the array, setting `i` to be each element it encounters in turn.

```
    s += i
```

The only really eccentric feature of Python is the way it **requires** indentation by one or more tab spaces to mark blocks of code that are marked with `{ ... }` in C or Java. Even comments. Indentation has to agree with the code environment. The lines that require subsequent indentation end in a colon. These include function definitions, loop beginnings, and conditionals.

Unlike in C or Java, `++` and `--` are not part of the language.

```
print "s = " , s
```

The command `print` works pretty much in the predictable way. Output is to standard output. Normally `print` output is followed by a carriage return, but the comma annuls that. Putting commas in the middle of `print` statements just concatenates output with a little extra space but no carriage returns.

It doesn't show up in this short program, but there are two other characteristic features of Python. The first is that, unlike C or C++ but just like Java, you do not have to allocate memory for objects. This is both good and bad news, since clever memory handling is a major component of fast programs. Still, it is very, very convenient.

36. Data types

The built-in data types of Python are

integers
floating point numbers
Boolean
strings
lists
dictionaries
files
functions
classes

Integers. These are different in Python from what they are in most programming languages, in that they are of arbitrary size. In other words, whereas in Java or C the maximum ordinary integer is in the range $[-2^{31}, 2^{31}-1]$, in Python the transition from 2^{31} to $2^{31}+1$ is done correctly and silently. There is a cost to this amenability, since

in most machines the CPU handles integers modulo 2^{32}, and in some modulo 2^{64}. Therefore, as soon as large integers are encountered in Python they will take up more than one machine word, and dealing with them will be correspondingly slow.

Integer division gives the integral quotient. Thus $6/5$ returns 1. One wonderful feature of Python for a mathematician is that integer division returns the true integral quotient. So `n/m` is always the integer q such that $qm \leq n < (q+1)n$, and similarly `n % m` (i. e. n modulo m) always lies in $[0, m)$. So `-6/5` is -2 and `-6 % 5` is 4.

Floating point numbers. Nothing special, except that there seem to be no equivalent in Python of single precision floating point numbers (floats in Java). All are double precision. I do not know to what extent Python floating point numbers depend on the machine at hand, or whether the compiler takes the IEEE specifications into account.

In displaying these numbers, the default is to reproduce the full double precision expression. This often annoying behaviour can be modified by using a format string, as in C. Thus

```
print "%f" % 3.14159265358979
```

will produce `3.141593`. Other options allow more control. See

`http://www.webdotdev.com/nvd/articles-reviews/python/numerical-programming-in-python-938-139_3.html`
`http://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks`

for more information on formatting of floats. (The second site has a useful table around the middle.)

Because division of integers yields the integer quotient, you must be extremely careful when dividing variables that might be either integers or floating numbers. In these circumstances, it is safest to convert either numerator or denominator to a float first, say by multiplying one or the other by 1.0*. Or by using the operator* `float`*, which changes* x *to a float.*

Thus `float(1)` returns 1.0. When you do this, be careful to group terms with parentheses. Thus $x = 1.0 * a/b$ (probably incorrect, because a/b might be evaluated first) is not the same as $(1.0 * a)/b$ (probably what you want).

Strings. To build strings by concatenation of data types other than strings, you must use the `str` function. Thus

```
print "x = " + str(x)
```

But here,

```
print "x = ", x
```

will also work or even

```
print "x = ",
print x
```

There are functions to turn strings back into other types, too. Thus `int("3")` returns 3 and `float("3.3")` returns 3.3. These are useful in parsing command lines (discussed in a later section).

Booleans. These are `True` and `False`. Boolean operations are English words: `not`, `or`, `and`, `xor`, ...

Lists. A list is essentially an array. It is different from other types of array in Python that I never use, in that you can change its entries, and you can enlarge it or shrink it. Any sequence of items can be put into a list—lists can be arrays of objects of very different types. Lists are normally grown in steps. Thus to get a list `a = [0,1,2,3,4, ..., n-1]` one writes

```
a = []
for i in range(n):
    a.append(i)
```

but if the list is fixed ahead of time you can just write `a = [0,1,2,3,4]`. You build lists by appending data to it, and you can delete entries from it. The most useful deletion method is `pop`, which removes and returns the last item in the list. This allows you to simulate stacks in Python very easily.

Dictionaries. A dictionary is a special kind of list, one of of keys and values. It is the analogue of hash tables in other languages, but in Python is one of the basic types of data. Here is a sample dictionary:

```
d= {"red":[1,0,0], "green":[0,1,0], "blue":[0,0,1]}
```

You access the values for a given key very conveniently: `r = d["red"]`,a nd you can add entries to the dictionary equally conveniently: `d["orange"] = [1,0.5,0]`. You can check if a key is in the dictionary: `if d.has_key("red"): ...` , and in fact you should do that if you are not sure whether or not a key exists, because referring to `d[k]` when `d` does not have `k` as a key is an error. You can list all keys: `k = d.keys()`

Functions. A function is defined like this:

```
def sum(m, n):
    s = 0
    for i in range(m, n):
        s += i
    return s
```

If you are calling a function with no arguments, be sure to use parentheses, because functions are also objects. They can be passed as arguments. Thus

```
def newton(f, x):
    for i in range(10):
        fx = f(x)
        x -= fx[0]/fx[1]
    return(x)

def sqr(x):
    return([(x*x-2.0), 2.0*x])

print newton(sqr,1)
```

is OK. (Note how I have made `sqr` return floating point numbers.) Using functions as variables makes up to some extent for the lack of a `switch` or `case` in Python, which is for some of us a major and mysterious lack. You can find lots of puzzled complaints about this on the 'Net.

If you want to change the value of a global variable inside a function you must declare it to be global inside the function. Thus

```
def set(n):
    global a
    a = n
```

not

```
def set(n):
    a = n
```

Functions return the `nil` object by default.

Files. The basic procedures involved in using files for input and output are very simple.

```
f = open("output.txt", "w")
f.write("abc\n")
```

and

```
f = open("output.txt", "r")
s = f.read()
```

and

```
f = open("output.txt", "r")
s = f.readline()
while (s):
    s = f.readline()
```

But there are some things to watch out for. The most important is that when you are through using your file, you must close it. For one thing, files definitely don't flush their output until `flush()` or `close()` is called. For another, at least in Windows, you will not be able to do anything else with the file, such as remove it, until it is closed. Also more important in a Windows environment is that a binary file should be handled with a tag "b", as in "rb" or "wb", since otherwise Windows interprets some characters, such as EOF, specially.

Classes. As in other object oriented programming languages, you can create new data types. Here is one that defines a stack object:

```
class Stack:
    def __init__(self):
        self.list = []

    def push(self, x):
        self.list.append(x)

    def pop(self):
        return self.list.pop()

    def toString(s):
        return str(s.list)
    toString = staticmethod(toString)

# --- sample usage -----------

s = Stack()
s.push(0)
print Stack.toString(s)
n = s.pop()
```

This isn't a very useful class, since it does nothing more than lists, but if you want to feel at home in Python, maybe you'll use it. As you can see from this, Python can be rather `selfish`. This gets a bit annoying, I have to say. It is the analogue of `this` in Java, but whereas it is implicit in Java it is screaming right in your face in Python. For better or worse. When your program has `s.pop()`, this translates to `Stack.pop(s)` in the class—i. e. the calling instance becomes the first argument. One pleasant feature of classes is that you can overload certain operators like + and −, but I won't explain that here.

37. Variables

The most fascinating thing about variables in Python is that their types are dynamically determined. You should think of every thing in the program having a type that gets stuck to it transmitted in assignments. Variable types can vary. Thus

```
x = 3
x = "Hi, there!"
```

is an entirely legal successive pair of lines.

Also, as in Java, memory allocation is handled automatically by garbage collection. This is both good and bad news, since hideous efficiency in C++ programs often relies on clever memory allocation.

38. Loops

I use just two kinds, `for` and `while` loops. These are especially useful when combined with `break` (which exits the current loop) and `continue` (which goes back to the beginning of the loop) inside the loops. The basic `for` loop is standard:

```
for i in range(3, 17):
    ...
```

but `range` denotes here the array of numbers `[3, ... , 16]` and can be replaced by any kind of sequence.

39. Conditionals

```
if ...  :
    ...
elif ...  :
    ...
else:
    ...
```

Testing equality is done with `==`.

40. Modules

There are many packages in Python which may be optionally loaded. One of the most important is the `math` module, which is used in a program like this:

```
import math
...
c = math.sqrt(a*a + b*b)
...
C = 2*math.pi*r
```

or

```
from math import *
...
c = sqrt(a*a + b*b)
```

```
C = 2*pi*r
```

Importing basically makes names of variables and functions available. You have to be careful—in the last example, the `sqrt` from `math` will replace whatever `sqrt` function you have defined locally.

You can also make your own modules, which makes for much less redundant programming.

41. Command line arguments

If the file `sum` is

```
#!/usr/bin/python

import sys
m = int(sys.argv[1])
n = int(sys.argv[2])

# calculates sum of integers from m up to n
s = 0
for i in range(m, n):
    s += i
print "s = " , s
```

then

```
sum 1 10
```

will print out the correct sum. Also, unlike in some other languages, `argv[0]` is the command itself. Thus in

```
python sum.py 5 10
```

the 0-th argument is `sum.py`.

42. Learning more

The following list of web sites devoted to quirks of Python was brought to my attention by Christophe K.:

```
http://jaynes.colorado.edu/PythonIdioms.html
http://zephyrfalcon.org/labs/python_pitfalls.html
http://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks
http://www.onlamp.com/pub/a/python/2004/02/05/learn_python.html
http://www.ferg.org/projects/python_gotchas.html
```