

Python Primer
Spring 2003

David M. Reed, Ph.D.

January 21, 2003

Session 1

Python Primer

1.1 Background

Python is a open source language that is becoming more and more popular in the computing industry. It is both compiled and interpreted although the compilation step is hidden for the most part. It has an interactive interpreter that makes it easy to try commands and experiment. Python has support for object-oriented programming, but does not force you to use classes for everything as Java does. Python has many built-in data structures and modules for a wide variety of common tasks (regular expressions for text processing). Python does not have a built-in GUI toolkit, although it comes with support for Tk and there are bindings for many other widget sets (GTK+, Qt, wxWindows, etc.). The interpreter is written in C making it easy to integrate Python code with C and C++ code.

Because Python is interpreted, it is slower than C/C++, but if you use its built-in data structures and constructs correctly, it is fast enough for most applications. If you need more speed, it is possible to rewrite the code that needs optimized in C or C++ and call it from the Python code. It is fairly common to build the interface and much of the code with Python and use libraries written in C/C++ to do the time consuming calculations.

Python has many of the features of Perl, but its syntax is much simpler and cleaner. Python does not use semicolons at the end of statements. Indentation is used to denote blocks of code instead of braces. Variables do not need to be declared and are automatically dynamically allocated and deallocated.

Python comments begin at any point on the line with a # and continue to the end of the line. There is no multiline comment marker.

1.2 Data Types

- int: 3
- float: 2.5
- str: 'abc', "abc"
- list: [0, 1, 2], [0, 1, 'the']

- tuple: (0, 1, 2), (0, 1, 'the')
- dict: {'a': 1, 'Ohio': 'Columbus', 2: 'b'}
- the `str` and `tuple` types are immutable meaning you cannot change them

1.3 Interactive Interpreter

Below is an example of using the interactive interpreter that shows the basic data types of Python.

```
Python 2.2.1 (#3, Jul 24 2002, 10:20:36)
[GCC 2.95.3 20010315 (release)] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 3 * 4
14
>>> 5 / 2
2
>>> 5.0 / 2
2.5

>>> x = 'abc'
>>> y = 'def'
>>> x + y
'abcdef'
>>> x, y = y, x
>>> x
'def'
>>> y
'abc'
>>> 3*x
'defdefdef'

>>> l = range(0,10,2)
>>> l
[0, 2, 4, 6, 8]
>>> l[0]
0
>>> l[1], l[2]
(2, 4)
>>> l[-1]
8
>>> l[-2]
6
>>> l[2:4]
[4, 6]
>>> l[:-1]
[0, 2, 4, 6]
>>> l[1:]
[2, 4, 6, 8]
>>> l[2] = 10
>>> l
[0, 2, 10, 6, 8]
```

```
>>> l[3] = 'a'
>>> l
[0, 2, 10, 'a', 8]
>>> type(l)
<type 'list'>

>>> t = (0, 1, 2, 3)
>>> type(t)
<type 'tuple'>
>>> t[-1]
(0, 1, 2)
>>> t[2] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

>>> s = 'the quick brown fox jumps over the lazy dog'
>>> s.split()
['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
>>> s = 'dreed:fdasfjdsf:/home/faculty/dreed:/bin/csh'
>>> s.split(':')
['dreed', 'fdasfjdsf', '/home/faculty/dreed', '/bin/csh']

>>> d = {'a': 1, 'Ohio': 'Columbus', 2: 'b'}
>>> d['a']
1
>>> d['Ohio']
'Columbus'
>>> d[2]
'b'
>>> d[a]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
>>> d[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 3
>>> a = 2
>>> d[a]
'b'
>>> d.keys()
['a', 'Ohio', 2]
>>> d.values()
[1, 'Columbus', 'b']
>>> d.items()
[('a', 1), ('Ohio', 'Columbus'), (2, 'b')]
```

1.4 Basic Input/Output

The two basic functions for performing input are `input` and `raw_input`. Each function takes an optional string parameter that is the prompt for the input. The `raw_input` function returns

exactly what is entered as a string (without the end of line character generated by pressing return). The `input` function first evaluates the string that is entered and returns the result.

```
>>> x = input('enter a number: ')
enter a number: 3
>>> x
3
>>> x = input('enter a number: ')
enter a number: 3+4
>>> x
7
>>> x, y = input('enter two numbers: ')
enter two numbers: 3, 4
>>> x
3
>>> y
4
>>> x = raw_input('enter a number: ')
enter a number: 3
>>> x
'3'
>>> name = raw_input('enter name: ')
enter name: Dave Reed
>>> name
'Dave Reed'
```

The `print` method is used for simple output. Data inside of quotes (either the double quote or single quote character) are output exactly. Data not inside of quotes is evaluated and output. Each `print` statement puts a new line character at the end, unless a trailing comma is used. A comma is also used to separate multiple items being output on one line. Each comma inserts a space in the output.

```
>>> c = 4
>>> print 'a', 1+2, c
a 3 4
```

1.5 Decision and Looping Statements

Python uses whitespace to mark blocks of code. A colon is placed at the end of lines when the next line needs to be indented to mark a block of code. The `if` statement works almost the same as in C/C++. Parentheses are not necessary around the condition and the words `and`, `or`, and `not` are used instead of `&&`, `||`, and `!`. The same 6 comparison operators are used: `==`, `!=`, `<`, `<=`, `>`, and `>=`. The keyword `elif` is used instead of `else if`. Below are some examples:

```
if x > 0:
    print 'x > 0'

if x > 0 and y < 0:
    print 'a'
elif y < -10 or x < 5:
    if z < 10:
```

```
        print 'b'
    else:
        print 'c'
else:
    print 'd'
```

Python has two looping statements, the `for` loop and the `while` loop. It does not have a `do while` loop as C/C++ does. The `while` loop works the same as in C/C++ using the same syntax conventions as the Python `if` statement.

```
i = 0
while i < 10 and x > 0:
    x = input('enter a number')
    i = i + 1
    total = total + x
print x, total
```

The Python `for` statement is different from the C/C++ version. The syntax is:

```
for <var> in <sequence>:
```

A sequence is a list, tuple, or string. Each time through the `for` loop, the `var` gets the next value from the sequence. The Python `range` function can be used to generate a sequence of numbers. See the examples below:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(2, 7)
[2, 3, 4, 5, 6]
>>> range(2, 11, 3)
[2, 5, 8]
```

Since `range` generates a list, it can be inefficient for large values. The `xrange` function works in `for` loops but does not explicitly generate the list and should be used for large values.

```
>>> for i in range(4):
...     print i,
...
0 1 2 3
>>> for i in xrange(4):
...     print i,
...
0 1 2 3
>>> for c in 'the quick':
...     print c,
...
t h e   q u i c k
```

1.6 Functions

Python functions are created with the `def` statement. All parameters are passed by value. Functions may return zero or more values so pass by reference is not needed.

```
def sum_diff(a, b):
    return a+b, a-b

x, y = sum_diff(8, 3)
```

Any type can be passed (int, str, list, tuple, dict, and user-defined types). Python also provides for default values and allows functions to accept any number of parameters.

```
def f(a, b=3, c='abc'):
    print a, b, c

>>> def f(a, b=3, c='abc'):
...     print a, b, c

...
>>> f(1)
1 3 abc
>>> f(2, 'b')
2 b abc
>>> f(2, 'b', 4)
2 b 4

>>> def g(a, b, *args):
...     print a
...     print b
...     print args
...
>>> g(2, 3)
2
3
()
>>> g(2, 3, 4)
2
3
(4,)
>>> g(2, 3, 4, 5)
2
3
(4, 5)
>>> g([1, 2, 3], 4, 5, 6)
[1, 2, 3]
4
(5, 6)
```

1.7 Variables as References

All Python variables are references.

```
>>> x = 3
>>> y = x
>>> print id(x), id(y)
135278828 135278828
>>> x = 4
```

```
>>> y = 4
>>> print id(x), id(y)
135278816 135278816

l1 = [0, 1, 2]
l2 = [0, 1, 2]
>>> l1 = [0, 1, 2]
>>> l2 = [0, 1, 2]
>>> print id(l1), id(l2)
135463132 135630812
>>> l1 == l2
1
>>> l1 is l2
0
>>> l1 = [3, 4, 5]
>>> l2 = l1
>>> l1 is l2
1
>>> print id(l1), id(l2)
135565628 135565628
>>> print id(l1), id(l2)
135565628 135565628
>>> print id(l1[0]), id(l2[0])
135278828 135278828
>>> l1[0] = 6
>>> print l1, l2
[6, 4, 5] [6, 4, 5]
>>> print id(l1), id(l2)
135565628 135565628
```

1.8 Classes

Python classes are defined using the `class` keyword. Classes have a constructor named `__init__`. All Python methods have an explicit first parameter (named `self` by convention) that is the instance of the object it is called with (same concept as the implicit `this` in C++). The syntax for calling a Python method is the same as C++ (a period is always used instead of `->`). The explicit `self` parameter is not listed after the beginning parenthesis.

The `self` parameter is used to specify members of the class. In the example below, `sides` and `num_rolls` are *instance variables* for the class. Variables without the `self` prefix are local variables and are destroyed when the method ends.

```
from random import randrange

class Die:

    def __init__(self, sides):
        self.sides = sides
        self.num_rolls = 0

    def roll(self):
        self.num_rolls = self.num_rolls + 1
```

```
        r = randrange(1, self.sides+1)
        return r

    def roll_count(self):
        return self.num_rolls

    def call_roll(self):
        return self.roll()

d = Die(6) # causes the constructor to be called
print d.roll() # d is passed as the self parameter
print d.roll()
print d.roll_count()
```

1.9 File I/O

- Python syntax for opening files:
`<filevar> = open(<name>, <mode>)`
or in newer versions, the following also works
`<filevar> = file(<name>, <mode>)`

- there are three methods for reading a file:

```
<filevar>.read() # read entire file
<filevar>.readline() # read one line from file (can use this repeatedly)
<filevar>.readlines() # reads entire file creating a list of lines
```

- note: `readline()` will include the final newline character (whereas `raw_input()` removes it)
- example

```
infile = open('infile.txt', 'r')
for i in range(5):
    line = infile.readline()
    print line[:-1] # remove newline since print will go down to next line
```

- what would happen if we just did: `print line`?
- `readlines()` is useful for looping through entire file one line at a time

```
infile = open('infile.txt', 'r')
for line in infile.readlines():
    print line[:-1] # remove newline since print will go down to next line
```

- writing files is similar except that the mode `'w'` is used; note: opening a file for writing overwrites the file if it already exists!
- the `write` method takes a string as a parameter (numeric data must be converted to a string)

```
outfile = open('example.out', 'w')
count = 1
outfile.write('This is the first line\n')
count = count + 1
outfile.write('This is line number %d' % (count))
outfile.close()
```

- there are additional methods for writing (including a `writelines` that takes a list), but for now just worry about the `write` method
- closing a file you are writing is important as discussed above; not as crucial but it is good practice to close files opened for reading
- reading/writing binary files works the same for the most part; they are read and written as a string with escape characters to represent the unprintable characters outside the ASCII range; the Python `struct` module is used to convert back and forth between int/float and the byte representation of them

1.10 Modules

Any Python file is a module. To use one Python file in another file, you use the `import` statement. The example below shows how import works:

```
# mod.py
def f1():
    print 'f1 in mod.py'

def f2():
    print 'f2 in mod.py'
```

```
# main.py
import mod

from mod import f2

mod.f1()
f2()
```

You may also use `from mod import *` but then you need to worry about namespace conflicts (i.e., the same identifier being used in both files).

1.11 Built-in Modules

Python has a relatively small syntax (although newer versions of Python keep adding more features) and relies on modules to extend the basic capabilities of the languages. See: <http://www.python.org/doc/current/modindex.html>

for a list of the modules and documentation for them.

Some of the commonly used modules include: `os`, `sys`, `string`, `re`, `math`, `time`. I suggest you take a quick look at the online documentation for each of these.

1.12 Pychecker

Because Python is dynamic and does not use variable declarations and is very dynamic, it is easy to make mistakes that may not be caught unless you write test cases that cause every

single line of code to be executed. The following program has a problem, but as long as positive numbers are input, the program runs fine.

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

def main():
    x = input('enter a number: ')
    if x > 0:
        print x
    else:
        print z

if __name__ == '__main__':
    main()
```

The `pychecker` program was developed to help track down errors like this and many other. I suggest you copy the `.pycheckrc` from my account using the command:
`cp ~/dreed/pub/.pycheckrc ~/`

You can then run the `pychecker` program using the command:
`pychecker file.py`

1.13 Example

```
#!/usr/bin/env python
# Dave Reed
# CS160 1PM

import string

#-----

def madlibs():

    '''creates a madlib by reading a file with [] around words that are to be
    replaced and writes the output to a file.'''

    in_name = raw_input('Enter input filename: ')
    out_name = raw_input('Enter output filename: ')

    in_file = open(in_name, 'r')
    out_file = open(out_name, 'w')

    for line in in_file.readlines():
        # split line into words
        words = string.split(line)
        for w in words:
            # for each word, check if it starts with a [
            if w[0] == '[':
                out = raw_input('Enter a ' + w[1:-1] + ': ')
            else:
```

```
        out = w
        out_file.write(out + ' ')
        out_file.write('\n')
    out_file.close()

#-----

def main():
    madlibs()

#-----

if __name__ == '__main__':
    main()
```

1.14 Resources

- <http://www.python.org>
- <http://www.python.org/doc/current/tut/tut.html>
- <http://www.python.org/doc/current/modindex.html>
- <http://www.hetland.org/python/instant-python.php>