Python Tutorial

Python Language Reference

1.0 What Is Python

- Interpreted, object-oriented, programming language
- Used primarily for scripting
- Over 10 years old
- Growing in popularity very fast
- used by Yahoo!, Google, ILM, Ubisoft and many others.
- code can be run in an interactive interpreter or stored in a file and run like a script

2.0 Where Can I Get It ?

- http://www.python.org (official site)

3.0 Using the Interpreter

- How to run Python programs

3.a The Interpreter

- No need to compile and then run an executable
- Run your code in the interpreter
- Interactive mode is like a shell
- Allows you to try code out quickly
- Type `python.exe` (`python` on Unix) to start the interactive interpreter

3.b.i Environment

- No special requirements for basic usage
- Need to set PYTHONPATH environment variable to use modules that are not shipped with Python
- PYTHONPATH – a list of directories containing extra packages and modules

3.b.ii Executable Scripts

- A text file with Python code in it.
- Doesn't need to end with .py
- Best way to run script:
  - `python myscript.py`
- Otherwise, first line should be

  `#!/usr/bin/env python` on Unix and

  `#!C:\Python24\python.exe` on Windows

4.0 Language Intro

- Introduction to the basic elements and syntax of the language

4.a Comments

- describes the code, assumptions, etc.
- 3 ways of writing comments:

```
# This is a comment
"""This is a long comment.

It takes up 3 lines
"""
'''Here is another'''
```

- May use single- or double quotes.

4.b Variables

- used to store data
- first character in name must be letter or an underscore
- all other characters can be any letter, underscore or number.
- GOOD: numItems, START3, __myname
- BAD: 2points, -city

4.c Arithmetic

- Numbers: 2 ➔ integer, 2.0 ➔ float
- Supports basic arithmetic operations:
  - +, -, *, /
- Integer division truncates the result

  `5/2` ➔ `2`
- 2 ways to force the use of float:

  `5/2.0` ➔ `2.5`

  `5/float(2)` ➔ `2.5`
- force use of integer by using `int()`

4.c Arithmetic (cont'd)

- `int()` truncates the float value.

  `int(3.6)` ➔ `3`

- Use `round()` to round off float values

  `round(3.6)` ➔ `4.0`

  `round(3.4)` ➔ `3.0`

- Use `'%'` to get remainder

  `5%3` ➔ `2`

4.d Strings

- Many ways to specify a string:

  'single quotes', """triple quotes""", r"raw strings"

- Accessing characters in a string:
  - "hello"[0] ➔ "h" # indexing
  - "hello"[-1] ➔ "o" # (from end)
  - "hello"[1:4] ➔ "ell" # slicing
  - len("hello") ➔ 5 # length

4.d Strings (cont'd)

- Operations on strings:
  - `"hello" < "jello"` ➔True # comparison
  - `"e" in "hello "` ➔ True # basic search
  - `"hello".startswith('h')` ➔ True
  - `"hello".endswith('o')` ➔ True
  - `"hello "+"world"` ➔ "hello world" # concatenation
  - `"hello" * 3` ➔ "hellohellohello" # repetition

4.d Strings (cont'd)

- Operations on strings (cont'd)
  - `"  hi !   ".strip()` → `"hi !"` # strip whitespace
  - `print " Hello ! "` → `Hello !` # print to shell
- Special characters:
  - `\n` → newline
  - `\t` → tab
  - `\r` → carriage return
- Escape characters to avoid interpretation when using `print()`:

  `'The filename was\'t specified'`

  `"Unix uses / and DOS uses \\"` →

  `"Unix uses / and DOS uses \"`

4.e Lists

- store any number of items of any data type:

```
a = [99, "bottles ", ["on", "wall"]]
```

- Access the items like a string:

```
a[0], a[-1], a[1:4]
```

- Same operators as for strings:

```
a+b, a*3, len(a)
```

- Other operators:

```
[1,2].append([3,4]) ➔ [1,2,[3,4]]
[1,2].extend([3,4]) ➔ [1,2,3,4]
```

4.e Lists (cont'd)

- Item and slice assignment

```
a = [99, "bottles "]
a[0] = 3 ➔ [3, "bottles "]
a[1:2] = [”things", "of", "beer"] ➔
  [3, ”things", "of", "beer"]
```

- Delete an item
  - `del a[-1]➔ [3, ”things", "of"]`

4.e List (cont'd)

- Operations (cont'd)
  - a = range(5) ➔ [0,1,2,3,4]
  - a.append(5) ➔ [0,1,2,3,4,5]
  - a.pop() ➔ 5 # [0,1,2,3,4]
  - a.insert(0, 42) ➔ [42,0,1,2,3,4]
  - a.pop(0) ➔ 42 # [0,1,2,3,4]
  - a.reverse() # [4,3,2,1,0]
  - a.sort() # [0,1,2,3,4]

5.0 Control Flow

5.a `if` statements

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

5.b Loops

- used to repeat a set of statements
- Two types of loops: **`while`** and **`for`**

5.b Loops (cont'd)

- Repeat a set of instruction while a given condition is True

```
while condition:
    statements
```

- Repeat a set of instructions a certain number of times

```
for item in sequence:
    statements
```

5.b Loops (cont'd)

- **`break`** -- Stop looping immediately
- **`continue`** – go to the next iteration immediately
- **`pass`** -- do nothing

5.c Functions

- Simple definition of a function:

```
def MyFcn():
   #statments
   return some_value

def MyFcn( arg1, arg2, … ):
   #statements
   return some_value
```

5.c Functions (cont'd)

- keyword arguments:

```
myFcn(arg1='NTSC', arg2='10')
```

- Default arguments:

```
def myFcn( arg1, arg2="8")
```

5.c Functions (cont'd)

- Documentation

```
"""The first line is a blurb

The next lines contain whatever
info you want
"""
```

or

```
"""The first line only"""
```

5.c Functions (cont'd)

- Global variables

```
LAST_UPDATE= "10:30" # global variable
def doUpdate( data ):
    global LAST_UPDATE
    newTime = "11:00" # local variable
    LAST_UPDATE=newTime
```

6.0 Data Structures

- Using a list as a stack (last come, first served):

```
a.append(newItem) # add to stack
newItem = a.pop() # remove from stack
```

- Using a list as a queue (first come, first served):

```
a.append(newItem) # add to queue
newItem = a.pop(0)# remove from queue
```

6.b The `del` Statement

- Deleting from a list:

```
del a[0]    # delete the first item
del a[1:3]  # delete items 2 to 4
del a       # delete entire list
```

6.c Tuples and Sequences

- Tuple is a immutable (not modifiable) list:

```
t = (1234, "Yonge", 1.5, ["Jeff", "Lynn"])
t = (1234,) # singleton. Need trailing comma
t = ()  # empty tuple
```

- Access items like with lists and strings:

```
t[0] ➔ 1234, t[1:3] ➔ ('Yonge', 1.5)
```

- Compare tuples like lists:

```
(1,2) < (2,3) ➔ True
```

6.c Tuples and Sequences

- Unpacking

```
point = (128, 34, 255)

x, y, z = point
```

6.d Dictionaries

- Hash tables, "associative arrays "

```
d = {“Jim”: “red”, “Lisa”: “blue”}
```

- Accessing items:

```
d[“Jim”] ➔ “red“
d[“Dan”] # ERROR raises exception
```

- Delete, insert, overwrite:

```
d[“Dan”] = “green” #insert item
del d[“Jim”]    # delete item
d[“Lisa”] = “yellow” #overwrite item
```

6.d Dictionaries (cont'd)

- Merging dictionaries:

```
d.update(newDict)
```

- Checking if an item exists:

```
d.has_key('Dan') ➔ True
d.has_key('Maria') ➔ False
```

or

```
'Dan' in d ➔ True
'Maria' in d ➔ False
```

6.d Dictionaries (cont'd)

- Dictionaries in loops

```
for key, value in d.iteritems():
    print key, value
```

is the same as

```
for key in d.keys():
    print key, d[key]
```

6.d Dictionaries (cont'd)

- Keys, values, items:

  `d.keys()` ➔ list of all keys

  `d.values()` ➔ list of all items

  `d.items()` ➔ list of all key/item pairs as tuples

  e.g. `[("Dan","green"), ("Lisa", "yellow")]`

6.d Dictionaries (cont'd)

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is hashing (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

7.0 Modules

- What is a module ?
  - a file containing Python definitions and statements.
  - the module filename must end with **`.py`**
  - the name of the module is the filename, without the **`.py`**
    - e.g. the filename for the **`datetime`** module is **`datetime.py`**

7.b How To Use Modules

- Import a module:

```
import sys  # import the module
print sys.platform  # use it

import platform as pl   # pl is an alias
print "System = " + pl.system()

from getpass import getuser # import function
print "User = " + getuser() # use the function

from os import *  # import everything from os
```

7.c How Python Finds Modules

- First searches its installation directory for module files.
- Second, searches the paths in PYTHONPATH environment variable for module file.
- Interpreter adds value of PYTHONPATH to **`sys.path`** variable.

7.d Creating a Module

- Create a file called modulename.py
- Add the file's path to the PYTHONPATH environment variable
- import it

7.e Packages

- A way of organizing modules by grouping them together.
- A package is a directory with a file called **`__init__.py`**
- Parent directory must be in Python search path
- Import a package like you import a module

```
import maya
```

- Import a module from a package

```
from maya import OpenMaya
```

or

```
import maya.OpenMaya
```

7.e Packages (cont'd)

- Creating a package
  - create a file called `__init__.py` in the directory that you want to be a package.
  - put the module files in the package directory.

7.f Common Standard Modules

- os, sys, platform, re, datetime

8.0 Input and Output

- Fancier output formatting:

```
results = ['Italy', 'France', 'Germany']
for i, v in enumerate(results):
    print "%d: %s" % (i, v)
1: Italy
2: France
3: Germany
```

- Formatters:

```
%s → string
%d → decimal
%f → float
```

8.0 Input and Output

- Fancier output formatting:

  `%02d` → two digits, add leading zeros

  `%.3f` → round up to 3 decimal places

```
"Roses are %(roses)s" %({'roses':'red'})
```

8.b Reading and Writing Files

```
inFile = open("MyInFile.txt", "r")
outFile = open("MyOutFile.txt", "w")
try:
    allLines = inFile.readlines()
    for i, line in enumerate(allLines):
        print "Line %d: %s" %(i, line)
        outFile.write(line)
finally:
    inFile.close()
    outFile.close()
```

8.b Reading and Writing Files

- opening modes:

  r → read only. DEFAULT

  w → write only

  r+ → read and write

  a → write only, append

- More operations:

  `read()` → read the entire file as one string

  `readline()` → read the next line

8.c Directories

- Path operations – use os.path module

  `dirname(), basename(), exists(), split(), isfile(), isdir()`

- Create a path name:

  `os.path.join('C:\', 'Maya8.5', 'Python')` → `C:\Maya8.5\Python`

or

  `os.sep.join(['C:\', 'Maya8.5', 'Python'])` → `C:\Maya8.5\Python`

8.c Directories (cont'd)

- List a directory:

```
os.listdir("C:\Maya8.5")
```

9.0 Classes

- A structure that groups data and functions into one unit.
- Similar to a dictionary, but more powerful
- Core element of object-oriented programming
- help organize data and functions better

9.0 Classes (cont'd)

```
class Circle:
  def __init__( self ):
      self.x = self.y = 0
      self.radius = 1
      self.color = 'black'

  def move(self, x, y):
      if x <= self.x: x = self.x
      if y <= self.y: y = self.y
      self.x = x
      self.y = y
```

9.0 Classes (cont'd)

- use it like this:

```
>>> c = Circle()
>>> print "Origin=(%d, %d)" %(c.x,c.y)
Origin=(0, 0)
>>> c.move(8,9)
>>> print "Origin=(%d, %d)" %(c.x,c.y)
Origin=(8, 9)
>>> c.color="red"
>>> print "Color=" + c.color
red
```

10.0 Errors and Exceptions

- Exceptions are objects
- Exceptions are used as errors in Python
- Exceptions stop the execution
- Handle exceptions using `try/except`

```
try:
     myfile.write(someString)
except IOError, e:
     print "Unable to write to file:"+\
          str(e)
     raise   # re-raise exception
```

10.0 Errors and Exceptions (cont'd)

- Raise an exception:

```
raise RuntimeError("My error msg")
```

11.0 Pattern Matching

- Use regular expressions for complex string searches
- Allows you to specify patterns like with grep:
  - Test*.py
  - ^\s*Time \w+:\s*\d+$
- Can save the sub-string found

11.0 Pattern Matching (cont'd)

\w == a-z, A-Z, 0-9, _

\s == white space

\d == digit 0-9

. == any character

* == 0 or more occurances

+ == at least one occurance

? == one occurance

^ == beginning of the string

$ == end of the string

11.0 Pattern Matching (cont'd)

```
import re
pyRE = re.compile('^\w+.pyc$', re.I)
num = 0
for fname in os.listdir('C:\Python24'):
  mo = pyRE.search(fname)
  if mo:
     num += 1
print "Found %d Python files." % num
```

11.0 Pattern Matching (cont'd)

```
import re
pyRE = re.compile('^(\w+).pyc$', re.I)
modNames = []
for fname in os.listdir('C:\Python24'):
  mo = pyRE.search(fname)
  if mo:
     modNames.extend(mo.group(1))
print "Found %d Python files." % \
          len(modNames)
```

12.0 Date and Time

- Use **`datetime`** module to get **`datetime`** objects:

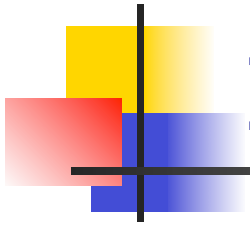
```
a=datetime.datetime.now()  # get current time
time.sleep(5)              # sleep for 5 seconds
b=datetime.datetime.now()
c=b-a   # calculate difference. Gives timedelta obj.
print "Diff=" + str(b-a)  # print the difference
(a + c) == b →True
```

13.0 Running External Commands

- Run a command in a sub-shell and don't capture the output:
  - os.system("mycommand")
- Use `os.popen()` to run a command and get its output

14.0 Sending E-mail

```
def sendMail( recipients, subject, message,
              fromaddr=None ):
  toaddrs = []
  for rp in recipients:
      if type(rp) == types.TupleType:
              rp = email.Utils.formataddr(rp)
      toaddrs.append(rp)
  subject = 'Subject: %s' % subject
  toheader = 'To: %s\n' % ', '.join(toaddrs)
  msg = subject.strip() + '\n' +toheader.strip()\
              + '\n\n' + message
  server = smtplib.SMTP(dlglobals.mail_server)
  server.sendmail(fromaddr, toaddrs, msg)
```



15.0 Tools

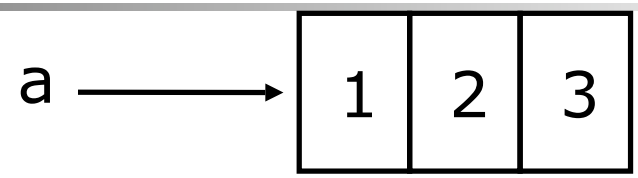
- Pychecker -- checks for errors without running your script
  - http://pychecker.sourceforge.net/
- Winpdb -- a Python debugger with a GUI
  - http://www.digitalpeers.com/pythondebugger

More Details

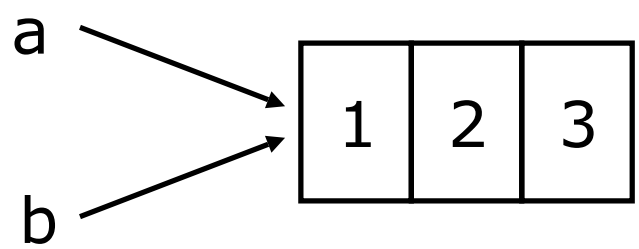
More advanced details on what we just learned.

16.0 Changing a Shared List

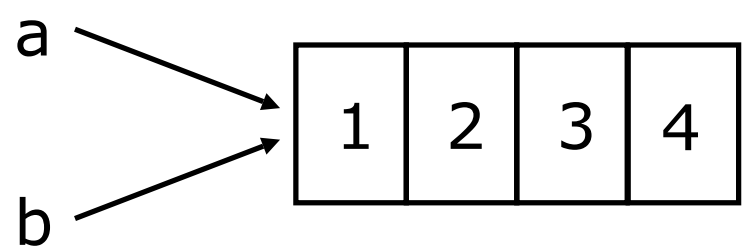
a = [1, 2, 3]



b = a



a.append(4)



17.a Functions

- Arbitrary argument lists
  - Accept any number of arguments. No keywords

```
def myFcn(*arg):
    # arg[0] ==> the first argument
    # arg[1] ==> the second argument
    # etc…

    e.g. myFcn(1, 'abc', ['blue', 'red'])
         myFcn(1, 'abc')
```

17.a Functions (cont'd)

- Arbitrary argument lists
  - Accept any number of arguments with keywords

```
def myFcn(*arg, **kw):
    # arg[0] ==> first non-keyword arg
    # arg[1] ==> second non-keyword arg
    # etc.
    # kw ==> a dict of all keyword arguments in arbitrary order

    e.g myFcn(1, 'abc', colours= ['blue', 'red'])
          myFcn(colours= ['blue', 'red'])
```

- All keyword args MUST follow the non-keyword args

18.0 Modules

- Dynamic importing

```
moduleName="sys"
mod = __import__(moduleName)
print "Python version "+mod.version
```

19.0 Command-line Args

- **`sys.argv`** -- a list of all arguments
  - `sys.argv[0]` is always the name of the script
  - `sys.argv[1]` is the first argument
- **`optparse`** -- a more powerful way of parsing arguments

19.0 Command-line Args (cont'd)

```
parser = optparse.OptionParser()
parser.add_option("-f", "--file”,
     dest="filename",
     help=”Specify the name of the file")
parser.add_option("-q", "--quiet",
     action="store_false",
     dest=”quiet", default=True,
     help="don't print status messages”)
(options, args) = parser.parse_args()
```

- -h and --help are automatically added.

20.0 Class Inheritance

- Used when you want to override some things in a class.

```
Class Shape:
   def __init__( self ):
       self.x = 0
       self.y = 0
   def move( self, x, y ):
       #change x and y

Class Circle( Shape ):
   def __init__( self ):
       Shape.__init__(self)
       self.radius = 1
```

TIME FOR QUESTIONS