CS 683 Emerging Technologies
Fall Semester, 2005
Doc 2 Python Control & Functions
Contents

References

Python Tutorial, Guido van Rossum,
http://www.python.org/doc/current/tut/tut.html

Python Reference Manual, Guido van Rossum,
http://docs.python.org/ref/ref.html

Python Library Reference, Guido van Rossum,
http://docs.python.org/lib/lib.html

Learning Python, Lutz & Ascher, O'Reilly, 1999

Reading Assignment

Sept 6. Chapters 1-4 of the Python Tutorial

Sept 8. Chapters 5-8 of the Python Tutorial

Sept 13. Chapters 9-11 of the Python Tutorial

Control Statements

- if
- while
- for

Basic Structure

```
HeaderLine:
   block
```

Block are indicated by indentation

Indentation indicated by

- Space
- Tab

Indentation **always** indicates a block

Program does not compile

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

print "Good Start"
  print "This is a compile error"
```

if

```
if <test>:
   <if block>
elif <test2>:                          #optional
   <elif block>
else:                                  #optional
   <else block>
```

Sample

```
for x in [2, 1, 0]:
   print 'x is ' , x
   if x:
      y = 2
      if y==x:
         print 'block2'
         print 'more block 2'
      print 'block1'
   print 'block0'
```

Output

```
x is  2
block2
more block 2
block1
block0
x is  1
block1
block0
x is  0
block0
```

while

```
while <test>:
   <while block>
else:                                   #optional
   <else block>
```

else is run if didn't exit from loop with a break

Example

```
x = 'cat'
while x:
   print x
   x = x[1:]
print 'The end'
```

Output

```
cat
at
t
The end
```

break, continue, pass

- break
  Jump out of closest enclosing loop
- continue
  Jump to top of the closest enclosing loop
- pass
  Does nothing, empty statement

Example

```
primeCandidate = someInteger
possibleFactor = primeCandidate /2
while possibleFactor > 1:
   if primeCandidate % possibleFactor == 0:
      print primeCandidate, 'has factor', possibleFactor
      break
   possibleFactor = possibleFactor -1
else:
   print primeCandidate, 'is prime'
```

for

```
for <target> in <object>:
   <for block>
else:                                    #optional
   <else block>
```

else is run only if break was not run in the for block

```
for x in [1, 3, 5, 7]:
   print x
```

What about for (int k =0;k< 10;k++){ }?

Does not exist in Python - use range()

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(5, 10, 3)          #3 is the increment
[5, 8]
```

Examples

```
for k in range(10):
   print k,      #prints 1  2  3  4  5  6  7  8  9

for k in range(1, 20, 2):
   print k       #prints odd number less than 20

list = ['cat', 'rat', 'bat']
for k in range(len(list)):
   print k, list[k],
```

Functions
Defining a Function

```
def fibonacci(n):
   result = []
   a, b = 0, 1
   while b < n:
      result.append(b)
      a, b = b, a+b
   return result

print fibonacci(20)
```

A function does not need an explicit return

```
def noReturn():
   pass
print noReturn()          #prints None
```

Multiple Return (Sort of)

```
def twoReturns():
   x = 2
   y = 3
   return x , y

a , b = twoReturns()
print 'a=', a, 'b=', b,
c = twoReturns()
print 'c=', c
```

Output

```
a= 2 b= 3 c= (2, 3)
```

Scope of Names

What does this print?

```
x = 10
def whichValue(x)
   print x

whichValue(5)
```

Local Variables to a Function

- Each call to a function creates a new local scope
- Arguments to the function are local
- Assigned names are local, unless declared global

```
x = 10
def printGlobal():
   print x

printGlobal()     # prints 10
x = 5
printGlobal()     # prints 5
```

Local Example

```
x = 10
def printLocal():
   x = 5       #Assignment make a local x
   print x

printLocal()    # prints 5
```

Runtime Error with Local

Local x accessed before assigned a value

```
x = 10
def printGlobal():
   print x        #runtime error here
   x = 5          #Still makes a local x
```

Global Declaration Example

```
x = 10
def globalDeclaration():
   global x
   x = 5

globalDeclaration()
print x                    # prints 5
```

Recursive Functions

```
def factorial(x):
   if x == 1:
      return 1
   else:
      return x * factorial(x - 1)

print factorial(4)
```

Parameter Passing

Python variables are references (pointers)

Parameters are passed by value

But passing pointers by value at times is like pass by reference

```
def passingParameters(x, y):
   x = 2
   y[0] = 2

a = 1
b = [1]
passingParameters(a, b)
print 'a=', a, 'b=', b
```

Output

```
a= 1 b= [2]
```

How does this work?

Default Parameter Values

```
def defaultValues(x, y=10):
   return x + y

defaultValues(2,3)        #returns 5
defaultValues(2)          #returns 12
```

Default value computed when function is defined

```
ouch = 1
def tricky(x = ouch ):
   print x

tricky()
ouch = 2
tricky()
```

Output

```
1
1
```

Positional Parameter Passing

```
def concat(x, y, z):
   return x + y + z

concat('a', 'b', 'c')              # 'abc'
concat('a', z='b', y='c')          #'acb'
concat('a', y='c',z='b')           #'acb'
concat(y='a', x='c',z='b')         #'cab'
```

Variable Arguments

Positional Arguments as tuple

```
def sum(*x):    #x tuple of positional arguments
   sum = 0
   for k in x:
      sum = sum + k
   return sum

def many(*x):
   print x

sum(1,2,3)                  #6
sum(1)                      #1
many(1, 'cat', 3)           #prints (1, 'cat', 3)
```

Keyword Arguments as Dictionary

```
def manyKeys(**x):
   for k in x.keys():
      print k, '=', x[k]

manyKeys(x='cat', a=5, foo=3.2)
```

Output

```
a = 5
x = cat
foo = 3.2
```

Using them all

In one function definition on can use:

- Default values for parameters
- Variable parameters

```
def tooMuch(a,b, c=1, *tuple, **dictionary ):
   print a, b, c, tuple, dictionary

tooMuch(1,2,3, 4, 5)
tooMuch(1,2)
tooMuch(b=1, a=2, d=3, e=5)
```

Rules in function definition

- Parameters with default values must follow those without default values
- *parameter must follow all explicit parameters
- **parameter must be last

Rules in the calling code

- Keyword arguments must appear after all nonkey-word arguments
- A parameter cannot have multiple matches

```
tooMuch(1,a=5)   #runtime error
```

Functions are Objects

```
def log(message):
   print message

tryThis = log
tryThis('cat')

def runFunction(func, arg):
   func(arg)

runFunction(log, 'this is a test')

def increase(x ):
   return x + 1

many = [log, increase]
print many[1](2)
```

Def is a Statement

```
import sys, string

input = sys.stdin.readline()
x = string.atoi(input[0])

if x < 0:
   def transform(y):
      return y - 1
else:
   def transform(y):
      return y + 1

print transform(0)
```

prints -1 or 1 depending on the value of x

Lambda Expressions

lambda arg1, arg2, ... , argn: expression

```
test = lambda x, y: x + y

print test(2, 3)                    #prints 5

three = [lambda x: x**2, lambda x: x**3,
            lambda x: x**4]

for function in three:
   print function(2),               #prints 4 8 16

noArg = lambda : 'cat'              #why do this?
print noArg()                       #prints 'cat'
```

Built-in Functions on Functions

map

map(function, list, ...)

Apply function to every item of list and return a list of the results.

```
def increase(x):
   return x +1

def add(x, y):
   return x + y

print map(increase, [1,2,3])        #prints [2, 3, 4]
print map(add, [1,2,3], [5, 6, 7]) #prints [6,8,10]
```

reduce

reduce(function, sequence[, initializer])

Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value

```
def add(x, y):
  return x + y

print reduce(add, [1,2,3])     #prints 6
```

filter

filter(function, list)

Construct a list from those elements of list for which function returns true.

```
def add(x, y):
  return x + y

print reduce(add, [1,2,3,4,5])        #prints [2, 4]
```