
CS 696 - Introduction to Grid Computing:

Lecture #3

Mary Thomas
San Diego State

Thursday, 25-Jan-07

Basics

- Mailing List: please sign up
 - <http://scilists.sdsu.edu/mailman/listinfo.cgi/cs696-grid>
- References:
 - Python Tutorial & Docs, Guido van Rossum,
 - <http://www.python.org/doc/current/tut/tut.html>
 - Learning Python, Lutz & Ascher, O'Reilly, 1999
 - Web Programming in Python, by Thiruvathukal, et al.
 - Out of print, but we will get electronic copy soon.

Assignments

- Reading Assignment:
 - Jan 30: Chapters 5-8 of the GVR Python Tutorial
 - Feb1: Chapters 9-11 of the GVR Python Tutorial
- Homework #1: Due Feb 6th,
 - see course website

Control Statements

Basic Structure

HeaderLine:
block

```
if
while
for
```

Block are indicated by indentation

Space

Tab

Indentation always indicates a block

Following does not compile

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

Importing modules (quick lesson)

```
#Fibonacci numbers module
#return Fibonacci series up to n
def fib(n):
    a, b = 0, 1
    while b < n :
        print b,
        a, b = b, a+b

def fib2(n):
    result = []
    Print 'fib2'
    a, b = 0, 1
    while b < n :
        result.append(b)
        a, b = b, a+b
    return result
```

```
[gidget:% chmod 755 fibo.py
[gidget:% ls fibo.py
-rwxr-xr-x ... fibo.py
```

```
>>> import fibo
>>> fibo.fin(20)
1 1 2 3 5 8 13
>>> fibo.fib2(50)
[1, 1, 2, 3, 5, 8, 13, 21, 34]

# edit changes fibo.py
>>> reload(fibo)
>>> fibo.fib2(50)
fib2
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

if

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

```
>>> x=4  
>>> if(x == 5): print 'y'  
...  
>>> x=5  
>>> if(x == 5): print 'y'  
...  
y
```

SAMPLE:

```
def iftst1():  
    for x in [2, 1, 0]:  
        print 'b0: x is ', x  
        if x:  
            y = 2  
            if y==x:  
                print '\t\tblock2'  
                print '\t\tmore block 2'  
            print '\tblock1'  
        print 'block0'
```

OUTPUT:

```
b0: x is 2  
        block2  
        more block 2  
        block1  
block0  
b0: x is 1  
        block1  
block0  
b0: 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:]  
... else:  
...     print 'else'  
...     print 'the end'  
...  
cat  
at  
t  
else  
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

```
>>> while x<5:  
...     x=x+1  
...     print x
```

```
...  
1  
2  
3  
4  
5
```

```
>>> while x<5:  
...     x=x+1  
...     if x == 3:  
...         break  
...     print x
```

```
...  
>>> x=0  
>>> while x<5:  
...     x=x+1  
...     if x == 3:  
...         break  
...     print x
```

```
...  
1  
2
```

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
```

#also can be written in two lines:

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

Range

A list with successive integers

```
range(upto)
range(start, upto)
range(start, upto, increment)
```

```
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 < than 20

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

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

Functions

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

```
def doesNothing():  
    pass
```

```
print fibonacci(20)  
print doesNothing()
```

Output

```
[1, 1, 2, 3, 5, 8, 13]  
None
```

Can define default arg values

```
def f(a=5):
```

or use key-value pairs

```
Def f( action='go' )
```

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
```

```
>>> def ll():  
...     x=55  
...     y=88  
...     return [x,y]  
...  
>>> ll()  
[55, 88]
```

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
```

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

>>> printGlobal()
8
>>> printLocal()
5
>>> printGlobal()
8
>>>
```

Runtime Error

"local variable 'x' referenced before assignment"

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

```
printGlobal()
>>> printGlobal()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in printGlobal
UnboundLocalError: local variable 'x' referenced before assignment
```

Global Declaration Example

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

```
globalDeclaration()
print x          # prints 5
```

Seems like something to avoid

Recursive Functions

```
def factorial(x):  
    if x == 1:  
        return 1  
    else:  
        return x * factorial(x - 1)  
  
print factorial(4)
```

Parameters are passed by value

Python variables are references (pointers)

Passing pointers by value 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]
```

Default Parameter Values

```
def defaultValues(x, y=10):  
    return x + y  
  
defaultValues(2,3) #returns 5  
defaultValues(2)  #returns 12
```

```
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

```
# *x = tuple of positional arguments
def sum(*x):
    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
```

Dictionaries are

- mutable - you can add/del key-values pairs
- associative - give a key you can find the value.

The key can be any datatype

Using them All

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

```
tooMuch(1,2,3, 4, 5)
```

Output

```
1 2 3 (4, 5) {}
```

```
tooMuch(1,2)
```

Output

```
1 2 1 () {}
```

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

Output

```
2 1 1 () {'e': 5, 'd': 3}
```

Rules

Function definition

Parameters with default values must follow those without default values

*parameter must follow all explicit parameters

**parameter must be last

Calling code

Keyword arguments must appear after all nonkeyword arguments

A parameter cannot have multiple matches

Function References

```
def log(message):  
    print message  
tryThis = log  
tryThis('cat')
```

Output
cat

```
def runFunction(func, arg):  
    func(arg)  
runFunction(log, 'this is a test')
```

this is a test

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

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

3

Def is a Statement

```
import sys, string

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

if x < 5:
    def transform(y):
        return y - 1
else:
    def transform(y):
        return y + 1

print transform(0)
```

lambda - Nameless functions

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

```
test = lambda x, y:  
x + y  
print test(2, 3)
```

Output

5

```
noArg = lambda :  
'cat'  
print noArg()
```

Output

'cat'

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

```
for function in three:  
    print function(2),
```

Output

4 8 16

lambdas remember context

```
x = 5
test = lambda: x
print test()

x = 7
print test()

def localX(func):
    x = 1
    print func()

localX(test)
```

Output

```
5
7
7
```

```
x = 1
def localLambda():
    x = 10
    return lambda: x

x=3
newFunction = localLambda()
x = 4

print newFunction()
```

Output

```
10
```

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])
```

Output

```
[2, 3, 4]
```

```
print map(add, [1,2,3], [5, 6, 7])
```

```
print map(lambda x, y:x+y, [1,2,3], [5, 6, 7])
```

Output

```
[6,8,10]
```

```
[6,8,10]
```

reduce(function, sequence[, initializer])

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

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

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

Output

10

filter(function, list)

Return elements of list for which function returns true

```
def isEven(x):  
    return x % 2 == 0
```

```
print filter(isEven, [1,2,3,4,5])
```

Output

```
[2, 4]
```