CS 683 Emerging Technologies
Fall Semester, 2006
Doc 2 Python - Control Structures
Aug 30, 2006

Reading Assignment

Aug 31. Chapters 1-4 of the Python Tutorial
Sept 5. Chapters 5-8 of the Python Tutorial
Sept 7. Chapters 9-11 of the Python Tutorial

Quiz & Assignment

First Quiz - Sept 12

Assignment 1 due - Sept 12

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

Control Statements

if
while
for

Basic Structure

HeaderLine:
    block

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"
```

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:]
else:
    print 'else'
print 'The end'
```

Output

```
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

```
x = 0
while x < 5:
    x = x + 1
    if x == 3:
        continue
    print x
```

Output

1

2

4

5

for

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

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

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

Range

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

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

```
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

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

def noReturn():
    pass
```

```
print fibonacci(20)
print noReturn()
```

Output

```
[1, 1, 2, 3, 5, 8, 13]
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
```

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

printLocal()        # prints 5
```

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()
```

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
```

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')

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)
```

Output

```
cat
this is a test
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]
```