CSC-001: Introduction to Computer Programming

2014-15-Spring Semester

Lecture #20, April 18, 2015

Recap

- Exhaustive search
- Bisection search, Binary search
- Successive approximation
  - Newton Raphson Method
- Function
  - Keyword Arguments, Default Parameters
- Scope of variables
  - Static or lexical scoping

Recap: Function Definition

def *name-of-function (list of parameters)*

body of function

Example:

```
def max (x, y) :
    if x > y :
        return x
    else :
        return y
```

Recap: Keyword Arguments

```
def printName (firstName, lastName, reverse) :
    if reverse :
        print lastName + ‘, ‘ + firstName
    else :
        print firstName, lastName
```

- some arguments can be called in different order, e.g.,
  - printName (lastName = ‘Sanghi’, firstName = ‘Dheeraj’, reverse=False)
- Keyword arguments can appear in any order, but they can not be followed by non-keyword arguments

Recap: Default Parameter

```
def printName (firstName, lastName, reverse = False) :
    if reverse :
        print lastName + ‘, ‘ + firstName
    else :
        print firstName, lastName
```

- We can call, printName (‘Dheeraj’, ‘Sanghi’)

Recap: Nested Scope

- Because variables are not declared, scoping can be confusing in the beginning
- Static or Lexical scoping
  - depends not on the position of a variable within a function, but its existence
- All variables are put on a stack with each function call
- They are removed from the stack after the call returns
- Which variable is to be accessed?
  - The one placed higher on the stack

Recap: Scoping

```
def  f ( ) :
    print x

def g ( ) :
    print x
    x = 1

x = 3
f ( )
g ( )
```

Recap: Function to find any root

```
def findRoot (x, power, epsilon) :
    “””Assumes x and epsilon int or float, power an int,
          epsilon > 0 & power >= 1
        Returns float y such that y**power is within epsilon of x.
          If such a float does not exists, it returns None.”””
    if x < 0 and power % 2 == 0 :
          return None
    low = min (-1.0, x)
    high = max (1.0, x)
    ans = (low + high) / 2.0
    while abs(ans**power – x) >= epsilon :
          if ans**power < x :
                    low = ans
          else :
                    high = ans
          ans = (high + low) / 2.0
    return ans
```

docstring

Recursion

```
def factI (n) :
    “””Assumes that n is an
    int > 0, Returns n!”””
    result = 1
    while n > 1 :
        result = result * n
        n -= 1
    return result
```

```
def factR (n) :
    “””Assumes than n is an
    int > 0, Returns n!”””
    if n == 1 :
        return n
    else :
        return n * factR (n-1)
```

Checking Palindromes

```
def isPal (s) :
    “””Assumes s is a str
        Returns true is the characters in s form a palindrome;
        False otherwise.”””
    if len(s) <= 1 :
        return True
    else :
        return s[0] == s[-1] and isPal (s[1:-1])
```

- Technique is called ‘**Divide and Conquer**’

Files

```
nameHandle = open (‘names’, ‘w’)
for i in range (2) :
    person_name = raw_input (‘Enter Name: ‘)
    nameHandle.write (person_name + ‘\n’)
nameHandle.close ( )
```

Common Files functions

- open (fn, ‘w’)
- open (fn, ‘r’)
- open (fn, ‘a’)
- fh.read ( )
- fh.write (s)
- fh.close ( )

Modules

- A module is **.py** file containing Python code
- For example, a file circle.py may contain:

```
pi = 3.14
def  area (radius)
    return pi * (radius**2)
def  circumference (radius)
    return 2 * pi * radius
def sphereSurface (radius)
    return 4 * area(radius)
def sphereVolume (radius)
    return (4.0/3.0)*pi* (radius**3)
```

Using Modules

```
import circle
print circle.pi
print circle.area (3)
print circle.circumference (3)
```

Using Modules

- Less preferred way

```
from circle import *
print pi
print area (3)
print circumference (3)
```

Global Variables

global *variable-name*

- The global variable should be defined in the outermost scope of the module
- It can now be accessed in all functions of that module

- Use with care
  - makes reading of the code very difficult

Tuples

- Ordered sequence of elements
- Individual elements can be of different types
- Written as comma separated list within a parenthesis

```
t1 = ()
t2 = (1, ‘two’, 3)
print t1
print t2
```

```
()
(1, ‘two’, 3)
```

Tuples

- A single element tuple is wrriten as:

  t3 = (1,)

  - t3 = (1) would have implied integer 1.
- Like strings, tuples can be concatenated, indexed and sliced

```
t1 = (1, ‘two’, 3)
t2 = (t1, 3.25)
print t2
print (t1 + t2)
print (t1 +t2)[3]
print (t1 + t2)[2:5]
```

```
((1, ‘two’, 3), 3.25)
(1, ‘two’, 3, (1, ‘two’, 3), 3.25)
(1, ‘two’, 3)
(3, (1, ‘two’, 3), 3.25)
```

Tuples

- A for statement can be used to iterate over tuple elements
- Example:
  - Print the common divisors of 20 and 100, and then print sum of all the divisors

```
def findDivisors (n1, n2) :
 “””Assumes that n1 and n2 are positive integers. Returns a
          tuple containing all common divisors of n1 and n2”””
    divisors = ( )  #the empty tuple
    for i in range (1, min (n1, n2) + 1) :
          if n1 % i == 0 and n2 % i == 0 :
                    divisors = divisors + (i, )
    return  divisors
divisors = findDivisors (20, 100)
print divisors
total = 0
for d in divisors :
    total += d
print total
```

Multiple Assignments

- Multiple assignment statement can be used to extract individual elements from a fixed size sequence (tuple or string).
- Example:

```
x, y = (3, 4)
a, b, c = ‘xyz’
```

```
def findExtremeDivisors (n1, n2) :
 “””Assumes that n1 and n2 are positive ints. Returns a tuple
        containing the smaller divisor > 1 and the largest
        common divisor of n1 and n2”””
    divisors = ( )   #the empty tuple
    minVal, maxVal = None, None
    for i in range (2, min(n1, n2) + 1) :
        if n1 % i == 0 and n2 % i == 0 :
                if minVal == None :
                        minVal = i
                maxVal = i
    return (minVal, maxVal)

minDivisor, maxDivisor = findExtremeDivisors (100, 200)
```

Lists

- Similar to tuples
- Written with square brackets

```
Univs = [‘MIT’, ‘Stanford’, ‘Harvard’, ‘Yale’]
```

Lists

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]
L3 = L1 + L2
print ‘L3 = ‘, L3
L1.extend (L2)
print ‘L1 = ‘, L1
L1.append(L2)
print ‘L1 =‘, L1
```

```
L3 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6, [4, 5, 6]]
```

Lists methods

- L.extend (L1) – adds elements of L1 at the end of L
- L.append (e) – adds object e to the end of L
- L.count (e) – returns the number of times object e occurs in L
- L.insert (i, e) – inserts object e into L at index i
- L.remove (e) – deletes the first occurrence of object e from L
- L.index (e) – returns the index of the first occurrence of e in L
- L.pop (i) – removes and returns the item at index i in L
- L.sort ( ) – sorts the elements of L in ascending order
- L.reverse( ) – reverses the order of elements in L

Strings, Tuples and Lists

- All are types of sequences
- We can do the following operations on all of them:

  seq[i] – returns the ith element of the sequence

  len(seq) – returns the length of the sequence

  seq1 + seq2 – returns the concatenation of two sequences

  n * seq – returns a sequence which repeats seq n times

  seq[start:end] – returns a slice of the sequence

  e in seq – returns True if e is contained in seq, else False

  e not in seq – returns the opposite of above

  for e in seq – iterates over the elements of the sequence

Strings, Tuples, and Lists

- Differences between them are:
  - strings have the basic type as characters, while others can have any type
  - lists are mutable, while others are not

List Comprehension

- A concise way to apply an operation to the values in a sequence.
- Creates a new list in which each element is the result of applying a given operation to a value from a sequence.
- Example:

```
L = [x**2 for x in range (1, 7)]
print L


[1, 4, 9, 16, 25, 36]
```

List Comprehension

```
mixed = [1, 2, ‘a’, 3, 4.0]
print [x**2 for x in mixed if type (x) == int]
```

```
[1, 4, 9]
```

map function

- **map**, in its simplest form, takes two arguments, a unary function, and a list, and returns another list by applying that unary function to each member of that list.
- Example:

```
print map (fact, [1, 2, 3])
```

```
[1, 2, 6]
```

map function

- **map** can, in general, take a function of N arguments, followed by N sequences
- Example:

```
L1 = [1, 28, 36]
L2 = [2, 57, 9]
print map (min, L1, L2)
```

```
[1, 28, 9]
```