Programming and Data Structures with Python

Lecture 4, 24 December 2020

Control flow: function definitions, assignment statements, if-else, for, while

```
def function1(...):
    stmt1
    stmt2
    return(...)

def function2(...):  # equivalent of function2 = ....
    stmt1
    stmt2
    ...
    return

# Main program
Statement 1
Statement 2   #refer to function1, function2 ...
...
Statement n
```

Functions need to be defined before they are used.

Illustrate some examples

Factors of a positive integer

In [1]:

```
def factors(n):   # Check 1,2,...,n
    flist = []    # flist is the list of factors
    for f in range(1,n+1):   # range(i,j) generates i, i+1, ... , j-1
        if (n%f == 0):
            flist = flist + [f]  # flist is the list of factors
            # flist.append(f)
    return(flist)
```

In [2]:

```
factors(17)
```

Out[2]:

```
[1, 17]
```

Comparision e1 < e2, e1 != e2 etc results in True/False (Boolean value)

Assign this outcome as a Boolean value to another name

```
outcome = (x+y == z+w)
```

In [3]:

```
def isprime(n):
    primecheck = (factors(n) == [1,n])
    return(primecheck)  # Returning a Boolean value
```

In [4]:

```
def isprime(n):
    return(factors(n) == [1,n])  # Returning a Boolean expression direct
```

In [5]:

```
def add3(a,b,c):
    #x = a+b+c
    #return(x)
    return(a+b+c)
```

In [6]:

```
isprime(18)
```

Out[6]:

```
False
```

In [7]:

```
factors(18)
```

Out[7]:

```
[1, 2, 3, 6, 9, 18]
```

In [8]:

```
isprime(2)
```

Out[8]:

```
True
```

In [9]:

```
factors(1) == [1,1]
```

Out[9]:

```
False
```

List of all primes from 1 to 100

In [10]:

```
primes100 = []
for n in range(1,101):
    if isprime(n):
        primes100 = primes100 + [n]
```

In [11]:

```
primes100w = []
i = 1
while (i <= 100):
    if isprime(i):
        primes100w = primes100w + [i]
    i = i+1
```

In [12]:

```
def primesupto(n):
    primelist = []
    for i in range(1,n+1):
        if isprime(i):
            primelist = primelist + [i]
    return(primelist)
```

In [13]:

```
primesupto(50)
```

Out[13]:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

First 100 primes = primesupto(100th prime), but what is the 100th prime?

In [14]:

```
def nprimes(n):
    i = 1
    primelist = []
    while (len(primelist) < n):   # < and not <=
        if isprime(i):
            primelist = primelist + [i]
        i = i+1
    return(primelist)
```

In [15]:

```
nprimes(1)
```

Out[15]:

```
[2]
```

Add up the numbers in a list of numbers

In [16]:

```
def sumlist(l):
    sum = 0
    for x in l:
        sum = sum + x
    return(sum)
```

In [17]:

```
def sumlistwhile(l):
    pos = 0
    sum = 0
    while (pos < len(l)):
        sum = sum + l[pos]
        pos = pos +1
    return(sum)
```

Two natural versions of f

```
for p in range(i,j):
    ...

for x in l:
    ...
```

Instead, using while

```
p = i
while (p < j):
    ...
    p = p + 1

p = 0
while (p < len(l)):
    x = l[p]
    ....
    p = p + 1
```

More about if-else

```
if condition:
    ...
else:
    ...
```

sgn(x) "sign" of x

- +1 if x is positive
- 0 if x is 0
- -1 if x is negative

abs(x) = x * sgn(x)

In [18]:

```
def sgn(x):
    if x > 0:
        return(1)
    else:
        if x == 0:   # Nested if
            return(0)
        else:
            return(-1)
```

k-way choice is implemented as a nested sequence of ifs

- Readability : not clear this is a k-way choice
- Insistence on indentation pushes code to the right

Python's solution: **elif**

In [19]:

```
def sgn(x):
    if x > 0:
        return(1)
    elif x == 0:    # Implicitly nested
        return(0)
    elif x < 0:
        return(-1)
```

In [20]:

```
sgn(-17)
```

Out[20]:

```
-1
```

True and False

- Other values can also be interpreted as True / False
- Numeric 0 is interpreted as False
- Empty list [] is interpreted as False
- Anything that is not intepreted as False is True

In [21]:

```
x = [3,4]
if (x):  # Same as if (x == True):  Similarly if(not(x)): is same as if
    y = x
else:
    y = 100000000
```

In [22]:

```
y
```

Out[22]:

```
[3, 4]
```

Slice of list

- sublist from position i to position j
- l[i:j] is [l[i],l[i+1],...,l[j-1]]
- If j <= i, result is empty

In [23]:

```
l = [0,1,2,3,4,5,6,7,8,9]
```

In [24]:

```
l[3:6]
```

Out[24]:

```
[3, 4, 5]
```

In [25]:

```
l[3:3]
```

Out[25]:

```
[]
```

In [26]:

```
l[3:4]
```

Out[26]:

```
[3]
```

In [27]:

```
l[3]
```

Out[27]:

```
3
```

Mutable and immutable values

In [28]:

```
x = 7
y = x
x = x+1
# What is the value of y after this?
```

In [29]:

```
(x,y)
```

Out[29]:

```
(8, 7)
```

In [30]:

```
l1 = [1,2,3]
l2 = l1
l1[0] = 4  # Reassign value at position 0 to 4
# What are the values of l1 and l2?
```

In [31]:

```
(l1,l2)
```

Out[31]:

```
([4, 2, 3], [4, 2, 3])
```

When I assign y = x, the value is copied - *immutable value*

When I assign l2 = l1, both names point to the same value - *mutable value*

So how do I make a copy of l1 in l2 that is not the same value?

Slices can be used for this

- If j <= i, then the resulting slice is empty
- If i is not provided, start of slice is implicitly 0
- If j is not provided, end of slice is implictly len(l)

In [32]:

```
l = [0,1,2,3,4,5,6,7,8,9]
```

In [33]:

```
l[:]
```

Out[33]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [34]:

```
l1 = [1,2,3]
l2 = l1[:]
l1[0] = 4  # Reassign value at position 0 to 4
# What are the values of l1 and l2?
```

In [35]:

```
(l1,l2)
```

Out[35]:

```
([4, 2, 3], [1, 2, 3])
```

Nested lists

In [36]:

```
m = [ [10,11], [12,13]]
```

In [37]:

```
m[0],m[1]
```

Out[37]:

```
([10, 11], [12, 13])
```

In [38]:

```
m[0][0]
```

Out[38]:

```
10
```

In [39]:

```
m[1][0]
```

Out[39]:

```
12
```

Pitfalls with mutability and multiple references to same list value

In [40]:

```
zerolist = [0,0]
matrix = [zerolist,zerolist]
```

In [41]:

```
matrix
```

Out[41]:

```
[[0, 0], [0, 0]]
```

In [42]:

```
matrix[0][0] = 7
```

In [43]:

```
matrix
```

Out[43]:

```
[[7, 0], [7, 0]]
```

In [44]:

```
zerolist
```

Out[44]:

```
[7, 0]
```

Difference between l.append(x) and l = l + [x]

In [45]:

```
l1 = [1,2,3]
l2 = l1
l1.append(4)
```

In [46]:

```
(l1,l2)
```

Out[46]:

```
([1, 2, 3, 4], [1, 2, 3, 4])
```

In [47]:

```
l3 = [1,2,3]
l4 = l3
l3 = l3+[4]
```

In [48]:

```
(l3,l4)
```

Out[48]:

```
([1, 2, 3, 4], [1, 2, 3])
```