Lecture 05, 22 August 2023

Control flow

- A Python program is a sequence of statements
  - Normal execution is sequential, top to bottom
- Most basic type of statement is **assignment**
  - `name = value`, where `value` can be an expression
- To perform interesting computations we need to control the flow
  - `if`, `for`, `while`

Functions

- Templates for re-usable code
- Instantiate with different arguments
- A function must be defined before it is used (just like any other name)
  - Typically, define your functions first, then the code that calls them

Updating lists

- Combine two lists into one - *concatenation* - `l1 + l2`
- Append a value to a list - `l.append(v)`
- `l.append(v)` is same as `l = l + [v]`

Example 1: Find the first position where `v` occurs in `l`

- If `v` is in `l`, first position lies between `0` to `len(l)-1`
- Return `-1` if no `v` in `l`

In [1]:
```
def locatepos(v,l):
    pos = 0
    for x in l:
        if x == v:
            return(pos)
        pos = pos+1
    return(-1) ## Could return(False), but not a good idea to have different types
```

In [2]:
```
l3 = [1,2,3,4,5,6,7,8,9,10]
```

In [3]:
```
locatepos(8,l3), locatepos(12,l3)
```

Out[3]:
```
(7, -1)
```

- We used a name `pos` to keep track of our position in the list and manually updated it with each iteration
- What we should be able to do instead is:
  - Set up a list `[0,1,2,...,len(l)-1]`
  - Run through these values and check if `l[i] == v`
  - Report the first such `i`

`range()`

- `range()` function generates a sequence of numbers

In [4]:
```
range(7) # generates the sequence 0,1,2,...,6
```

Out[4]:
```
range(0, 7)
```

In [5]:
```
for i in range(7):
    print(i)
```

```
0
1
2
3
4
5
6
```

- `range()` produces an sequence over which you can iterate
- output is **not** a list, but you can index into it

In [8]:
```
l = range(7)
```

In [9]:
```
type(l)
```

Out[9]:
```
range
```

In [10]:
```
l[2]
```
Out[10]: 2

- Use `list()` as a function to convert a sequence to a list

In [11]:
```
l = list(range(7))
```
In [12]:
```
l
```
Out[12]: [0, 1, 2, 3, 4, 5, 6]

- `list()` will complain if its argument is not a valid sequence

In [13]:
```
l = list(6)
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [13], line 1
----> 1 l = list(6)

TypeError: 'int' object is not iterable
```
In [14]:
```
def locatepos2(v,l):
    # pos = 0
    for pos in range(len(l)):
        if l[pos] == v:
            return(pos)
    return(-1) ## Could return(False), but not a good idea to have different types
```
In [15]:
```
locatepos2(8,l3), locatepos2(12,l3)
```
Out[15]: (7, -1)

More about `range()`

- `range(a,b)` - generates a, a+1, ..., b-1
- `range(a,b,d)` - generates a, a+d, a+2d, ... stop before it crosses b
- `range()` implicitly generates a sequence, so to "see" it, wrap it in `list()`

In [16]:
```
list(range(3,13))
```
Out[16]: [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

In [17]:
```
list(range(3,13,5))
```
Out[17]: [3, 8]

In [18]:
```
list(range(3,13,3))
```
Out[18]: [3, 6, 9, 12]

- Use negative step to count backwards
- Understand stopping criterian when counting backwards

In [19]:
```
list(range(10,5,-1))
```
Out[19]: [10, 9, 8, 7, 6]

In [20]:
```
len(l3)
```
Out[20]: 10

In [21]:
```
list(range(len(l3)-1,-1,-1))
```
Out[21]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

In [22]:
```
list(range(len(l3)-1,-1,-3))
```
Out[22]: [9, 6, 3, 0]

- `range()` requires `int` arguments

In [23]:
```
list(range(1.3,2.7,1))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [23], line 1
----> 1 list(range(1.3,2.7,1))

TypeError: 'float' object cannot be interpreted as an integer
```

`while` loop

- `for` loops iterate over a sequence that is known in advance
- sometimes, we need to iterate till a desired condition is satisfied

Example

- generating lists of prime numbers
- start with a definition of `isprime` based on the list of factors of a number

In [25]:
```
def factors(n):
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)
```

In [26]:
```
factors(10)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [26], line 1
----> 1 factors(10)

Cell In [25], line 4, in factors(n)
      2 for i in range(1,n+1):
      3     if n%i == 0:
----> 4         factorlist.append(i)
      5 return(factorlist)

NameError: name 'factorlist' is not defined
```

- `factorlist.append()` is like `factorlist = factorlist + [i]`
- `factorlist` needs to be initialized to `[]`, else Python does not know it is a list value

In [27]:
```
def factors(n):
    factorlist = []
    for i in range(1,n+1):
        if n%i == 0:
            factorlist.append(i)
    return(factorlist)
```

In [28]:
```
factors(10)
```

Out[28]:
```
[1, 2, 5, 10]
```

- For a number to be prime, `factors(n)` should be `[1,n]`
- Note: 1 is correctly reported to not be a prime since `[1]` is not the same as `[1,1]`
- Can also check `len(factors(n)) == 2`

In [29]:
```
def isprime(n):
    return(factors(n) == [1,n])
```

In [30]:
```
isprime(1),isprime(2),isprime(4)
```

Out[30]:
```
(False, True, False)
```

Listing out prime numbers

- Find all primes below m - primesupto(m)
- Can use a for - need to test numbers from 1 to m

In [31]:
```
def primesupto(m):
    primelist = []
    for i in range(1,m+1):
        if isprime(i):
            primelist.append(i)
    return(primelist)
```

In [33]:
```
primesupto(50)
```

Out[33]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

In [34]:
```
primesupto(10000)[-1]
```
Out[34]:
```
9973
```

Listing out prime numbers ...

- list out the first m primes
- do not know in advance how many values to run through, cannot use for
- while loop - terminates based on a suitable condition - like a repeated if

In [35]:
```
def firstmprimes(m):
    count = 0
    primelist = []
    i = 1
    while(count < m):
        if isprime(i):
            primelist.append(i)
            count = count + 1
        i = i + 1
    return(primelist)
```
In [37]:
```
firstmprimes(20)
```
Out[37]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```
In [38]:
```
len(firstmprimes(20))
```
Out[38]:
```
20
```

- need not keep track of `numprimes` separately since this is available as `len(plist)`

In [39]:
```
def firstmprimes2(m):
    # count = 0 -- always len(primelist)
    primelist = []
    i = 1
    while(len(primelist) < m):
        if isprime(i):
            primelist.append(i)
        i = i + 1
    return(primelist)
```
In [40]:
```
firstmprimes2(15)
```
Out[40]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

for vs while

- Use for when you know the upper bound of the iteration in advance
- Use while when this is not known in advance
- `for` will always terminate if you do not modify the sequence over which the iteration runs
- `while` may not terminate - need to ensure the condition eventually becomes false - "making progress"

Warning: Do not modify the list being iterated on by `for`

```
l = [1,2,3,4,5,6,7,8]
  for x in l:
    if x%2 == 0:
      l.append(x)
```

- The list `l` keeps growing, so the iteration never terminates
- In general, if you update the sequence while it is being iterated over, the outcome is unpredictable

Iterating over on lists

- Compute sum and average (mean) of a list
- Compute values above the mean
  - Requires two passes over the list
- `aboveaverage` is an example of *filtering* a list
  - Extracting a sublist satisfying a certain property

Many useful functions on lists are built-in to Python

In [41]:
```
l = [1,2,3,4,5,6,7,8]
```
In [42]:
```
len(l), sum(l), max(l), min(l)
```
Out[42]:
```
(8, 36, 8, 1)
```

Nested loops

- find all elements common to `l1` and `l2`

- for each `x` in `l1`, check if `x` is in `l2`
- for each `y` in `l2`, check if `x == y`

In [43]:
```
def findcommon(l1,l2):
    commonlist = []
    for x in l1:
        for y in l2:
            if x == y:
                commonlist.append(x)
    return(commonlist)
```

In [44]:
```
l1 = [1,2,3,4]
l2 = [3,4,5,6]
findcommon(l1,l2)
```

Out[44]:
```
[3, 4]
```

- Our function will list repetitions multiple times

In [45]:
```
l1 = [1,2,3,4]
l2 = [3,4,5,3]
findcommon(l1,l2)
```

Out[45]:
```
[3, 3, 4]
```

- Nested loops can be expensive
- 10^8 operations take about 10 seconds in Python
- Compare the running time of the following nested loops

In [46]:
```
for i in range(1000):
    for j in range(1000):
        x = i + j
print("Done")
```

```
Done
```

In [47]:
```
for i in range(10000):
    for j in range(10000):
        x = i + j
print("Done")
```

```
Done
```