Lecture 4, 30 Sep 2021

Typical structure of Python code

- First function definitions
- Then statements of "main" program

```
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
```

`for` loop

- runs over the elements in a list (or, more generally, a sequence of values)

Example 1

- `locate(l,v)` : `bool` - return `True` if `v` appears in `l`

```
def locate(l,v):
  # for each element of l, check if it is equal to v
  for x in l:   # x will take on each value in l from l[0] to l[len(l)-1
    if x == v:
      return(True)
  # if the for "loop" ends and we exit the loop, no x in l was equal to
  return(False)

mylist = [1,7,3,5,4,5,4]
locate(mylist,5)
```

```
True
```

Example 2

- `locatepos(l,v)` : `int` - returns i if first occurrence of v in l is l[i], return -1 if not found
- Need to keep track of where we are in the list - `pos`

```
def locatepos(l,v):
  pos = 0
  for x in l:
    if x == v:
      return(pos)
    pos = pos+1   # Increment pos outside the if
  return(-1)


locatepos(mylist,5)
```

```
3
```

- We have kept track of our position in the list "manually"
- Instead, can we directly make `for` cycle through the positions 0,1,2,...,len(l)-1?
- `range()` function generates such sequences
- `for` can directly run through values produced by `range()`
- However, to display the values we need to convert it to a list by invoking the function `list()`

```
list(range(7))  # Generates 0 to 7-1, and convert to list
```

```
[0, 1, 2, 3, 4, 5, 6]
```

```
def locatepos2(l,v):
  for i in range(len(l)):  # Generating positions 0,1,2,...,len(l)-1
    if l[i] == v:
      return(i)
  return(-1)


locatepos2(mylist,5)
```

```
3
```

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()`

```
list(range(5,18,-1))
```

```
[]
```

- If d is negative, count down

- Reaching the target from above
- Again, stop just before you achieve the target

```
list(range(18,5,-5))
```

```
[18, 13, 8]
```

```
def locatepos3(l,v):
  for i in range(len(l)-1,-1,-1):  # Target is -1 to ensure I reach 0
    if l[i] == v:
      return(i)
  return(-1)
```

```
locatepos2(mylist,5), locatepos3(mylist,5)
```

```
(3, 5)
```

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

```
def factors(n):
  flist = []
  for i in range(1,n+1):   # run through 1,2,..,n
    if n%i == 0:           # if i divides n
      flist.append(i)      # flist = flist + [i], need flist to be alre
  return(flist)
```

```
factors(24)
```

```
[1, 2, 3, 4, 6, 8, 12, 24]
```

- 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`

```
def isprime(n):
  return(factors(n) == [1,n])
  # Or return(len(factors(n)) == 2)
```

```
isprime(1), factors(1)
```

```
(False, [1])
```

Listing out prime numbers

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

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

```
primesupto(10)
```

```
[2, 3, 5, 7]
```

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`

```
def·firstnprimes(m):
··numprimes·=·0
··i·=·1
··plist·=·[]
··while·numprimes·<·m:···#·instead·len(plist)·<·m
····if·isprime(i):
······plist.append(i)
······numprimes·=·numprimes+1···#·Incremented·only·when·we·find·a·prime
····i·=·i+1·····················#·Incremented·each·time·the·loop·execute
··return(plist)
```

```
firstnprimes(10)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

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

```
def firstnprimes2(m):
  i = 1
  plist = []
  while len(plist) < m:
    if isprime(i):
      plist.append(i)
```

```
    i = i+1                       # Incremented each time the loop execute
  return(plist)
```

- If number of iterations is known in advance, use `for`
- `for` will always finish
- `while` may not terminate - need to ensure the condition eventually becomes false - "making progress"

```
def firstnprimes3(m):
  i = 1
  plist = []
  while len(plist) >= 0:          # Wrong condition, always true, never te
    if isprime(i):
      plist.append(i)
    i = i+1                       # Incremented each time the loop execute
  return(plist)
```

```
firstnprimes3(5)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-22-6e1e70310790> in <module>()
----> 1 firstnprimes3(5)

 ——————————————————————————— 2 frames ———————————————————————————

<ipython-input-12-e01e1ddb4913> in factors(n)
      2   flist = []
      3   for i in range(1,n+1):   # run through 1,2,..,n
----> 4     if n%i == 0:           # if i divides n
      5       flist.append(i)      # flist = flist + [i], need flist to be already
defined
      6   return(flist)

KeyboardInterrupt:
```

SEARCH STACK OVERFLOW

Operating on lists

- Add up the numbers in a list - `sumlist`
- `for` loop accumulates the sum in a variable initialized to `0`

```
def·sumlist(l):
··sum·=·0··········#·Accumulated·value,·initially·0
··for·x·in·l:
····sum·=·sum·+·x
··return(sum)
```

```
sumlist([1,2,3,4,5]), sum([1,2,3,4,5]), sum([])
```

```
(15, 15, 0)
```

- find the maximum value in a list

- keep track of maximum seen so far, in `max`, and update each time we see a larger number
- need to initialize `max` sensibly
- if we do not know a lower bound in advance, use the first element of the list
- need to take care of the empty list

```
def maxlist(l):
  max = l[0]     # Be careful if l == []
  for x in l:
    if x > max:
      max = x
  return(max)

def maxlist2(l):
  if l == []:
    return
  max = l[0]     # Be careful if l == []
  for x in l:
    if x > max:
      max = x
  return(max)
```

- `return` with no argument returns a special value `None`

```
v = maxlist2([])

print(v)
```

```
None
```

- find `average`
- find sum and divide by length

```
def average(l):
  if l == []:
    return
  sum = 0
  for x in l:
    sum = sum + x
  return(sum/len(l))  # or return(sumlist(l)/len(l))

average(list(range(1,100,3)))
```

```
49.0
```

- find all values above the average?
- need a second loop, to compare with average computed in the first loop

```
def·aboveaverage(l):
··if·l·==·[]:
····return
··sum·=·0
··for·x·in·l:
····sum·=·sum·+·x
··avg·=·sum/len(l)
··aboveavglist·=·[]
··for·x·in·l:
····if·x·>·avg:
······aboveavglist.append(x)
··return(aboveavglist)
```

- In general, should incrementally build a solution re-using earlier function
- Use `sumlist` to define `average`
- Use `average` to define `aboveaverage`

```
def sumlist(l):
  sum = 0           # Accumulated value, initially 0
  for x in l:
    sum = sum + x
  return(sum)

def average(l):
  if l == []:
    return
  return(sumlist(l)/len(l))

def·aboveaverage(l):
··if·l·==·[]:
····return
··avg·=·average(l)
··aboveavglist·=·[]
··for·x·in·l:
····if·x·>·avg:·
······aboveavglist.append(x)
··return(aboveavglist)
```

- If we rewrite `aboveaverage` to not use `avg`, we would have to recompute `average(l)` for each `x`

```
def aboveaverage(l):
  if l == []:
    return
  aboveavglist = []
  for x in l:
    if x > average(l):        # No avg, but wasteful recomputation of a
      aboveavglist.append(x)
```

```
return(aboveavglist)
```