Lecture 04, 18 August 2022

Data type

- Numeric types `int` and `float` ; logical values `boolean` ; sequence of values `list`
- Determines what operations are allowed
  - `len(x)` does not make sense if value of `x` is not a list
- Names inherit their type from the values they currently hold

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

Conditionals -- take different paths based on the values computed so far

- Basic statement is `if`

Example 1: Compute absolute value

In [1]:
```
def absval(x): # Returns absolute value of x
  y = x
  if y < 0:    # if boolean-expression:
     y = -y    # Indented, locally uniform, not globally uniform
  return(y)
```

In [2]:
```
absval(-8),absval(9)
```

Out[2]:
```
(8, 9)
```

- Provide an alternative to execute using `else`
- The following is equivalent to the above

In [3]:
```
def absval2(x):
  if x < 0:
    y = -x
  else:
    y = x
  return(y)
```

Example 2: Check if input x lies in the range [a,b]

In [4]:
```
def inrange(a,b,x):  # Return True if a <= x <= b
  if (x >= a) and (x <= b):
    return(True)
  else:
    return(False)
```

In [5]:
```
inrange(7,10,9)
```

Out[5]:
```
True
```

In [6]:
```
type(None)
```

Out[6]:
```
NoneType
```

- `return()` exits the function, so can group useful exits up front and have a final default `return()` that is the alternative to all the useful cases.

In [7]:
```
def inrange2(a,b,x):
  if (x >= a) and (x <= b):
    return(True)    # End of the execution if the condition holds
  return(False)     # Is reached only if condition did not hold --- same effect as else
```

In [8]:
```
def ineitherrange(a,b,c,d,x): # Return True if x in [a,b] or x in [c,d]
  if (x >= a) and (x <= b):
    return(True)
  if (x >= c) and (x <= d):
    return(True)
  return(False)
```

In [9]:
```
def ineitherrange2(a,b,c,d,x): # Return True if x in [a,b] or x in [c,d]
  if ((x >= a) and (x <= b)) or ((x >= c) and (x <= d)):
    return(True)
  return(False)
```

More about lists

- concatenation of two lists: `[1,2,3] + [4,5,6] --> [1,2,3,4,5,6]`
- `append()` : appends a value: `l.append(4)` or `l+[4]`

In [10]:
```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = l1+l2
l3
```

Out[10]:
```
[1, 2, 3, 4, 5, 6]
```

In [11]:
```
l3.append(7)   # Updates l3 in place
l3
```

Out[11]:
```
[1, 2, 3, 4, 5, 6, 7]
```

Be careful

- `l.append(x)` updates `l` but returns `None`
- `l = l.append(x)` would result in `l` being set to `None`

In [12]:
```
l3 = l3 + [8]
l3
```

Out[12]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

In [13]:
```
l3 + 9
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_952637/2147644336.py in <module>
----> 1 l3 + 9

TypeError: can only concatenate list (not "int") to list
```

In [14]:
```
9 + l3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_952637/784020746.py in <module>
----> 1 9 + l3

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

In [15]:
```
[0] + l3
```

Out[15]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

In [16]:
```
l1.extend(l2)
l1
```

Out[16]:
```
[1, 2, 3, 4, 5, 6]
```

Recall the fuction `myadd3` from last time

In [17]:
```
def myadd3(a,b):
    return(a+b)
```

`myadd3(l1,l2)` will also work since `+` will be interpreted as list concatenation

In [18]:
```
myadd3([1,2,3],[4,5,6])
```

Out[18]:
```
[1, 2, 3, 4, 5, 6]
```

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`

In [19]:
```
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 v
  return(False)
```

In [20]:
```
mylist = [1,7,3,5,4,5,4]
locate(mylist,5)
```

Out[20]:
```
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`

In [21]:
```
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)
```

In [22]:
```
locatepos(mylist,11)
```

Out[22]:
```
-1
```

- 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()`

In [23]:
```
list(range(7))  # Generates 0 to 7-1, and convert to list
```

Out[23]:
```
[0, 1, 2, 3, 4, 5, 6]
```

In [24]:
```
range(7), type(range(7))
```

Out[24]:
```
(range(0, 7), range)
```

In [25]:
```
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)
```

In [26]:
```
locatepos2(mylist,5)
```

Out[26]:
```
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()`

In [27]:
```
list(range(5,18,-1))
```

Out[27]:
```
[]
```

- If `d` is negative, count down
- Reaching the target from above
- Again, stop just before you achieve the target

In [28]:
```
list(range(18,5,-5))
```

Out[28]:
```
[18, 13, 8]
```

In [29]:
```
list(range(8,8))
```

Out[29]:
```
[]
```

In [30]:
```
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)
```

In [31]:
```
locatepos2(mylist,5), locatepos3(mylist,5)
```

Out[31]:
```
(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

In [32]:
```
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 already defined
  return(flist)
```

In [33]:
```
factors(24)
```

Out[33]:
```
[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`

In [34]:
```
def isprime(n):
  return(factors(n) == [1,n])
  # Or return(len(factors(n)) == 2)
```

In [35]:
```
isprime(1), factors(1)
```

Out[35]:
```
(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`

In [36]:
```
def primesupto(m):
  plist = []
  for i in range(1,m+1):
    if isprime(i):
      plist.append(i)
  return(plist)
```

In [37]:
```
primesupto(10)
```

Out[37]:
```
[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`

In [38]:
```
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 executes
  return(plist)
```

In [39]:
```
firstnprimes(10)
```

Out[39]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

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

In [40]:
```
def firstnprimes2(m):
  i = 1
  plist = []
  while len(plist) < m:
    if isprime(i):
      plist.append(i)
    i = i+1                    # Incremented each time the loop executes
  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"