PDSP 2025, Lecture 05, 21 August 2025

Conditional statement

`if` allows conditional execution

```
if condition:
    statement 1
    ...
    statement k
else:
    statement 1'
    ...
    statement k'
```

- If `condition` evaluates to `True`, the first block is executed, otherwise the second block.
- The `else:` block is optional. If there is no `else:` block and the `condition` evaluates to `False`, execution skips over to the next statement after the `if`

- Example: Compute the absolute value of a number

In [1]:
```python
def myabs(x):  # myabs to avoid any confusion with built-in abs()
    if x < 0:
        return(-x)
    else:
        return(x)
```

In [2]:
```python
myabs(-9), myabs(7)
```

Out[2]:
```
(9, 7)
```

Multiway branching --- `elif`

Suppose we want to compute $sign(x) = \begin{cases} x < 0 & = & -1, \\ x = 0 & = & 0, \\ x > 0 & = & 1 \end{cases}$

In Python, we would have to nest `if` statements like this:

```
if x < 0:
    return(-1)
else:
    if x == 0:
        return(0):
    else:
        return(1)
```

- As we see, the indentation of the nested `if` pushes the code to the right
- With more cases, this would become worse
- Python provides `elif` to avoid this cascaded nesting

```
if x < 0:
    return(-1)
elif x == 0:
    return(0):
else:
    return(1)
```

- Can have as many `elif` blocks as you need
- `else` is still optional

In [3]:
```python
def sign(x):
    if x < 0:
        return(-1)
    elif x == 0:
        return(0)
    else:
        return(1)
```

In [4]:
```python
sign(-7)
```

Out[4]:
```
-1
```

In [5]:
```python
sign(8)
```

Out[5]:
```
1
```

In [6]:
```python
sign(0)
```

Out[6]:
```
0
```

Lists

- Sequences of values, indexed by position
- For a list with `n` values, valid positions are `0` to `n-1`
  - `len(l)` gives the length of a list
- Accessing a position beyond `len(l)-1` results in `IndexError`

In [7]:
```python
l = list(range(20,40))
```

In [8]:
```python
len(l), l[3], l[19]
```

Out[8]:
```
(20, 23, 39)
```

In [9]:
```python
l[20]
```

```
---------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call las
t)
Cell In[9], line 1
----> 1 l[20]

IndexError: list index out of range
```

- What about indices below `0`?
- Index `-j` is interpreted as `len(l)-j`
  - Useful for accessing values from the end of the list
  - Valid indices in reverse are `-1`, `-2`, ..., `-len(l)`

In [10]:
```
l[-1], l[-20]
```

Out[10]:
```
(39, 20)
```

Slices

- Recall that `nprimes(n)` computed the first `n` primes

In [11]:
```
def isprime(n):
    for j in range(2,n):
        if n % j == 0:
            return(False)
    return(True)

def nprimes(n):
    plist = []
    j = 2
    while (len(plist) < n):
        if isprime(j):
            plist.append(j)
        j = j+1
    return(plist)
```

In [12]:
```
first20primes = nprimes(20)
```

In [13]:
```
first20primes
```

Out[13]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71]
```

- What are the primes from 11 to 15?
- Need a sublist of the original list
- `l[i:j]` is the list `[l[i], l[[i+1], ..., l[j-1]]`
- Similar to
  ```
  newl = []
  for k in range(i,j):
     newl.append(k)
  ```

In [14]:
```
first20primes[11:16]
```

Out[14]: [37, 41, 43, 47, 53]

- like `range()` if the indices don't make sense, you get an empty list

In [15]:
```
first20primes[11:10]
```

Out[15]: []

- Unlike accessing `l[i]`, can give upper bound beyond the list
- `l[i:len(l)+10]` is interpreted as `l[i:len(l)]`

In [16]:
```
first20primes[11:40]
```

Out[16]: [37, 41, 43, 47, 53, 59, 61, 67, 71]

- Can omit the upper bound, defaults to `len(l)`

In [17]:
```
first20primes[15:]
```

Out[17]: [53, 59, 61, 67, 71]

- Likewise, omit the lower bound, defaults to `0`

In [18]:
```
first20primes[:10]
```

Out[18]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

- Omit both lower and upper bound to get a `full slice`
  - Full slice returns a new list that is a copy of the list
  - Significance will become clearer later

In [19]:
```
first20primes[:]
```

Out[19]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]

More about `range()`

- `range(i,j)` generates the sequence `i,i+1,...,j-1`
- `range(n)` generates the sequence `0,1,...,n-1` -- implicitly starts with `0`
- What if we want to skip over some numbers
  - All even numbers from `4` to `40`
- Optional third argument is the step size
  - `range(i,j,k)` is `i,i+k,...,i+mk` for the largest `m` such that `i+mk < j` and `i+(m+1)k >= j`

In [20]:
```
list(range(4,41,2)) # Even numbers from 4 to 40
```

Out[20]: [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 4
0]

In [21]:
```
list(range(4,40,2)) # Even numbers from 4 to 38
```

Out[21]:
```
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
```

- Can also count down -- give a negative step!

In [22]:
```
list(range(10,0,-1))
```

Out[22]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

- `range(i,j,k)` generate `i, i+k,...` so that the sequence does not *cross* `j`
- Depending on whether it is increasing or decreasing, the last value will be less than `j` or greater than `j`

Stepped slices

- Can similarly give a third argument in a slice

In [23]:
```
first20primes[2:20:3]
```

Out[23]:
```
[5, 13, 23, 37, 47, 61]
```

In [24]:
```
first20primes[19:0:-1]
```

Out[24]:
```
[71, 67, 61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5,
3]
```

- Explain the following output. (Hint, what is `l[-1]` ?)

In [25]:
```
first20primes[19:-1:-1]
```

Out[25]:
```
[]
```

- Can omit upper and lower bounds but give a step
- `l[::-1]` is the entire list in reverse
  - Note that the default lower and upper bound are determined by the step

In [26]:
```
first20primes[::-1]
```

Out[26]:
```
[71, 67, 61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5,
3, 2]
```

Assigning slices

- Can assign a list to a slice

In [27]:
```
l = list(range(20,30))
```

In [28]:
```
l[3:6] = [53,54,55]
```

In [29]:
```
l
```

Out[29]:
```
[20, 21, 22, 53, 54, 55, 26, 27, 28, 29]
```

- Can contract or expand the slice when reassigning
- Indices of values to the right will change

In [30]:
```
l = list(range(20,30))
l[3:5] = [53,54,55,63,64,65]
```

In [31]:
```
l
```

Out[31]:
```
[20, 21, 22, 53, 54, 55, 63, 64, 65, 25, 26, 27, 28, 29]
```

In [32]:
```
l = list(range(20,30))
l[3:5] = []
```

In [33]:
```
l
```

Out[33]:
```
[20, 21, 22, 25, 26, 27, 28, 29]
```

Operations on lists

- Recall that `+` concatenates two lists
- Returns a new list
- Original lists are unchanged

In [34]:
```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = l1 + l2
```

In [35]:
```
l3, l1, l2
```

Out[35]:
```
([1, 2, 3, 4, 5, 6], [1, 2, 3], [4, 5, 6])
```

- A useful invariant about slices
- For any list `l`, and any integer `j`, `l == l[:j] + l[j:]`

In [36]:
```
l3[:-1]+l3[-1:]
```

Out[36]:
```
[1, 2, 3, 4, 5, 6]
```

In [37]:
```
l3[:2]+l3[2:]
```

Out[37]:
```
[1, 2, 3, 4, 5, 6]
```

In [38]:
```
l3[:9]+l3[9:]
```

Out[38]:
```
[1, 2, 3, 4, 5, 6]
```

Applying functions to lists

- `l.append(v)` is the same as `l = l+[v]`
  - Ask the list `l` to append `v` to itself
  - `l.append(v)` updates `l` in place
  - `l = l+[v]` creates a new list and reassigns the list pointed to by `l`
  - Again, we will see the significance of this later

In [39]:
```
l3.append(7)
```

In [40]:
```
l3
```

Out[40]:
```
[1, 2, 3, 4, 5, 6, 7]
```

- It is a mistake to reassign a list after an `append()`

In [41]:
```
l3 = l3.append(8)
```

In [42]:
```
l3
```

- Assignment `v = e` stores return value of `e` in `v`
- Return value of `l.append()` is empty

Other functions

- `l.insert(pos,val)` inserts `val` at position `p`
  - Similar to `l = l[:pos] + [val] + l[pos:]`
- `l.extend(newl)` extends `l` with a list of values `newl`
  - Similar to `l = l + newl`
- Like `l.append(v)`, these update the list in place, do not reassign return value

In [43]:
```
l3 = [1,2,3,4,5,6,7]
```

In [44]:
```
l3.insert(0,0)
```

In [45]:
```
l3
```

Out[45]:
```
[0, 1, 2, 3, 4, 5, 6, 7]
```

In [46]:
```
l3.extend([8,9,10])
```

In [47]:
```
l3
```

Out[47]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sorting

- `l.sort()` sorts a list in place
  - Python allows lists of mixed types
  - To sort a list, the values must be of a uniform comparable type

In [48]:
```
l = [1,15,3,7,9,2]
```

In [49]:
```
l.sort()
```

In [50]:
```
l
```

Out[50]:
```
[1, 2, 3, 7, 9, 15]
```

In [51]:
```
badl = [1,'CSK',True,7.5]
```

In [52]:
```
badl.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[52], line 1
----> 1 badl.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

In [53]:
```
blist = [True,False]
```

In [54]:
```
blist.sort()
```

In [55]:
```
blist
```

Out[55]:
```
[False, True]
```

- If you want a sorted copy of `l` without disturbing `l`, use `sorted(l)`

In [56]:
```
l = [15, 1000, 9, 7, 3, 2, 1]
```

In [57]:
```
l
```

Out[57]:
```
[15, 1000, 9, 7, 3, 2, 1]
```

In [58]:
```
sorted(l)
```

Out[58]:
```
[1, 2, 3, 7, 9, 15, 1000]
```

In [59]:
```
l
```

Out[59]:
```
[15, 1000, 9, 7, 3, 2, 1]
```

- Can store a copy of the sorted list in another list

In [60]:
```
newl = sorted(l)
```

In [61]:
```
newl, l
```
Out[61]:
```
([1, 2, 3, 7, 9, 15, 1000], [15, 1000, 9, 7, 3, 2, 1])
```

- Can we, instead, first copy `l` and sort the copy in place using `sort()` ?

In [62]:
```
newl = l
```
In [63]:
```
newl.sort()
```
In [64]:
```
newl
```
Out[64]:
```
[1, 2, 3, 7, 9, 15, 1000]
```
In [65]:
```
l
```
Out[65]:
```
[1, 2, 3, 7, 9, 15, 1000]
```

- Sorting `newl` also sorts `l`
- This does not happen with types like `int`
- We will investigate this later

In [66]:
```
y = 7
```
In [67]:
```
x = y
```
In [68]:
```
x, y
```
Out[68]:
```
(7, 7)
```
In [69]:
```
x = 17
```
In [70]:
```
x, y
```
Out[70]:
```
(17, 7)
```

- Many other built-in functions on lists
  - `l.reverse()` reverses a list
- Look up Python documentation