Lecture 2-3

Lists Part 2 and Strings

Week 2 Friday

Miles Chen, PhD

Adapted from Chapter 6 of Think Python by Allen B Downey

List content adapted from "Whirlwind Tour of Python" by Jake VanderPlas

Lists are mutable

This means that methods change the lists themselves. If the list is assigned to another name, both names refer to the exact same object.

In [1]:
```python
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
print(fam)
second = fam    # second references fam. second is not a copy of fam.
second[0] = "sister"  # we make a change to the list 'second'
print(second)
print(fam) # changing the list 'second' has changed the list 'fam'
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['sister', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['sister', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [2]:
```python
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
print(fam)
second = fam[:]  # creates a copy of the list
# second = fam.copy() # you can also create a list using the copy() method
```

```
second[0] = "sister"
print(second)
print(fam) # changing the list second does not modify fam because second is a copy
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['sister', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [3]:
```
third = fam.copy()
print(third)
third[1] = 1.65
print(third)
print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.65, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [4]:
```
fam
```

Out[4]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [5]:
```
list2 = list(fam)
list2[1] = 1.9
print(list2)
print(fam)
```

```
['liz', 1.9, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

You can use list slicing in conjuction with assignment to change values

In [6]:
```
print(fam)
fam[1:3] = [1.8, "jenny"]
print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.8, 'jenny', 1.68, 'mom', 1.71, 'dad', 1.89]
```

List Methods

- `list.copy()`
    - Return a shallow copy of the list. Equivalent to a[:]
- `list.append(x)`
    - Add an item to the end of the list. Equivalent to a[len(a):] = [x].

In [7]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam.append("me")   # unlike R, you don't have to "capture" the result of the function.
# the list itself is modified. You can only append one item.
print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89, 'me']
```

In [8]:
```
fam = fam + [1.8]  # you can also append to a list with the addition `+` operator
# note that this output needs to be 'captured' and assigned back to fam
print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89, 'me', 1.8]
```

In [9]:
```
fam.append('miles')
```

In [10]:
```
fam
```

Out[10]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89, 'me', 1.8, 'miles']
```

In [11]:
```
fam.append(['miles', 1.78, 'joe', 1.8]) # append will add the entire object as one list entry
```

In [12]:
```
fam
```

Out[12]:
```
['liz',
 1.73,
 'emma',
 1.68,
 'mom',
 1.71,
 'dad',
 1.89,
 'me',
 1.8,
 'miles',
 ['miles', 1.78, 'joe', 1.8]]
```

```
In [13]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         fam + ['miles', 1.78, 'joe', 1.8] # plus operator concatenates the lists
```

```
Out[13]: ['liz',
          1.73,
          'emma',
          1.68,
          'mom',
          1.71,
          'dad',
          1.89,
          'miles',
          1.78,
          'joe',
          1.8]
```

```
In [14]: fam
```

```
Out[14]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

```
In [15]: fam * 2
```

```
Out[15]: ['liz',
          1.73,
          'emma',
          1.68,
          'mom',
          1.71,
          'dad',
          1.89,
          'liz',
          1.73,
          'emma',
          1.68,
          'mom',
          1.71,
          'dad',
          1.89]
```

Copy vs. Deep Copy Example

`list.copy` and `list[:]` both create shallow copies. A shallow copy creates a copy of the list, but does not create copies of any objects that the list references.

a deep copy will copy the list and create copies of objects that the list references.

```
In [16]: a = ["a", 1, 2]
         a
```

```
Out[16]: ['a', 1, 2]
```

```
In [17]: b = ["b", 3, 4]
         b
```

```
Out[17]: ['b', 3, 4]
```

```
In [18]: c = [a, b]
         c # c is a list containing list a and list b.
```

```
Out[18]: [['a', 1, 2], ['b', 3, 4]]
```

```
In [19]: d = c.copy()  # d is a shallow copy of c
```

```
In [20]: d
```

```
Out[20]: [['a', 1, 2], ['b', 3, 4]]
```

```
In [21]: import copy
         e = copy.deepcopy(c)  # e is a deep copy of c
```

```
In [22]: e
```

```
Out[22]: [['a', 1, 2], ['b', 3, 4]]
```

```
In [23]: c.append("x")  # modify c
```

```
In [24]: print(c) # c reflects the change and now has 'x' appended to the end.
```

```
[['a', 1, 2], ['b', 3, 4], 'x']
```

```
In [25]: print(d) # d is a copy and is not changed

         [['a', 1, 2], ['b', 3, 4]]

In [26]: print(e) # e is a copy and is not changed

         [['a', 1, 2], ['b', 3, 4]]

In [27]: a.append("z")  # modify list a, an element in c

In [28]: a

Out[28]: ['a', 1, 2, 'z']

In [29]: print(c) # c still contains the 'x' from before and reflects changes made to list a

         [['a', 1, 2, 'z'], ['b', 3, 4], 'x']

In [30]: print(d) # d is a copy of c with references to list a and list b. it reflect the change made to list a.

         [['a', 1, 2, 'z'], ['b', 3, 4]]

In [31]: print(e) # e is a deep copy and contains copies of list a and list b. when list a was changed, the copy inside e is u

         [['a', 1, 2], ['b', 3, 4]]

In [32]: c[1]

Out[32]: ['b', 3, 4]

In [33]: d[1]

Out[33]: ['b', 3, 4]

In [34]: e[1]

Out[34]: ['b', 3, 4]

In [35]: c[1] is d[1] # check to see if c[0] is the same object as d[0]

Out[35]: True
```

```
In [36]: d[1] is e[1] # check to see if d[1] is the same object as e[1]
```

```
Out[36]: False
```

```
In [37]: d[1] == e[1] # check to see if d[1] is equivalent in value as e[1]
```

```
Out[37]: True
```

- `list.insert(i, x)`
    - Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).
- `list.extend(iterable)`
    - Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.

```
In [38]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         fam.insert(4, "joe") # inserts joe at the location of the 4th comma between 1.68 and mom
         print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'joe', 'mom', 1.71, 'dad', 1.89]
```

```
In [39]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         fam.insert(4, ["joe", 2.0])  # trying to insert multiple items by using a list inserts a list
         print(fam)
```

```
['liz', 1.73, 'emma', 1.68, ['joe', 2.0], 'mom', 1.71, 'dad', 1.89]
```

```
In [40]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         fam.insert(4, "joe", 2.0)  # like append, you can only insert one item
         # trying to insert multiple items causes and error
         print(fam)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[40], line 2
      1 fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
----> 2 fam.insert(4, "joe", 2.0)  # like append, you can only insert one item
      3 # trying to insert multiple items causes and error
      4 print(fam)

TypeError: insert expected 2 arguments, got 3
```

In [41]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam.extend(["joe", 2.0]) # lets you add multiple items, but at the end
print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89, 'joe', 2.0]
```

In [42]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam[4:4] = ["joe", 2.0] # Use slice and assignment to insert multiple items in a specific position
print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'joe', 2.0, 'mom', 1.71, 'dad', 1.89]
```

- `list.remove(x)`
  - Remove the first item from the list whose value is x. It is an error if there is no such item.
- `list.pop([i])`
  - Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.
- `list.clear()`
  - Remove all items from the list. Equivalent to del a[:].

In [43]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam.remove("liz")
print(fam)
```

```
[1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

```
In [44]: fam = ["liz", 1.71, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         fam.remove(1.71) # only removes the first match
         print(fam)
```

```
['liz', 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

```
In [45]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         j = fam.pop()  # if you don't specify an index, it pops the last item in the list
         # default behavior of pop() without any arguments is like a stack. last in first out
         print(j)
         print(fam)
```

```
1.89
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad']
```

```
In [46]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         j = fam.pop(0)  # you can also specify an index.
         # Using index 0 makes pop behave like a queue. first in first out
         print(j)
         print(fam)
```

```
liz
[1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

```
In [47]: fam.clear()
         print(fam)
```

```
[]
```

- `list.index(x)`
  - Return zero-based index in the list of the first item whose value is x. Raises a ValueError if there is no such item.
- `list.count(x)`
  - Return the number of times x appears in the list.

```
In [48]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         fam.index("emma")
```

```
Out[48]: 2
```

```
In [49]: fam.index(3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[49], line 1
----> 1 fam.index(3)

ValueError: 3 is not in list
```

In [50]:
```
letters = ["a", "b", "c", "a", "a"]
print(letters.count("a"))
```

```
3
```

In [51]:
```
fam2 = [["liz", 1.73],
["emma", 1.68],
["mom", 1.71],
["dad", 1.89]]
print(fam2.count("emma"))  # the string by itself does not exist
print(fam2.count(["emma", 1.68]))
```

```
0
1
```

- `list.sort(key=None, reverse=False)`
  - Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).
- `list.reverse()`
  - Reverse the elements of the list in place.

In [52]:
```
fam
```

Out[52]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [53]:
```
fam.reverse()  # no output to 'capture', the list is changed in place
```

In [54]:
```
print(fam)
```

```
[1.89, 'dad', 1.71, 'mom', 1.68, 'emma', 1.73, 'liz']
```

In [55]:
```
fam.sort()  # can't sort floats and string
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[55], line 1
----> 1 fam.sort()  # can't sort floats and string

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

In [56]:
```
some_digits = [4, 2, 7, 9, 2, 5.1, 3]
some_digits.sort()  # the list is sorted in place. no need to resave the output
```

In [57]:
```
print(some_digits)  # preserves numeric data types
```

```
[2, 2, 3, 4, 5.1, 7, 9]
```

In [58]:
```
type(some_digits[4])
```

Out[58]:
```
float
```

In [59]:
```
some_digits.sort(reverse = True)
print(some_digits)
```

```
[9, 7, 5.1, 4, 3, 2, 2]
```

In [60]:
```
some_digits = [4, 2, 7, 9, 2, 5.1, 3] # create a new list
sorted(some_digits)  # sorted will return a sorted copy of the list
```

Out[60]:
```
[2, 2, 3, 4, 5.1, 7, 9]
```

In [61]:
```
some_digits  # the list is unaffected
```

Out[61]:
```
[4, 2, 7, 9, 2, 5.1, 3]
```

Strings

A string is a sequence

In [62]:
```
fruit = "bananas"
```

```
In [63]: fruit[0]  # Python is 0-indexed
```

```
Out[63]: 'b'
```

```
In [64]: fruit[1]
```

```
Out[64]: 'a'
```

```
In [65]: fruit[-1] # last letter
```

```
Out[65]: 's'
```

```
In [66]: fruit[1.5]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[66], line 1
----> 1 fruit[1.5]

TypeError: string indices must be integers, not 'float'
```

`len()` tells you the length of a string

```
In [67]: len(fruit)
```

```
Out[67]: 7
```

Subsetting Strings and strings as iterables

You can subset and slice a string much like you would a list or tuple:

```
In [68]: s = 'abcdefghijklmnopqrstuvwxyz'
```

```
In [69]: s[4:9]
```

```
Out[69]:  'efghi'
```

```
In [70]:  s[-6:]
```

```
Out[70]:  'uvwxyz'
```

```
In [71]:  for x in s[0:5]:
              print(x + '!')
```

```
a!
b!
c!
d!
e!
```

Strings are immutable

This means that when you use a method on a string, it does not modify the string itself and returns a new string object.

```
In [72]:  # strings are immutable. You cannot modify a string that has been created.
          s[0] = 'b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[72], line 2
      1 # strings are immutable. You cannot modify a string that has been created.
----> 2 s[0] = 'b'

TypeError: 'str' object does not support item assignment
```

```
In [73]:  'b' + s[1:] # if i wanted the string where the first letter is now b
```

```
Out[73]:  'bbcdefghijklmnopqrstuvwxyz'
```

String Methods

In [74]:
```python
name = "STATS 21 python and other technologies for data science"
print(name.upper())
print(name.capitalize()) # first character is capitalized
print(name.title())     # first character of each word is capitalized
print(name.lower())
print(name) # string itself is not modified
```

```
STATS 21 PYTHON AND OTHER TECHNOLOGIES FOR DATA SCIENCE
Stats 21 python and other technologies for data science
Stats 21 Python And Other Technologies For Data Science
stats 21 python and other technologies for data science
STATS 21 python and other technologies for data science
```

Count how many times a letter appears

In [75]:
```python
count = 0
for letter in name:
    if letter == "e":
        count = count + 1
print(count)
```

```
5
```

In [76]:
```python
# can be achieved with a simple method:
name.count("e")
```

Out[76]:
```
5
```

In [77]:
```python
name.index('A') # index of the first instance
```

Out[77]:
```
2
```

In [78]:
```python
name.endswith("k")
```

Out[78]:
```
False
```

In [79]:
```python
name.endswith("e")
```

Out[79]:
```
True
```

```
In [80]: name.startswith("s")  # case sensitive
```

```
Out[80]: False
```

```
In [81]: # create multi-line strings with triple quotes
         name2 = '''   miles chen


         '''
         print(name2)
```

```
   miles chen



```

```
In [82]: name2.strip()  # removes extra whitespace
```

```
Out[82]: 'miles chen'
```

```
In [83]: name2 # remember strings are immutable, the original string still has the white space
```

```
Out[83]: '   miles chen \n\n\n'
```

string.split()

```
In [84]: name2.split() # the result of split() is a list
```

```
Out[84]: ['miles', 'chen']
```

```
In [85]: num_string = "2,3,4,7,8"
         print(num_string.split()) # defaults to splitting on space
         print(num_string.split(','))
```

```
['2,3,4,7,8']
['2', '3', '4', '7', '8']
```

```
In [86]: # list comprehension (covered later) to convert the split strings into int
         [int(x) for x in num_string.split(',')]
```

```
Out[86]: [2, 3, 4, 7, 8]
```

```
In [87]: # the list comprehension is a more concise version of the following code
         l = []
         for x in num_string.split(','):
             l.append(int(x))
         l
```

```
Out[87]: [2, 3, 4, 7, 8]
```

```
In [88]: print(name)
         print(name.isalpha()) # has spaces and digits, so it is not strictly alpha
         name3 = "abbaAZ"
         name3.isalpha()
```

```
STATS 21 python and other technologies for data science
False
```

```
Out[88]: True
```

```
In [89]: name4 = "abbaAZ4"
         name4.isalpha()
```

```
Out[89]: False
```

```
In [90]: # strings can span multiple lines with triple quotes
         long_string = """Lyrics to the song Hallelujah
         Well I've heard there was a secret chord
         That David played and it pleased the Lord
         But you don't really care for music, do you?"""
         shout = long_string.upper()
         print(shout)
         word_list = long_string.split() # separates at spaces
         print(word_list)
```

```
LYRICS TO THE SONG HALLELUJAH
WELL I'VE HEARD THERE WAS A SECRET CHORD
THAT DAVID PLAYED AND IT PLEASED THE LORD
BUT YOU DON'T REALLY CARE FOR MUSIC, DO YOU?
['Lyrics', 'to', 'the', 'song', 'Hallelujah', 'Well', "I've", 'heard', 'there', 'was', 'a', 'secret', 'chord', 'Tha
t', 'David', 'played', 'and', 'it', 'pleased', 'the', 'Lord', 'But', 'you', "don't", 'really', 'care', 'for', 'musi
c,', 'do', 'you?']
```

In [91]:
```
long_string.splitlines() # separates at line ends
# you'll notice that python defaults to using single quotes, but if the string contains an apostrophe,
# it will use double quotes
```

Out[91]:
```
['Lyrics to the song Hallelujah',
 "Well I've heard there was a secret chord",
 'That David played and it pleased the Lord',
 "But you don't really care for music, do you?"]
```

In [92]:
```
long_string.count("e")
```

Out[92]:
```
15
```

Searching for a letter

```
long_string = """Lyrics to the song Hallelujah
Well I've heard there was a secret chord
That David played and it pleased the Lord
But you don't really care for music, do you?"""
```

In [93]:
```
def myfind(string, letter):
    index = 0
    while index < len(string):
        if string[index] == letter:
            return index
        index = index + 1
    return -1
```

In [94]:
```
myfind(long_string, "t")
```

Out[94]:
```
7
```

```
In [95]: # Python already has a find method built in
         long_string.find("t") # index of the first instance of 't'
```

```
Out[95]: 7
```

```
In [96]: long_string.index('t') # string.index() and string.find() are similar.
```

```
Out[96]: 7
```

```
In [97]: long_string.find('$') # string.find() returns a -1 if the character doesn't exist in the string
```

```
Out[97]: -1
```

```
In [98]: long_string.index('$')  # string.index() returns error if the character doesn't exist in the string.
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[98], line 1
----> 1 long_string.index('$')  # string.index() returns error if the character doesn't exist in the string.

ValueError: substring not found
```

`in` operator

returns a boolean value if the first string is a substring of the second string.

```
In [99]: 'a' in 'bananas'
```

```
Out[99]: True
```

```
In [100… 'nan' in 'bananas'
```

```
Out[100… True
```

```
In [101… 'bad' in 'bananas'
```

```
Out[101… False
```

String comparisons

Use of `>` or `<` compares strings in alphabetical order.

```
In [102… 'A' < 'B'
```

```
Out[102… True
```

```
In [103… 'a' < 'b'
```

```
Out[103… True
```

```
In [104… 'Z' < 'a'
```

```
Out[104… True
```

```
In [105… # digits are less than capital letters
         '1' < 'A'
```

```
Out[105… True
```

```
In [106… '0' < '00'
```

```
Out[106… True
```

```
In [107… # must treat digits like "letters" with alphabetical rules
         '11' < '101'
```

```
Out[107… False
```

```
In [108… '!' < '@' # the sorting of symbols feels very arbitrary
```

```
Out[108… True
```

```
In [109…  # sorted order
          string = '!@#$%^&*()[]{}\|;:,.<>/?1234567890ABCXYZabcxyz'
          x = sorted(string)
          print(x)
```

```
['!', '#', '$', '%', '&', '(', ')', '*', ',', '.', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';',
'<', '>', '?', '@', 'A', 'B', 'C', 'X', 'Y', 'Z', '[', '\\', ']', '^', 'a', 'b', 'c', 'x', 'y', 'z', '{', '|', '}']
```

```
<>:2: SyntaxWarning: invalid escape sequence '\|'
<>:2: SyntaxWarning: invalid escape sequence '\|'
C:\Users\miles\AppData\Local\Temp\ipykernel_30532\3270986160.py:2: SyntaxWarning: invalid escape sequence '\|'
  string = '!@#$%^&*()[]{}\|;:,.<>/?1234567890ABCXYZabcxyz'
```