Lecture 3-1

Lists Part 2 and Strings

Week 3 Monday

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.

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]:
```
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]
```

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]:
```
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]:
```
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]
```

Ways to make a copy of a list that is a different object:

fam[:}
fam.copy()
list(fam)

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].

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']
```

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]
```

fam.apend() modifies the list itself
if we want to append with + [1.8] we have to reassign to the name fam

In [9]:

```
fam.append('miles')
```

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 [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 [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]]
```

**Will append the entire list as 1 item,
now we have a nested list inside our list.
If we use the + with the list then they get appended
and we don't get a list nested in our list**

[ ] and list() are both used to create lists

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 [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]
```

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.

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 [15]:

```
a = ["a", 1, 2]
b = ["b", 3, 4]
c = [a, b]

import copy
d = c[:]  # d is a shallow copy of c
e = copy.deepcopy(c)  # e is a deep copy of c

c.append("x")  # modify c
print(c) # c reflects the change
print(d) # d is a copy and is not changed
print(e) # e is a copy and is not changed
```

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

Neither the deep copy nor the shallow copy are changed in this case

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 [15]:

```
a = ["a", 1, 2]
b = ["b", 3, 4]
c = [a, b]

import copy
d = c[:]  # d is a shallow copy of c
e = copy.deepcopy(c)  # e is a deep copy of c

c.append("x")  # modify c
print(c) # c reflects the change
print(d) # d is a copy and is not changed
print(e) # e is a copy and is not changed
```

You must import copy to make deep copies. The normal copy syntax creates shallow copies

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

In [16]:

```
a.append("z")  # modify list a, an element in c
print(c) # c reflects change
print(d) # d copies the structure of c and reflects the change
print(e) # is a deep copy and is not affected by changes to underlying elements
```

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

- `list.insert(i, x)` **if insert a list, will have a nested list now**

  you can only insert one item at a time
    - 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)` **if insert a list, will not have a nested list**

    - Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.

In [17]:

```
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]
```

Insert “joe” right after the fourth comma

In [17]:

```
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 [18]:

```
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 [17]:

```
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 [18]:

```
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 [19]:

```
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)
<ipython-input-19-cb6806003168> in <module>
      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() takes exactly 2 arguments (3 given)
```

In [20]:

```
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 [20]:

```
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 [21]:

```
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]
```

This is the way to insert multiple items in the middle of the list without having to create a nested list

- `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[:].

- `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 [22]:

```
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 [23]:

```
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 [23]:

```
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 [24]:

```
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)

fam.clear()
print(fam)
```

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

- `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.

- `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 [25]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam.index("emma")
```

Out[25]:
```
2
```

- `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 [25]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam.index("emma")
```

Out[25]: 2

In [26]:
```
fam.index(3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-63a35f148e9b> in <module>
----> 1 fam.index(3)

ValueError: 3 is not in list
```

- `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 [25]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam.index("emma")
```

Out[25]:
```
2
```

In [26]:
```
fam.index(3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-63a35f148e9b> in <module>
----> 1 fam.index(3)

ValueError: 3 is not in list
```

In [27]:
```
letters = ["a", "b", "c", "a", "a"]
print(letters.count("a"))
```

```
3
```

In [28]:

```
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.

- `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 [29]:
```
fam.reverse()  # no output to 'capture', the list is changed in place
```

- `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 [29]:
```
fam.reverse()  # no output to 'capture', the list is changed in place
```

In [30]:
```
print(fam)
```

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

- `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 [29]:
```
fam.reverse()  # no output to 'capture', the list is changed in place
```

In [30]:
```
print(fam)
```

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

In [31]:
```
fam.sort()  # can't sort floats and string
```

cant sort the combo

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-b0f2370e264b> in <module>
----> 1 fam.sort()  # can't sort floats and string

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

In [32]:

```
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 [32]:
```
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 [33]:
```
print(some_digits)  # preserves numeric data types
```

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

In [32]:
```
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 [33]:
```
print(some_digits)  # preserves numeric data types
```

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

In [34]:
```
type(some_digits[4])
```

Out[34]:
```
float
```

In [32]:
```
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 [33]:
```
print(some_digits)  # preserves numeric data types
```

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

In [34]:
```
type(some_digits[4])
```

Out[34]:
```
float
```

In [35]:
```
some_digits.sort(reverse = True)
print(some_digits)
```

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

In [36]:

```
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[36]:

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

In [36]:
```
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[36]:
```
[2, 2, 3, 4, 5.1, 7, 9]
```

In [37]:
```
some_digits  # the list is unaffected
```

Out[37]:
```
[4, 2, 7, 9, 2, 5.1, 3]
```

Strings

A string is a sequence

Strings

A string is a sequence

In [38]:
```
fruit = "bananas"
```

Strings

A string is a sequence

In [38]:
```
fruit = "bananas"
```

In [39]:
```
fruit[0]  # Python is 0-indexed
```

Out[39]:
```
'b'
```

Strings

A string is a sequence

In [38]:
```
fruit = "bananas"
```

In [39]:
```
fruit[0]  # Python is 0-indexed
```

Out[39]:
```
'b'
```

In [40]:
```
fruit[1]
```

Out[40]:
```
'a'
```

Strings

A string is a sequence

In [38]:
```
fruit = "bananas"
```

In [39]:
```
fruit[0]  # Python is 0-indexed
```

Out[39]:
```
'b'
```

In [40]:
```
fruit[1]
```

Out[40]:
```
'a'
```

In [41]:
```
fruit[-1] # last letter
```

Out[41]:
```
's'
```

Strings

A string is a sequence

In [38]:
```
fruit = "bananas"
```

In [39]:
```
fruit[0]  # Python is 0-indexed
```

Out[39]:
```
'b'
```

In [40]:
```
fruit[1]
```

Out[40]:
```
'a'
```

In [41]:
```
fruit[-1] # last letter
```

Out[41]:
```
's'
```

In [42]:
```
fruit[1.5]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-42-bf9cc58e8398> in <module>
----> 1 fruit[1.5]
```

```
TypeError: string indices must be integers
```

This is nchar() in R

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

In [43]:
```
len(fruit)
```

Out[43]:
```
7
```

Subsetting Strings and strings as iterables

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

Subsetting Strings and strings as iterables

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

In [44]:
```
s = 'abcdefghijklmnopqrstuvwxyz'
```

Subsetting Strings and strings as iterables

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

In [44]:
```
s = 'abcdefghijklmnopqrstuvwxyz'
```

In [45]:
```
s[4:9]
```

Out[45]:
```
'efghi'
```

Subsetting Strings and strings as iterables

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

In [44]:
```
s = 'abcdefghijklmnopqrstuvwxyz'
```

In [45]:
```
s[4:9]
```

Out[45]:
```
'efghi'
```

In [46]:
```
s[-6:]
```

Out[46]:
```
'uvwxyz'
```

Subsetting Strings and strings as iterables

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

In [44]:
```
s = 'abcdefghijklmnopqrstuvwxyz'
```

In [45]:
```
s[4:9]
```

Out[45]:
```
'efghi'
```

In [46]:
```
s[-6:]
```

Out[46]:
```
'uvwxyz'
```

In [47]:
```
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.

cannot modify a string in place.

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 [48]:
```
# strings are immutable. You cannot modify a string that has been created.
s[0] = 'b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-26412a6b3b58> in <module>
      1 # strings are immutable. You cannot modify a string that has been create
d.
----> 2 s[0] = 'b'

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

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 [48]:
```
# strings are immutable. You cannot modify a string that has been created.
s[0] = 'b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-26412a6b3b58> in <module>
      1 # strings are immutable. You cannot modify a string that has been create
d.
----> 2 s[0] = 'b'

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

In [49]:
```
'b' + s[1:] # if i wanted the string where the first letter is now b
```

Out[49]:
```
'bbcdefghijklmnopqrstuvwxyz'
```

different strings always point to different objects i n memoery

String Methods

String Methods

In [50]:

```
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

Count how many times a letter appears

In [51]:

```
count = 0
for letter in name:
    if letter == "e":
        count = count + 1
print(count)
```

```
5
```

Count how many times a letter appears

In [51]:
```
count = 0
for letter in name:
    if letter == "e":
        count = count + 1
print(count)
```

```
5
```

In [52]:
```
# can be achieved with a simple method:
name.count("e")
```

Out[52]:
```
5
```

In [53]:
```
name.index('A') # index of the first instance
```

Out[53]:
```
2
```

In [53]:
```
name.index('A') # index of the first instance
```

Out[53]:
```
2
```

In [54]:
```
name.endswith("k")
```

Out[54]:
```
False
```

In [53]:
```
name.index('A') # index of the first instance
```

Out[53]:
```
2
```

In [54]:
```
name.endswith("k")
```

Out[54]:
```
False
```

In [55]:
```
name.endswith("e")
```

Out[55]:
```
True
```

In [53]:
```
name.index('A') # index of the first instance
```

Out[53]:
```
2
```

In [54]:
```
name.endswith("k")
```

Out[54]:
```
False
```

In [55]:
```
name.endswith("e")
```

Out[55]:
```
True
```

In [56]:
```
name.startswith("s")  # case sensitive
```

Out[56]:
```
False
```

In [57]:

```
# create multi-line strings with triple quotes
name2 = '''   miles chen


'''
print(name2)
```

```
   miles chen
```

In [57]:
```
# create multi-line strings with triple quotes
name2 = '''   miles chen


'''
print(name2)
```

```
   miles chen
```

In [58]:
```
name2.strip()  # removes extra whitespace
```

Out[58]:
```
'miles chen'
```

In [57]:
```
# create multi-line strings with triple quotes
name2 = '''   miles chen


'''
print(name2)
```

```
   miles chen
```

In [58]:
```
name2.strip()  # removes extra whitespace
```

Out[58]:
```
'miles chen'
```
Does not actually modify the string

In [59]:
```
name2 # remember strings are immutable, the original string still has the white space
```

Out[59]:
```
'   miles chen \n\n\n'
```

string.split()

string.split()

In [60]:
```
name2.split() # the result of split() is a list
```

Out[60]:
```
['miles', 'chen']
```

default split is at whitespace

string.split()

In [60]:
```
name2.split() # the result of split() is a list
```

Out[60]:
```
['miles', 'chen']
```

In [61]:
```
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']
```

string.split()

In [60]:
```
name2.split() # the result of split() is a list
```

Out[60]:
```
['miles', 'chen']
```

In [61]:
```
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 [62]:
```
# list comprehension (covered later) to convert the split strings into int
[int(x) for x in num_string.split(',')]
```

Out[62]:
```
[2, 3, 4, 7, 8]
```

string.split()

In [60]:
```
name2.split() # the result of split() is a list
```

Out[60]:
```
['miles', 'chen']
```

In [61]:
```
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 [62]:
```
# list comprehension (covered later) to convert the split strings into int
[int(x) for x in num_string.split(',')]
```

Out[62]:
```
[2, 3, 4, 7, 8]
```

In [63]:
```
# 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[63]:
```
[2, 3, 4, 7, 8]
```

In [64]:

```
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[64]:

```
True
```

In [64]:
```
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[64]:
```
True
```

In [65]:
```
name4 = "abbaAZ4"
name4.isalpha()
```

Out[65]:
```
False
```

In [66]:

```
# 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', 'That', 'David', 'played', 'and', 'it', 'pleased',
'the', 'Lord', 'But', 'you', "don't", 'really', 'care', 'for', 'music,', 'do', 'y
ou?']
```

In [67]:

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

Out[67]:

```
['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 [67]:
```
long_string.splitlines() # separates at line ends
# you'll notice that python defaults to using single quotes, but if the string contains an apostroph
# it will use double quotes
```

Out[67]:
```
['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 [68]:
```
long_string.count("e")
```

Out[68]:
```
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?"""
```

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 [69]:

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

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 [69]:

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

In [70]:

```
myfind(long_string, "t")
```

Out[70]: 7

In [71]:
```
# Python already has a find method built in
long_string.find("t") # index of the first instance of 't'
```

Out[71]:
```
7
```

In [71]:
```
# Python already has a find method built in
long_string.find("t") # index of the first instance of 't'
```

Out[71]:
```
7
```

In [72]:
```
long_string.index('t') # string.index() and string.find() are similar.
```

Out[72]:
```
7
```

In [71]:
```
# Python already has a find method built in
long_string.find("t") # index of the first instance of 't'
```

Out[71]:
```
7
```

In [72]:
```
long_string.index('t') # string.index() and string.find() are similar.
```

Out[72]:
```
7
```

In [73]:
```
long_string.find('$') # string.find() returns a -1 if the character doesn't exist in the string
```

Out[73]:
```
-1
```

The only difference between .find() and .index() is that when you use .find() on something that is not in the string, then it returns a -1 and .index() has an error

In [71]:
```
# Python already has a find method built in
long_string.find("t") # index of the first instance of 't'
```

Out[71]: 7

In [72]:
```
long_string.index('t') # string.index() and string.find() are similar.
```

Out[72]: 7

In [73]:
```
long_string.find('$') # string.find() returns a -1 if the character doesn't exist in the string
```

Out[73]: -1

In [74]:
```
long_string.index('$')  # string.index() returns error if the character doesn't exist in the string.
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-74-5b5715be5537> in <module>
----> 1 long_string.index('$')  # string.index() returns error if the character d
oesn'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` operator

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

In [75]:
```
'a' in 'bananas'
```

Out[75]:
```
True
```

`in` operator

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

In [75]:
```
'a' in 'bananas'
```

Out[75]:
```
True
```

In [76]:
```
'nan' in 'bananas'
```

Out[76]:
```
True
```

`in` operator

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

In [75]:
```
'a' in 'bananas'
```

Out[75]:
```
True
```

In [76]:
```
'nan' in 'bananas'
```

Out[76]:
```
True
```

In [77]:
```
'bad' in 'bananas'
```

Out[77]:
```
False
```

String comparisons

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

In [78]:
```
'A' < 'B'
```

Out[78]:
```
True
```

In [78]:
```
'A' < 'B'
```

Out[78]: True

In [79]:
```
'a' < 'b'
```

Out[79]: True

In [80]:
```
'Z' < 'a'
```

Out[80]: True

In [81]:
```
# digits are less than capital letters
'1' < 'A'
```

Out[81]: True

In [82]:
```
'0' < '00'
```

Out[82]: True

In [83]:
```
# must treat digits like "letters" with alphabetical rules
'11' < '101'
```

Out[83]: False

In [84]:
```
'!' < '@' # the sorting of symbols feels very arbitrary
```

Out[84]: True

In [85]:

```
# 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', '{', '|', '}']
```