name = “Fuck me Up”

name.lower() 🡪 “fuck me up”

name.upper() 🡪 “FUCK ME UP”  
name.capitalize() 🡪 “Fuck me up”

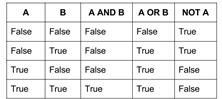
name.title() 🡪 “Fuck Me Up”

name.count(“u”) 🡪 2

name.find(“ “) 🡪 4 (returns index)

“aaaabbbffssassa”.strip(“a”) 🡪 bbbffssss

“abracadabra”.replace(“a”,””) 🡪 brcdbr



list.insert(i, x): inserts an item at a given position (the first argument is the index of the element before which to insert)

list.remove(x): removes the first item from the list whose value is x (it is an error if there is no such item)

list.pop(i): removes the item at the given position in the list, and returns it (if no index is specified, it removes and returns the last item in the list)

list.index(x): returns the index in the list of the first item whose value is x (it is an error if there is no such item)

.sort() changes the original list even if you assign it to another variable

sorted() creates a new list without changing the original list

L[:] returns a copy of the list, does not change original list

Writing a text file:

f\_out = open("outfile.txt","w")

f\_out.write("Hello world!\n")

Appending to text file:

f\_out = open("outfile.txt","a")

Reading text file:

for line in open(“filename.txt”)

lines.strip() #strips extra white space

random.seed(): initializes internal state of the random number generator

random.randint(a, b): returns a random integer N such that a <= N <= b.

random.choice(seq): returns a random element from the non-empty sequence seq (if seq is empty, raises IndexError)

random.random(): return the next random floating point number in the range [0.0, 1.0]

s.add(x): adds element to set

s.clear(): clears out set, making it empty

s1.difference(s2): creates new set with values from s1 that are not in s2 (s1 – s2)

s1.intersection(s2): creates new set that contains only values that are in both sets (s1 & s2)

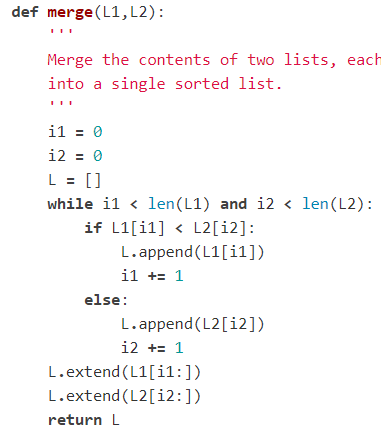
s1.union(S2): creates a new set that contains values that are in either set (s1 | s2)

s1.symmetric\_difference(s2): creates a new set that contains values that are in s1 or s2 but not in both (s1 ^ s2)

s1.issubset(s2): are all elements of s1 also in s2? s1 <= s2

s1.issuperset(s2): are all elements of s2 also in s1? s1 >= s2

other operator: x in s

sorted(dictionary.key())

list(dictionary.values())

d.clear(): removes all items

d.get(): returns value of key

d.popitem(): returns and removes element

d.update(): updates dictionary

d.remove(x): removes specified element, but fails if its not there

d.discard(x): removes specified element, but does nothing if its not there

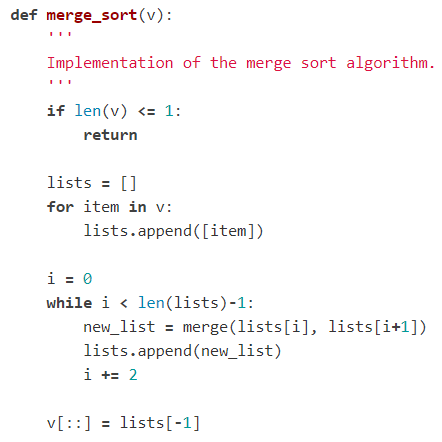
class Name(object)

\_\_init\_\_(self, var1,var2):

self.v1 = var1

self.v2 = var2

function1(self):

 add = self.v1 + self.v2

return add

x = Name(2,5)

x.function()

map(function,list)

map(len/sort/max,list\_of\_lists)

map(area,radii)

can use your own created function as function argument

using map: print(list(map(blah,blah)))

def cmp(s):

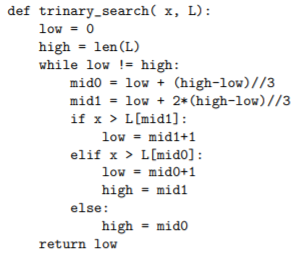
return len(s), s

L = [ "apple", "ball", "car", "card", "basket", "car" ]

L.sort(key=cmp)

print(L)

['car', 'car', 'ball', 'card', 'apple', 'basket']



lambda x: x\*\*2

lambda p: (p[0]\*\*2 + p[1]\*\*2)\*\*0.5

list(map(lambda x: x\*\*2, array))

extracting/eliminating values in list: filter

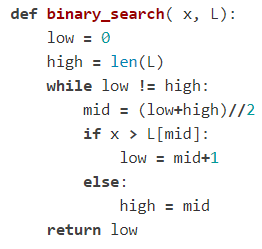
v = [ 1, 9, -4, -8, 10, -3 ]

list(filter(lambda x: x>0, v))

[1, 9, 10]

words = ['python', 'queue', 'blue', 'coconut', 'true', 'grail']

list(filter(lambda word: 'ue' in word, words))

['queue', 'blue', 'true']

sortng in reverse/sorting by something else:

sorted(list, reverse = True)

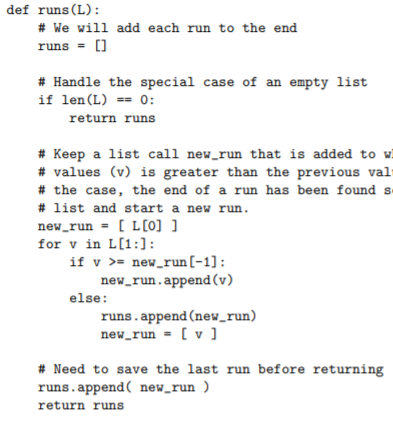
L.sort(key = lambda x: [x[1], x[0]])

sorted(list, key = lambda x: x[1])

list comprehension:

[ i\*i for i in range(1,9) ]

[1, 4, 9, 16, 25, 36, 49, 64]

v = [ 1, 9, -4, -8, 10, -3 ]

[ x for x in v if x>0 ]

[1, 9, 10]

recursive to iterative:

1. Ensure that your recursive call is the last statement in the function and that it is just a recursive call by itself (not part of a more complex expression). In order to do that, introduce, if necessary, an additional parameter in the function that will store the "accumulated" value of whatever we are trying to compute.

2. Put your code in a loop. Replace a recursive call with an assignment. Remove the now unnecessary additional parameter from the function signature.

