#___________ ways of using functions _____________________________

#0.py

# Your text-book calls certain functions "higher-order function". Why?
# Such a function takes as arguments: some function (or list of functions),
# and a set of data values.
# It applies the function (or list of functions) to each of the data values
# and returns, in general,
# a list of results.

# We will see examples of these higher-order functions in file 1.py.
# Before that, let's look at some simple things we can do with functions.
# This is the output from the Idle shell. Try it out yourself.

>>> >>>
>>> abs # making f represent the abs function
>>> f
<built-in function abs>
>>> f(-4)
4
>>> flist = [abs, math.sqrt] # flist is a list of two functions
>>> flist
[<built-in function abs>, <built-in function sqrt>]
>>> # Here is another way to call the math.sqrt function
>>> flist[1](9)
3.0
>>> #Now look at how we can pass a function to another function as an
# argument/parameter
>>> def example(function_name, data_for_the_function):
    return function_name(data_for_the_function) #calling the func,
returning result

```python
>>> example(abs, -4)
4
>>> example(math.sqrt, 9)
3.0
>>> example(math.sqrt, 2)
1.4142135623730951
>>> example(flist[1], 25)
5.0
```
def test1():
    f = lambda x, y: (x+y)*(x+y)  # want the function (a+b)^2 for parameters a, b

    # Now say x = 21 and y = 27
    result = f(21, 27)

    print("result = ",result)

# The use of Lambda becomes clear when you use it together with the "map" function

# What is the map function? Let's understand what the map function does.
# The map function allows you to apply a given function to all of the items in a sequence.

result = map (some_function, sequence)

So you see the first parameter is a function, and then comes any sequence you want the function to act on

# Example: 1 mi (mile) = 1.60934 km (kilometers). And 1 km = 0.621371 mi.
# Suppose we want conversion functions

def miles(km):
    return(km*0.621371)  # takes in km, returns miles

def kilometers(mi):
    return(mi*1.60934)  # takes in mi, returns km

# Let's say that for our Christmas vacation we have the choice of driving to five places. Since we do things in terms of "miles" in the US, the distances are all in miles.
# Say distances = [121, 164, 197, 229, 315] five distances in miles
# We'd like to convert them all to km, and maybe also then back to miles
def test2():
    
dists_in_miles = [121, 164, 197, 229, 315]

K = list(map(kilometers, dists_in_miles))  # get list of distances in km

print("Distances in km:", ["%0.2f" % d for d in K])  # also made the distances strings in output

M = list(map(miles, K))  # get list of distances in miles

print("Distances in miles:", ["%0.2f" % d for d in M])  # also made the distances strings in output

# You can apply map to several lists, but each list has to have the same length.

# Suppose we want to find the products of numbers in three lists. First we'll write a function
# that map can use: this function will find the product of three numbers

def prod3(x,y,z): return(x*y*z)

def test3():
    
    Products = list(map(prod3, [2, 4, 6, 8], [3, 5, 7, 9], [2, 2, 2, 2]))

    print("List of products: ", Products)
    
    #Observe how it uses the first item in each list, then the second, then the third, etc.

    ...

    IMORTANT: Note that the map function applied some function to DATA in a sequence
                ...

    # Now we understand the "map" function. Let's look at how to use the map function together
    # with the lambda operator
    #
    # How to combine the lambda operator and map function
Suppose you did not want to have to write and keep functions like "kilometers", "miles" and "prod3". Instead you want to define and use them on-the-fly WITHOUT NAMES, and discard them immediately after using them (so storage requirement are reduced)

```python
def miles(km):
    return (km*0.621371)  # takes in km, returns miles

def kilometers(mi):
    return (mi*1.60934)

def test4():
    dists_in_miles = [121, 164, 197, 229, 315]

    # Now these functions will seem "anonymous" ...... they have no names
    K = list(map(lambda mi: mi*0.621371, dists_in_miles))  # define & use this function only here

    print(" Distances in km:", ["%0.2f" %d for d in K])  # also made the distances strings in output

    M = list(map(lambda km: km*1.60934, K))  # define and use this function only here

    print(" Distances in miles:", ["%0.2f" %d for d in M])  # also made the distances strings in output

    Products = list(map(lambda a, b, c: a*b*c, [2, 4, 6, 8], [3, 5, 7, 9], [2, 2, 2, 2]))

    print(" List of products: ", Products)
```

# Now the map and lambda idea do not apply to just data. They can apply to functions too.

# Lets say we want x^2, x^3, x^4, x^5, x^6, x^7 for some input x. We'll use simple one-line functions
# in our example, but they need not be so simple

```python
def power2(x):
    return(x**2)
def power3(x):
    return(x**3)
def power4(x):
    return(x**4)
def power5(x):
    return(x**5)
def power6(x):
    return(x**6)
def power7(x):
    return(x**7)
```
def test5():

    # Recall that the "map" function has two arguments. The second argument is a sequence.
    # We'll give "map" the following list of functions for this sequence.
    function_list = [power2, power3, power4, power5, power6, power7]  # list of function names

    # The first argument to "map" is a function. Since we want to apply the above power functions to
    # a list of numbers, we'll use a lambda function to generate that list of numbers.
    for i in range(7):
        powers = list(map(lambda x: x(i), function_list))
        print(powers)  # first all powers of 0, then 1, then 2, then 3, ...
        # up to powers of 6

    # The filter function enables us to apply a function to each of the elements in a list. Each item for
    # which the result is True will be included in the result list; the others are ignored.
    # filter(function, list)

def test6():

    # Let's filter a list to get only the even numbers
    data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
    evens = list(filter(lambda x: not (x%2), data))
    print(evens, "\n")

    # Let's filter a list to get names only longer than 5 characters
    longer_names = list(filter(lambda x: len(x)>5, names))
    print(longer_names)

    #___________ Designing Algorithms to solve problems
#2.py

''' What will we learn here?

1. What is "searching"? How do we implement the LINEAR search and BINARY search algorithms?

2. How do we analyze the above two algorithms? That means: how can we tell "how much time" each algorithm will take to run, given some "input work of size n".

LINEAR SEARCH
------------------------

Let's say a data file contains a list of numbers. We want to know if some number x (say x = 5)
is in the list, and if so, what is its position.

Because linear search is so "well-defined", Python has a built-in function that implements it.

    search(x, list):
        if x is in the list, return it's position
        if x is not in the list, return -1

Python implements it's built-in linear search as the "index" method for a list. But there is a small difference from our above definition of search. If x is not in the list, Python's "index" method raises an exception instead of returning -1.

Here is an example.

...  

# When x (x = 5) is not in the list, we get an exception with Python's linear search

...  

>>> somelist = [10, 20, 12, 17, 78, 99, 197, 23]

>>> somelist.index(5)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    somelist.index(5)
ValueError: 5 is not in list

# But when 5 is in the list, "index" returns its position
>>> somelist = [10,20,12,17,5,78,99,197,23]
>>> somelist.index(5)
4

# Let's implement our linear search algorithm. What do we have to do?

# 1. Traverse the list from left to right.
# 2. If we encounter the item x we are search for, return the position in
# the list (i.e., the index)
# 3. If we reach the end of the list and have not found x, return -1
# (because x is not in list).

def search0 (x, numberList):    # this search cheats and calls Python's
                                # for an easy way out :)
    try:
        return numberList.index(x)
    except:                        # the index method raises an
        return(-1)                  # exception if x is not in the list

# Now let's implement the search from scratch.

def search(x, numberList):
    for j in range(len(numberList)):    # walking down the list from left
to right
        if (x == numberList[j]):
            return (j)                    # Hey! we found x.
So return it's index.

    return(-1)                        # we traversed the
    entire list and x was not in there.

# IMPORTANT NOTE: If there are n items in the list, then in order to check
# if x is in there
# you have to check every list item. Because this takes "n comparisons" we
call this an
# O(n) algorithm.
Let $n$ be the size of the input to the algorithm. Let $f(n)$ be the number of comparisons (also called the worst-case run time) of the algorithm.

If there exists a constant $c > 0$ such that

$$f(n) \leq c \cdot n \quad \text{for all } n \geq 0$$

then the algorithm is an $O(n)$ algorithm.

That expression says: the number of comparisons is capped from above (or bounded from above) by some (i.e., any satisfactory) constant times $n$, for all $n$ that is non-negative.

# What if we did something crazy and jumped through the list at random hoping to find $x$ quickly if it is in there?

# We may get lucky and find it fast, but on average it will still be $O(n)$, and sometimes it may take much longer.

# Reason in this way. Say the list is of size $n$, and put $x$ in the last position. Now with probability
# $1/n$ you will hit $x$ if you jump in randomly. But with probability $(n-1)/n$ you will hit one of
# the other numbers in the list.

# This will repeat on every try, until you hit $x$.

# The number of tries to hit $x$ is called a Geometric random variable (just as if you were
# tossing a coin that came up Heads with probability $1/n$, and Tails with probability $(n-1)/n$,
# and you were counting the number of tries for it to come up Heads).

# Since the parameter of this Geometric random variable is $1/n$, the average time to
# find $x$ is $n$. So it is still an $O(n)$ time algorithm on average, but its worst-case time can
# be much larger than the simple linear search. Why? You may have really bad luck
# and keep hitting the non $x$ numbers for a really long time.

# But there is another problem. If $x$ is not in the list, the algorithm might
run forever.
# There are different ways of handling this. On way is to stop after a
certain number of
# tries. Another way is to build a list of new positions that have been
encountered. If
# we do this then we can make the algorithm not repeat it's search
positions. But now,
# at the expense of more book-work (maintaining lists of already checked
positions;
# generating random positions to check from the rest) we are slowly
converging on
# exactly the linear search, except that we are searching in a random order.

# So in both cases, we end up with an O(n) algorithm.

# At this point we understand the linear search very well.

# Can we do better than O(n). That is, can we beat the linear search?
# YES! Provided the list is already sorted.

# __________________________ Binary Search __________________________

#3.py

# We need the list to be sorted in order to run the binary search algorithm.

# Example:
'

19 elements in this sorted list. We want to look for the number 3.

list:  1  3  4  7  9  11  21  27  30  31  35  39  41  52  56 57  59  61  64

(middle of list is 10th element 31; but one to the left or right is also
okay -- approximately middle)

step1: jump to the middle of the list and check if that position holds x =
3.

step2: If not equal, is x small or larger? If smaller, the new list to
search is the sublist on the left

If larger, the new list to
search is the sublist on the right

In our case 3 is smaller than 31, so the new list we'll work with
There are 9 elements in this list. Simply repeat Steps 1 and 2 until we either find 3 in the list or realize that we have reached a sublist of size 1 and that number is not 3.

Once again 3 is not equal to middle element (9), but 3 is smaller. So the new list to work on is:

1 3 4 7

Now there are 4 elements in the list (even number). In the previous two cases the list had an odd number of elements and so the middle element was easy to define. But which element will we use as the middle element now?

Convention: midpoint = (low + high) / / 2 (integer division; divide and truncate)

in our case: (0 + 3) // 2 = 1

(Note: this midpoint formula will work for both the odd n and even n cases, and it does not cause overflow and generate a bad index. It will not give the midpoint we used in the first two cases above, but it will always work, without worrying about odd and even sized lists)

But now when we compare 3 to list[1] we find an exact match, and return our answer,

which is "position 1".

If 3 was not in the list, search would continue until we ended up with a sublist of size 1 where the element in the sublist was not 3.

...
#You can delete the print statements. We've included them so you can trace execution.

def binsearch(x, numberList):
    
    low = 0    # index
    high = len(numberList) - 1    # " "
    last  " " " " "
    
    while low <= high:
        # when low == high the sublist size is 1
        print(numberList[low:high+1]," low:",low," high:",high)
        
        mid = (low + high) // 2    # get index of "middle" element in list
        print("mid = ",mid)
        
        if (x == numberList[mid]):       # is there a match?
            
            print("Found", x," at index", mid);
            
            return mid                   # if yes, return the index it was found in
        
        elif (x < numberList[mid]):     # should we search the left sublist?
            
            print("Search the left sublist")
            
            high = mid - 1                # if so, update high, leave low the same
        
        else:
            
            print("Search the right sublist")
            
            low = mid + 1                # else we must search the right sublist
            
            update low, leave high the same
        
    print("Not in list!")
    
    return (-1)                        # we did not return mid in the loop; if we reached here it means
size 1 with no match. Hence Hence # we must return -1 to show that the
binsearch failed to find x

```python
def main():

    nlist = [1, 3, 4, 7, 9, 11, 21, 27, 30, 31, 35, 39, 41, 52, 56, 57, 59, 61, 64]

    binsearch(3, nlist)
```

#____________________ Analysis______________________________________________

# How do we count the number of comparisons made by binsearch() before it either returns
# a position for x (success) or returns -1 (failure) ?

... Let's make things simple by assuming the size of the list is $2^n$ (2 to the power n).

To make it simpler still, let $n = 4$; the list size is now 16. Let's also assume $x$ is not in list.

positions: [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15] # note, the numbers are POSITIONS!

pass 1: low = 0, high = 15

    mid = 7. After comparing $x$ to list[7], if there is no match, we must go to the
    left or right sublist, depending on whether $x < \text{list}[7]$ or not.
    
    Suppose $x > \text{list}[7]$. We must now look at right sublist.
positions: [8 9 10 11 12 13 14 15]

pass 2: low = 8, high = 15

    mid = 11. After comparing $x$ to list[11] .... say we go to right sublist.
positions: [12 13 14 15]

pass 3: low = 12, high = 15

    mid = 13. After comparing x to list[13]...... say we go to right sublist again.

positions: [14 15]

pass 4: low = 14, high = 15

    mid = 14. After comparing x to list [14] ... say we go to right sublist again

positions: [15]

pass 5: low = 15, high = 15

    mid = 15. On comparing x to list[15] we find that x is not in list.

Analysis: list of size $2^4$ required 4 comparisons (actually 5, but we can ignore that last step. Why? Because $2^n$ for very large $n$ requires $n$ comparisons, and 1 extra step does not matter since $n$ is large).

idea: list of size $2^n$ requires $n$ comparisons.

How many comparisons does a list of size $n$ need?

size $2^n$ ------------------------> needs $\log_2(2^n) = n$ comparisons

size $n$ ------------------------> needs $\log_2(n)$ comparisons

Hence: Binary search is a $O(\log(n))$ algorithm. It is much faster than the $O(n)$ required by linear search. [But remember that binary search had an advantage --- the list was already sorted!]