Lecture 6 - More Lists, Strategies for writing functions

Week 3 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

More on Lists

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 [1]:
```
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 functi
on.
# 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 [2]:
```
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 [3]:
```
fam
```

Out[3]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89, 'me', 1.8]
```

In [4]:
```
fam.append(['miles', 1.78, 'joe', 1.8]) # append will add the entire object as one list entry
```

In [5]:
```
fam
```

Out[5]:
```
['liz',
 1.73,
 'emma',
 1.68,
 'mom',
 1.71,
 'dad',
 1.89,
 'me',
 1.8,
 ['miles', 1.78, 'joe', 1.8]]
```

In [6]:
```
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[6]:
```
['liz',
 1.73,
 'emma',
 1.68,
 'mom',
 1.71,
 'dad',
 1.89,
 'miles',
 1.78,
 'joe',
 1.8]
```

In [7]:
```
fam
```

Out[7]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

- `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 [8]:

```
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 an
d mom
print(fam)
```

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

In [9]:

```
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 [10]:

```
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-10-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 [11]:
```
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 [12]:
```
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 sp
ecific 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 [13]:
```
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 [14]:

```
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 ou
t
print(j)
print(fam)
```

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

In [15]:

```
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.

In [16]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam.index("emma")
```

Out[16]: 2

In [17]:
```
fam.index(3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-63a35f148e9b> in <module>
----> 1 fam.index(3)

ValueError: 3 is not in list
```

In [18]:
```
letters = ["a", "b", "c", "a", "a"]
print(letters.count("a"))
```

```
3
```

In [19]:
```
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 [20]:
```
fam.reverse()  # no output to 'capture', the list is changed in place
```

In [21]:
```
print(fam)
```

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

In [22]:
```
fam.sort()  # can't sort floats and string
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-b0f2370e264b> in <module>
----> 1 fam.sort()  # can't sort floats and string

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

In [23]:
```
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 [24]:
```
print(some_digits)  # preserves numeric data types
```

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

In [25]:
```
type(some_digits[4])
```

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

In [26]:
```
some_digits.sort(reverse = True)
print(some_digits)
```

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

In [27]:
```
some_digits = [4,2,7,9,2,5.1,3]
sorted(some_digits)  # sorted will return a sorted copy of the list
```

Out[27]:
```
[2, 2, 3, 4, 5.1, 7, 9]
```

In [28]:
```
some_digits  # the list is unaffected
```

Out[28]:
```
[4, 2, 7, 9, 2, 5.1, 3]
```

Coding Strategy: Incremental Development

Chapter 6 - Think Python

Develop your code incrementally.

Don't try to get everything working all at once.

Add one or two lines of code at a time. Check to make sure it works with different test cases. Move to add another piece.

Example: Write a distance function

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

In [29]:
```
# start with a basic function definition:
def distance(x1, y1, x2, y2):
    return 0.0
```

In [30]:
```
distance(1, 2, 4, 6)
```

Out[30]:
```
0.0
```

The function doesn't compute the distance, but it runs without errors.

We add a few lines to computer some intermediate values. Using intermediate values are always helpful. It eases the mental burden of having to parse values, especially if you give your intermediate values good variable names.

In [39]:
```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

In [40]:
```
# we test the function. We know that dx should be 3 and dy should be 4
distance(1, 2, 4, 6)
```

```
dx is 3
dy is 4
```

Out[40]:
```
0.0
```

The results match our expectation.

In [41]:
```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is:', dsquared)
    return 0.0
```

In [42]:
```
distance(1, 2, 4, 6) # check to see if we get our expected value
```

```
dsquared is: 25
```

Out[42]:
```
0.0
```

In [43]:
```
import math
```

In [44]:
```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

In [45]:
```
distance(1, 2, 4, 6)
```

Out[45]:
```
5.0
```

The `print` statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is called scaffolding because it is helpful for building the program but is not part of the final product.

1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.
2. Use variables to hold intermediate values so you can display and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

In [52]:
```
# more concise, but harder to read
def distance(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
```

In [53]:
```
distance(1, 2, 4, 6)
```

Out[53]:
```
5.0
```

Coding Strategy: Composition

Write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`.

We can break this into a couple steps:

Find the radius using our distance function.

```
radius = distance(xc, yc, xp, yp)
```

Find the area of a circle given the radius, which we will need to define.

```
result = area(radius)
```

In [54]:
```
def area(radius):
    result = math.pi * radius ** 2
    return result
```

In [55]:
```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

In [56]:
```
# we can choose to make our code more concise if we wish. But this is not necessar
y.
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

In [58]:
```
circle_area(0, 0, 3, 4)
```

Out[58]:
```
78.53981633974483
```

In [59]:
```
circle_area(0,0, math.sqrt(2), math.sqrt(2))
```

Out[59]:
```
12.566370614359172
```

Useful idea: Boolean Functions

Boolean functions are function that return either `True` or `False`

Convention says that these functions should be written as a yes/no question like `is_divisible(x,y)`

In [60]:
```
# example
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

In [61]:
```
is_divisible(6, 4)
```

Out[61]:
```
False
```

In [62]:
```
is_divisible(6, 3)
```

Out[62]:
```
True
```

Because the result of the `==` operator is boolean, we can simplify by simply returning the expression directly.

In [63]:
```
# more concise and readable
def is_divisible(x, y):
    return x % y == 0
```

We can use the result in conditional statements like:

In [64]:
```
x = 6
y = 3
if is_divisible(x, y):
    print('x is divisible by y')
```

```
x is divisible by y
```

Don't write a statement that is equivalent to `if True == True:`

That extra comparison is unnecessary.

In [65]:
```
# if is_divisible(x, y): is much preferred
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

```
x is divisible by y
```

Revisit the Factorial function

Let's write the factorial function incrementally:

First we simply define the function. Right now it does nothing.

In [66]:
```
def factorial(n):
    pass
```

Let's add the base case. If the argument is 0, we return 1.

In [67]:
```
def factorial(n):
    if n == 0:
        return 1
```

In [68]:
```
factorial(0)
```

Out[68]:
```
1
```

In [69]:
```
factorial(4) # returns nothing because we haven't written the code yet.
```

If we have anything other than 0, we make a recursive call.

We keep in mind that 4! = 4 3!, and more generally n! = n (n-1)!

In [70]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n - 1)
        result = n * recurse
        return result
```

In [71]:
```
factorial(4)
```

Out[71]:
```
24
```

Debugging your recursive code

When you are trying to write a recursive function, you can quickly run into errors and will want some strategies to debug.

One simple method is to add print statements declaring what is being done.

In the following function, we add a print statement preceeded by spaces to indicate the steps.

We put a print statement at the beginning of the function. We also put a print statement before the `return` statement.

In [72]:
```
def factorial(n):
    spaces = " " * (4 * n)
    print(spaces, 'factorial', n) # print statement for function call
    if n == 0:
        print(spaces, 'returning 1') # print statement for return values
        return 1
    else:
        recurse = factorial(n - 1)
        result = n * recurse
        print(spaces, 'returning', result) # print statement for return values
        return result
```

In [73]:
```
factorial(4)
```

```
                    factorial 4
                factorial 3
            factorial 2
        factorial 1
    factorial 0
    returning 1
        returning 1
            returning 2
                returning 6
                    returning 24
```

Out[73]: 24

Checking for input types

It is often important to check input types of a function so that executing the function does not cause errors.

The function `isinstance(x, type)` can be used to check if object x is of a particular type.

In [74]:
```
isinstance(5, int)
```

Out[74]: True

In [75]:
```
isinstance(5.3, float)
```

Out[75]: True

In [77]:
```
isinstance(False, bool)
```

Out[77]: True

In [78]:
```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative values.')
        return None
    elif n == 0:
        return 1
    else:
        recurse = factorial(n - 1)
        result = n * recurse
        return result
```

In [79]:
```
factorial(1.2)
```

```
Factorial is only defined for integers.
```

In [80]:
```
factorial(-3)
```

```
Factorial is not defined for negative values.
```