Homework 2

Due Sept 14, 2020, 4PM EST

This assignment is all about functions. DRY! (Don't Repeat Yourself)

Get comfortable with this important feature

In [9]:
```
# Fill in your name

first_name = 'Shariq'
last_name = 'Jamil'

assert len(first_name) != 0, "First name is blank"
assert len(last_name) != 0, "Last name is blank"
```

Problem 1: Make HTML elements

HTML uses 'tags' to markup webpages. You can read about them here

https://www.geeksforgeeks.org/most-commonly-used-tags-in-html/ (https://www.geeksforgeeks.org/most-commonly-used-tags-in-html/)

We aren't ready to pull a webpage apart, but we can start to paste one together.

Write a function make_tag() which takes two strings, 'tag' and 'body', and returns an HTML element.

We can use Python's Type Hints to describe the parameters and return value.

```
def make_element(tag: str, body: str) -> str:
```

For example, if the tag was 'i' and the body was 'very', you would return the string for putting *very* in italics

```
'<i>very</i>'
```

Fill in your function definition in the cell below. You could solve this with string addition, but f-string formatting is more powerful. Here is one discussion of f-strings formatting: https://www.geeksforgeeks.org/formatted-string-literals-f-strings-python/

In [10]:
```
"""
This function will take in two strings: 'tag' and 'body'
It will return 'tag' as an HTML tag with 'body' nested within the tag
More info: https://www.geeksforgeeks.org/most-commonly-used-tags-in-html/
"""
def make_element(tag: str, body: str) -> str:
    # format the opening tag
    opening_tag = '<'+tag+'>'
    # format the closing tag
    closing_tag = '</'+tag+'>'
    # return a string with 'body' nested within the two strings created above
    return opening_tag + body + closing_tag
```

Run the cell below to call your function.

In [11]:
```
make_element('i', 'very')
```

Out[11]:
```
'<i>very</i>'
```

Unit tests for make_tags()

In [12]:
```
def validate_make_tag():
    assert make_element('i', 'very') == '<i>very</i>', "Expected '<i>very</i>'"
    assert make_element('html', 'In production') == '<html>In production</html>', "
Expected '<html>In production</html>'"
    assert make_element('head', 'Tale') == '<head>Tale</head>', "Expected '<head>Ta
le</head>'"
    assert make_element('title', 'Heavyweight Champion') == '<title>Heavyweight Cha
mpion</title>', "Expected '<title>Heavyweight Champion</title>'"

    print('Success!')

validate_make_tag()
```

```
Success!
```

Problem 2: Sleep In

When I am working, I get up at 6AM.

I sleep in on vacation and I sleep in on the weekends.

Write a Boolean function which takes two Boolean parameters, weekday, and vacation.

```
def sleep_in(weekday: bool, vacation: bool) -> bool:
```

You should return **True** if I would sleep in, and **False** if I need to work and get up early.

Hint: *Your function should return a Boolean expression based on the two Booleans, **weekday** and **vacation**.*

Fill in your function definition in the cell below.

In [13]:
```
def sleep_in(weekday: str, vacation: str) -> bool:
    # Sleep in if
        # you are on vacation OR its not a weekday
    return (vacation or (not weekday))
```

Unit tests for sleep_in()

Run the cell below to call your function.

In [14]:
```
def validate_sleep_in():
    assert sleep_in(True, True), "I can sleep on Vacation!"
    assert sleep_in(False, True),"I can sleep on the weekends and on Vacation!"
    assert not sleep_in(True, False), "I have to get up during the work week!"
    assert sleep_in(False, False), "I can sleep on the weekends!"

    print('Success!')

validate_sleep_in()
```

```
Success!
```

Problem 3: Justify yourself!

Your goal is to right justify a line of text.

Write a function named right_justify() that takes a string named text as a parameter and returns a string of length 70 that begins with blanks and ends with the string text. See the example in the comments below.

```
def right_justify(text: str) -> str:
        pass
```

Hint: Use string concatenation and repetition and the Python function len(). Do not use loops or other ideas we haven't discussed yet.

Fill in your function definition in the cell below.

In [15]:
```
# right_justify('monty') should return a string of length 70, ending with monty:
# returns: '                                                               monty'
def right_justify(text):
    # find out how many blanks are needed
    blanks_needed = 70 - len(text)
    # create a string with the required number of blanks
    fill_blanks = (' ')*blanks_needed
    # return the string of blanks with the given text appended
    return fill_blanks + text
```

Does it work?

Devise three interesting test cases to convince yourself that your function works properly.

What strings might make your functions's task difficult?

Run the two cells below to test your function.

In [16]:
```
# Print a ruler to check your results
print('0123456789' * 7)

# Test your routine on my example
print(right_justify('monty'))

# Add your test cases below - replace my strings with more interesting test cases
print(right_justify('one * one'))
print(right_justify('/ / ! #'))
print(right_justify('45/75* 9'))
```

```
01234567890123456789012345678901234567890123456789012345678901234567890123456789
                                                                          monty
                                                                    one * one
                                                                      / / ! #
                                                                     45/75* 9
```

Unit tests for justify()

Now run the Unit Tests to check your work

In [17]:
```
def validate_justify(before: str):
    after = right_justify(before)
    assert len(after) == 70, f"Wrong length {len(after)} after justifying {before}"
    if (before):
        assert before == after[-len(before):], f"Should find '{before}' on right en
d"

def validate_suite():
    validate_justify(' ')
    validate_justify('one')
    validate_justify('two')
    validate_justify('three')
    validate_justify('Monty')
    validate_justify('Project Oxygen shocked everyone by concluding that, among the
eight...')
    validate_justify('')

    print('Success!')

validate_suite()
```

```
Success!
```

Problem 4: Mechanical Metathesis

Write a function that confounds two strings in hopes of comic effect by swaping the first two characters of each string.

```
def metathesis(word1: str, word2: str) -> str:
        pass
```

You may assume that both words are at least two characters long.

metathesis('dog', 'cat') should return 'cag dot'.

Fill in your function definition in the cell below.

In [18]:
```
'''
This function swaps the first two characters
of the given strings
'''
def metathesis(word1, word2):
    # store the first two characters of word1
    first_two_w1 = word1[0:2]
    # store the first two characters of word2
    first_two_w2 = word2[0:2]
    # store the rest of word1
    rest_w1 = word1[2:]
    # store the rest of word2
    rest_w2 = word2[2:]

    # return a string with both words with the first two characters
    # replaced with those of the other
    return first_two_w2 + rest_w1 + ' ' + first_two_w1 + rest_w2
```

Try your function

In [19]:
```
print(metathesis('dog', 'cat'))         # should print 'cag dot'
print(metathesis('hot', 'tuna'))        # should print 'tut hona'
print(metathesis('spinach', 'bread'))   # should print 'brinach spead'
```

```
cag dot
tut hona
brinach spead
```

Unit tests for metathesis

Run the cell below to test your function.

In [20]:
```
def validate_metathesis():
    assert metathesis('dog', 'cat') == 'cag dot', f"Did not handle 'dog', 'cat'"
    assert metathesis('hot', 'tuna') == 'tut hona', f"Did not handle 'hot', 'tuna'"
    assert metathesis('to', 'be') == 'be to', f"Did not handle 'to', 'be'"
    assert metathesis('pizza', 'pie') == 'pizza pie', f"Did not handle 'pizza', 'pi
e'"
    assert metathesis('pasta', 'pomodoro') == 'posta pamodoro', f"Did not handle 'p
asta', 'pomodoro'"

    print('Success!')

validate_metathesis()
```

```
Success!
```

Problem 5: Boxed in

A) Write a function that draws a 2 x 2 grid like this using what we have learned so far.

Don't use loops of any kind - we haven't covered them yet.

```
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
```

B) Write a function that draws a 4 x 4 grid:

```
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
```

This is an exercise in creating reusable components. It should be done using only the statements and other features we have learned so far. That is, no loops of any kind should be used.

Here are two ways to solve this problem. I prefer mine to Downey's

Hint 1: Downey's approach

Downey notes that to print more than one value on a line, you can print a comma-separated sequence of values:

```
print('+', '-')
```

By default, print advances to the next line, but you can override that behavior and put something else at the end, like this:

```
print('+', end='')
    print('-')
```

The output of these statements is '+-' on the same line, as you can see by running the cell below. The output from the next

print statement would begin on the next line.

In [21]:
```
# Example for Hint 1 A
# Produces two lines
print('+')
print('-')

# Add a blank line to set off the next line
print()

# Produces one line
print('+', end='')
print('-')
```

```
+
-

+-
```

Hint 2: Alternative approach

I find it simpler to assemble each line of the output as a string and assemble a long string and print that rather than printing each piece.

Lines are combined to return one string that holds the whole figure, which the caller can print.

Here is an obfuscated example of how this works. Each call to combine() produces a new line of the figure.

It may be hard to predict the outcome, but I hope the method used to assemble the result is clear.

In [22]:
```
def combine(s):
    return s + s[::-1][1:] + "\n"

def kilroy():
    a = "         ||"
    b = "----ooO-(_"
    c = "        (o "

    return combine(a) + combine(c) + combine(b)
```

Predict what kilroy() returns before running the cell below.

In [23]:
```
print(kilroy())
```

```
         |||
        (o o(
----ooO-(_(-Ooo----
```

Building Strings

We will soon learn a better way to add a series of strings together

Hint 3: Can you write one function that solves both 3A and 3B?

Your Solution:

Fill in your function to print a 2x2 grid in the cell below and call it. We expect to see the 2x2 box below the cell.

In [24]:
```
# 5A: Write a function to print a 2x2 grid
'''
A function that takes in numbers of rows and columns and
prints a grid with the corresponding dimensions
'''
def create_grid(rows = 2, columns = 2):
    # store a grid block's border string that is as wide as
    # the columns given
    divider = (('+-----') * columns) + '+' + "\n"
    # store a grid block's body string that is as wide as
    # the columns given
    lines = (('|     ') * columns + '|' + "\n") * 3

    # print a series of lines sandwiched between dividers based
    # on the number of rows given
    print((divider + lines) * rows + divider)

create_grid()
```

```
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
```

Fill in your function to print a 4x4 grid in the cell below and call it. We expect to see the 4x4 box below the cell.

In [25]:
```
# 5B: Write a function to to print a 4x4 grid
'''
A function that takes in numbers of rows and columns and
prints a grid with the corresponding dimensions
'''
def create_grid(rows = 4, columns = 4):
    # store a grid block's border string that is as wide as
    # the columns given
    divider = (('+-----') * columns) + '+' + "\n"
    # store a grid block's body string that is as wide as
    # the columns given
    lines = (('|     ') * columns + '|' + "\n") * 3

    # print a series of lines sandwiched between dividers based
    # on the number of rows given
    print((divider + lines) * rows + divider)

create_grid()
```

```
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
+-----+-----+-----+-----+
```

Problem 6: Show me the way

Import the library **sys**, and get the return value of the sys attribute **path**.

You can read about the sys library here: https://docs.python.org/3.8/library/sys.html (https://docs.python.org/3.8/library/sys.html)

Print your path, the first directory on the path, and the last directory on the path.

Different people will see different answers, so there are no Unit Tests

Hint: Use the same indexing we use for strings to get first and last elements

What directories does Python search?

In [26]:
```
# Write your solution to problem 6 below.  Import sys and
# Print the path, print the first directory, and print the last directory on path
import sys
# print whole path of modules available to Python
print(sys.path)
# print the first directory
print(sys.path[0])
# print the last directory
print(sys.path[len(sys.path)-1])
```

```
['C:\\Users\\tress\\Documents\\Development\\Harvard\\Python\\Homework 2', 'C:\\U
sers\\tress\\anaconda3\\python38.zip', 'C:\\Users\\tress\\anaconda3\\DLLs', '
C:\\Users\\tress\\anaconda3\\lib', 'C:\\Users\\tress\\anaconda3', '', 'C:\\User
s\\tress\\anaconda3\\lib\\site-packages', 'C:\\Users\\tress\\anaconda3\\lib\\sit
e-packages\\win32', 'C:\\Users\\tress\\anaconda3\\lib\\site-packages\\win32\\lib
', 'C:\\Users\\tress\\anaconda3\\lib\\site-packages\\Pythonwin', 'C:\\Users\\tre
ss\\anaconda3\\lib\\site-packages\\IPython\\extensions', 'C:\\Users\\tress\\.ipy
thon']
C:\Users\tress\Documents\Development\Harvard\Python\Homework 2
C:\Users\tress\.ipython
```

Post mortem

How long did it take you to solve this problem set? Did anything confuse you or cause difficulty?

In [27]:
```
# About three hours. Nothing was really confusing but I did have a lot of fun doing
this assignment.
```