Name:______________________________ **Partner:** ____________________________

Python Activity 28: Recursion

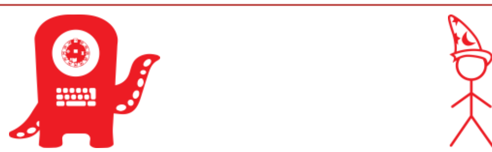
Some solutions are just the same steps repeated again and again on a subset of the input data.

Learning Objectives
Students will be able to:
Content:
- Define **recursion** and explain why it is useful.
- List the **three steps** for building recursive solutions.

Process:
- Predict the output of recursive programs.
- Write code that uses recursion.

Prior Knowledge
- Python concepts: functions, conditionals, data structures, return

Concept Model:
Consider the following story:
There once was a monster and a sorcerer's apprentice. The apprentice was tasked with determining whether a list has any odd numbers (in contrast to *even*, not as in *unusual*), and he needed the monster's help. The apprentice went down to the dungeons and asked, "Monster, I need to know if any of the numbers in this list are odd: [3142, 5798, 6550, 8914]"



The monster, being surly and rather dissatisfied with his dungeon accomodations replied:

"Sorry, I can only tell you if the *first* number of the list is odd."

The apprentice pleaded, "But I need to know if *any* number in the list is odd, not just the first!" Unmoved, the monster retorted,

"Well, I'll only look at the first number, but I'll look at as many lists as you like."

1. What should the sorcerer's apprentice do to solve his problem?

____

____

And so, the sorceror's apprentice presented the monster with the original list.

Apprentice: `[3142, 5798, 6550, 8914]`
Monster: The first number is **not odd**.

And then, the sorcerer's apprentice presented the monster with a modified version of the list:

Apprentice: [~~3142,~~ `5798, 6550, 8914]`
Monster: The first number is **not odd**.

And then, the sorcerer's apprentice presented the monster with a modified version of the list:

Apprentice: **[~~3142, 5798,~~ 6550, 8914]**
Monster: The first number is **not odd**.

Once more, the sorcerer's apprentice presented the monster with a modified version of the list:

Apprentice: **[~~3142, 5798, 6550,~~ 8914]**
Monster: The first number is **not odd**.

Finally, the sorcerer's apprentice presented the monster with a modified version of the list:

Apprentice: **[~~3142, 5798, 6550, 8914~~]**
Monster: That's an **empty list**, it can't be odd!

Quite satisfied with the monster's input, the apprentice smiled and remarked, "Ah, so none of the numbers are odd, thank you!" The monster responded, "But how can you know that, I only told you if the first number was odd!"

2. How does the apprentice know that all the numbers are not odd?

____

____

The apprentice replied, "I gave you the following sub-lists of my original list, and you gave me an answer for the first item in each:"

```
[3142, 5798, 6550, 8914]
[      5798, 6550, 8914]
[            6550, 8914]
[                  8914]
                       []
```

The monster simply grumbled, "It looks like you've discovered recursion."

3. How does the monster know when to stop checking values?

____

____

4. What is the process the monster repeatedly executes?

____

____

5. How does the apprentice get the monster to proceed to the next number?

____

____

6. Based on this story, how might you define recursion?

____

____

FYI: ***Recursion*** is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. There are three steps to consider when building a recursive problem solution:

1. What is the stopping condition / base case?
2. What is the small, repeated step?
3. How do we break the journey down into a smaller piece?

Critical Thinking Questions:

1. Examine the sample code below from interactive python which represents the problem & solution from the Concept Model story above:

Interactive Python

```
0 >>> def list_has_odd(lst):
1 ...       if len(lst) < 1:
2 ...            return False
3 ...       elif lst[0]%2 != 0:
4 ...            return True
5 ...       return list_has_odd(lst[1:])
6 >>> mylist = [3142, 5798, 6550, 8914]
7 >>> print("Has odd number?", list_has_odd(mylist))
```

a. What does each line of code do?

0 ____

1 ____

2 ____

3 ____

4 ____

5 ____

6 ____

7 ____

b. On which line(s) is the stopping condition? ____________________

c. On which lines are the small repeated steps? ____________________

d. On which line is the journey broken down into smaller pieces? __________

e. Which lines might be said to be the Monster's actions? ________________

f. Which lines might be said to be the Apprentice's actions? ________________

g. When `mylist = [3142, 5798, 6550, 8914]`, what is the argument passed to `list_has_odd(..)` each time it's called on line 5?

On line 7:________________ **Third time (5):**________________

First time (5):________________ **Fourth time (5):**________________

Second time (5):________________

h. If `mylist = [0, 7, 9, 4, 2]`, how many times will `list_has_odd(..)` be called? What will the arguments passed to the recursive call each time be?

____

____

____

____

____

2. Examine the sample code below from interactive python which contains another recursive function:

Interactive Python

```
0 >>> def power(a, n):
1 ...     if n == 0:
2 ...             return 1
3 ...     else:
4 ...             return a * power(a, n-1)

5 >>> print(power(5, 0))
6 1
7 >>> print(power(5, 4))
8 625
```

a. What does each line of code do?

0 ____

1 ____

2 ____

3 ____

4 ____

5 ____

7 ____

b. On which line is the stopping condition? ___________________________

c. On which line(s) are the small repeated steps? ___________________________

d. On which line is the journey broken down into smaller pieces? ______________

e. When the initial arguments passed to `power(..)` are `5` and `4`, as on line 7, fill out what happens on line 4 each time `power(..)` is called?

On line 7: `power(`___, ___`)`

First (4): `return` ___ `* power(`___, ___`)`

Second (4): `return` ___ `* power(`___, ___`)`

Third (4): `return` ___ `* power(`___, ___`)`

Fourth (4): `return` ___ `* power(`___, ___`)`

f. When the initial arguments passed to `power(..)` are `5` and `0`, as on line 5, how many times is the `power(..)` function called? ______________________ times

h. Compare the number of function calls to `power(..)` in (e) and (f). Why are these numbers different?

____

3. Examine the sample code below and its corresponding output:

mystery1.py

```
0 def mystery1(n):
1    if n < 1:
2          return 0
3    else:
4          print(n)
5          return mystery1(n-1)

6 if __name__ == '__main__':
7     print(mystery1(5))
```

Output

```
5
4
3
2
1
0
```

a. What does each line of code do?

0 ____

1 ____

2 ____

3 ____

4 ____

5 ____

7 ____

b. On which line is the stopping condition? ______________________

c. On which lines are the small repeated steps? ______________________

d. On which line is the journey broken down into smaller pieces? ____________

e. When the initial argument passed to `mystery1` is `5`, as on line 7, what is passed to `mystery1(..)` as an argument each time it's called on line 5?

On line 7: `mystery1(`___`)`

First (5): `return mystery1(`___`)`

2nd (5): `return mystery1(`___`)`

3rd (5): `return mystery1(`___`)`

4th (5): `return mystery1(`___`)`

5th (5): `return mystery1(___)`

f. When the initial argument passed to `mystery1` is `5`, as on line 7, what is returned by `mystery1(..)` each time it's called on line 5?

1st time (5): `return mystery1(`__________________`)` returns ____________

2nd time (5): `return mystery1(`__________________`)` returns ____________

3rd time (5): `return mystery1(`__________________`)` returns ____________

4th time (5): `return mystery1(`__________________`)` returns ____________

5th time (5): `return mystery1(`__________________`)` returns ____________

g. What might `mystery1(4)` return? ________________

h. What does the `mystery1(n)` function do?

____

4. Examine the sample code below and its corresponding output, which is similar to the previous question:

mystery2.py

```
0 def mystery2(n):
1    if n < 1:
2        return 0
3    else:
4        result = mystery2(n-1)
5        print(n)
6        return result
7 if __name__ == '__main__':
8     mystery2(5)
```

Output

```
1
2
3
4
5
```

a. Circle the code in `mystery2(..)` that differs from `mystery1(..)`. What do these new lines of code do?

Line _______: ____

Line _______: ____

Line _______: ____

Line _______: ____

b. On which line(s) is the stopping condition? __________________________

c. On which lines are the small repeated steps? ____________________________

d. On which line is the journey broken down into smaller pieces? ______________

e. When the initial argument passed to `mystery2` is `5`, as on line 8, what is returned by `mystery2(..)` each time it's called on line 4?

1st time (5): `return mystery1(`__________________`)` returns ____________

2nd time (5): `return mystery1(`__________________`)` returns ____________

3rd time (5): `return mystery1(__________________)` returns __________

4^{th} time (5): `return mystery1(__________________)` returns __________

5^{th} time (5): `return mystery1(__________________)` returns __________

g. What might `mystery2(4)` return? ______________

h. If we update line 8 to `print(mystery2(n))`, the output is the same as above, but a 0 appears after the 5 that is displayed. Why might that be? (*Hint: Where is* `result` *printed?)*

____

i. What does the `mystery2(n)` function do?

____

5. ***Fibonacci Sequences*** appear throughout nature, as in the branching of trees, fruitlets of a pineapple, the flowering of an artichoke, among many other examples[1]. Fibonacci numbers are often defined by a recursive relationship:

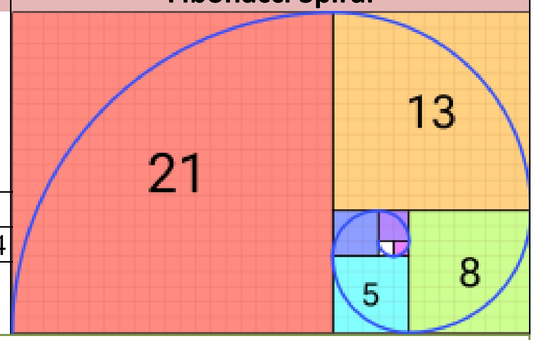
Fibonacci Numbers Definition

$F_0 = 0, F_1 = 1,$
and
$F_n = F_{n-1} + F_{n-2}$

The first 10 Fibonacci numbers: F_1

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}
0	1	1	2	3	5	8	13	21	34	55	89	144

Fibonacci Spiral



If we wish to find the *6*th Fibonacci number, we would perform the following calculations:

$F_6 =$
$= F_{6-1} + F_{6-2}$
$= (F_{5-1} + F_{5-2}) + (F_{4-1} + F_{4-2})$
$= ((F_{4-1} + F_{4-2}) + (F_{3-1} + F_{3-2})) + ((F_{3-1} + F_{3-2}) + (F_{2-1} + F_{2-2}))$
$= (((F_{3-1} + F_{3-2}) + (F_{2-1} + F_{2-2})) + ((F_{2-1} + F_{2-2}) + F_1)) + (((F_{2-1} + F_{2-2}) + F_1) + (F_1 + F_0))$
$= ((((F_{2-1} + F_{2-2}) + F_1) + (F_1 + F_0)) + ((F_1 + F_0) + F_1)) + (((F_1 + F_0) + F_1) + (F_1 + F_0))$
$= ((((F_1 + F_0) + F_0) + F_1) + (F_1 + F_0)) + ((F_1 + F_0) + F_1)) + (((1 + 0) + 1) + (1+0))$
$= ((((1 + 0) + 0) + 1) + (1+0)) + ((1 + 0) + 1)) + (((1 + 0) + 1) + (1+0))$
$= \mathbf{8}$

a. What might be the ***base case*** or *stopping condition* for calculating the `n`th Fibonacci number? ______________________________________

b. What might be the small, repeated steps? ______________________

c. How might we break down the journey into one small step and a smaller journey?

____

[1] https://en.wikipedia.org/wiki/Fibonacci_number#Nature

d. Complete the recursive function, `fibonacci(n)`, below such that it will print the `n`th Fibonacci value for a given number, `n`.

```
def fibonacci(n):
    # base case 1

    # base case 2

    # step + smaller journey



if __name__ == "__main__":
    print(fibonacci(6)) # should print 8
    print(fibonacci(7)) # should print 13
```

Application Questions: Use the Python Interpreter to check your work

1. Write a recursive function, `recursive_list_length(any_list)`, that will return the length of a given list, `any_list`, using recursive means:

2. What does the following program do? (Step-through with examples for `fi,in_sequence`).

```
def mystery3(fi, in_sequence):
    if not in_sequence:
        return False
    elif fi == in_sequence[0]:
        return True
    else:
        return False or mystery2(fi, in_sequence[1:])
```

3. Write a recursive function, `count_char(ch, any_string)`, that will count the number of occurrences of a given character, `ch`, in a given string, `any_string`, using recursive means:

4. Write a recursive function, `get_item(index, start, any_list)`, that will return the element located at `index`, in a given list, `any_list`, using recursive means. It should start looking at the `index` provided in `start` and should return `None` if that index is not found: