Folks, this is a brand new activity. If you encounter any issues/typos, please let Iris know!

Name:_____ **Partner:** _____

Python Activity 30: Recursive Function Frames

Using the function frame model can help us understand how recursion works.

Learning Objectives

Students will be able to:

Content:

- Summarize how the function frame stack works
- Describe how the function frame stack works for specific functions

Process:

- Predict the output of recursive programs.

Prior Knowledge

- Python concepts: recursion

Concept Model:

*Consider the concept of **factorial**:*

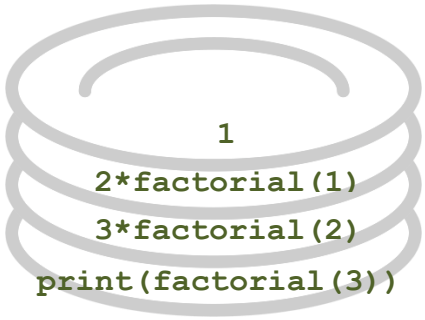
The factorial of a non-negative integer, `n`, denoted by `n!`, is the product of all positive integers less than or equal to `n`. The factorial of `n` also equals the product of `n` with the next smaller factorial: i.e. n! = n * (n-1) * (n-2) * (n-3) * …3 * 2 * 1 4! = 4 * 3 * 2 * 1 = 24

n! = n * (n-1)! 5! = 5 * 4! = 120

We can write this recursively in Python with the following code:

factorial.py

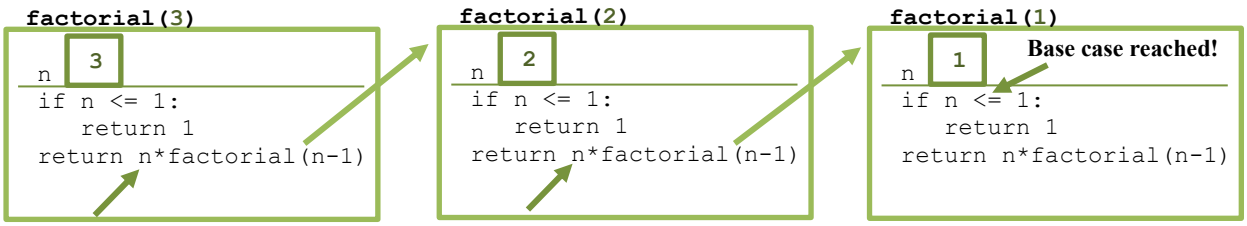
```
0 def factorial(n):
1     if n <= 1:
2         return 1
3     return n * factorial(n-1)

4 if __name__ == '__main__':
5     print(factorial(4)) # prints 24
6     print(factorial(3)) # prints 6
```



1. First, identify the important recursion steps in this example code:
   a. On which line is the stopping condition? _____
   b. On which line is the small repeated step? _____
   c. On which line is the journey broken down into smaller pieces? _____

When Python executes a function, it creates a frame for all the variables created in that function. Whenever a function calls another function, it waits until that function returns an answer before continuing (and that function call is replaced with the answer it returns). We call this a ***function frame stack***, as elements in a stack are added or removed from the top of the stack to the bottom, such as with a *stack of plates*.

In this example, when `print(factorial(3))` is first called on line 5, a new function frame is made for the `factorial(3)`, and nothing will be printed until that frame is executed completely (i.e., returns a value):



However, we see inside the function frame for `factorial(3)` that we have another call to `factorial(..)`, with `factorial(n-1)`. This creates a new function frame for `factorial(2)`, and the function frame for `factorial(3)` will not return until that new function frame is executed. Inside `factorial(2)`, the additional call to `factorial(n-1)` creates a new function frame for `factorial(1)`. Within the function frame for `factorial(1)`, we reach the base case and return the value `1`.

When the `1` is returned it replaces the call to `factorial(n-1)` in the function frame for `factorial(2)`. This in turn allows the function frame for `factorial(2)` to return the value `2*1`.

When the `2` is returned, it replaces the call to `factorial(n-1)` in the function frame for `factorial(3)`. This in turn allows the function frame for `factorial(3)` to return the value `3*2*1`.

When the `6` is returned, it replaces the call to `factorial(3)` in the function frame for `factorial.py`. This in turn allows the `print(..)` function to display the final value of `6`.

2. Describe how the function frame stack would be different for a call to `factorial(4)`:

FYI: When we return from a ***function frame***, "control flow" goes back to where the function call was made. The function frame, and the local variables inside it, *are destroyed after the return*. If a function does not have an *explicit* return statement, it returns `None` after all statements in the function body are executed. The return value replaces the function call.

Critical Thinking Questions:

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

count_down.py	Output
0 def count_down(n): 1 if n < 1: 2 return 0 3 else: 4 print(n) 5 return count_down(n-1) 6 if __name__ == "__main__": 7 count_down(5)	5 4 3 2 1

   a. Identify the recursive steps:

      On which line is the base case? _____

      On which lines are the small repeated steps? _____

      On which line is the journey broken down into smaller pieces? _____

   b. What might `count_down(4)` *return*? _____

   c. What might `count_down(4)` *print*?

   d. Draw a function frame stack diagram for a call to `count_down(3)`, similar to what we did for `factorial(3)` in the *Concept Model* example above. As there's a `print(..)` call in our recursive function, you should also keep track of what is printed (and when)!

Output

   e. How many function frames are created? _____

      (*Hint: How many function calls to* `count_down(n)` *does Python make?*)

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

countUp.py

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

Output

```
1
2
3
4
5
```

a. Circle the code in `count_up(..)` that differs from `count_up(..)`.

b. Identify the recursive steps:

On which line(s) is the stopping condition? _____

On which lines are the small repeated steps? _____

On which line is the journey broken down into smaller pieces? _____

c. What might `count_up(4)` return? _____

d. What might `count_up(4)` print?

e. Draw a function frame stack diagram for a call to `count_up(3)`, similar to what we did for `factorial(3)` in the *Concept Model* example above. As there's a `print(..)` call in our recursive function, you should also keep track of what is printed (and when)!

Output

f. How many function frames are created? _____

(*Hint: How many function calls to* `count_up(n)` *does Python make?*)