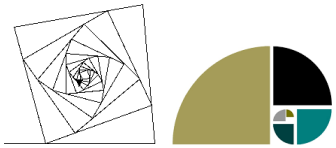
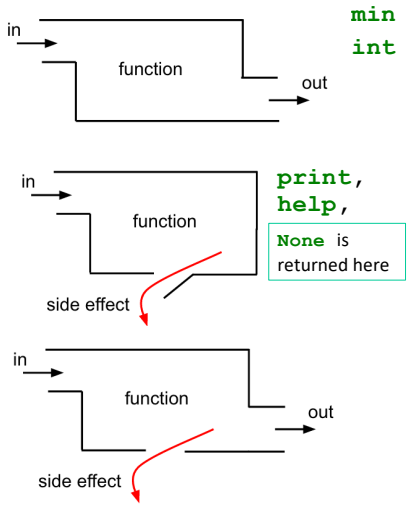
Fruitful Recursion
Turtle Recursion



CS111 Computer Programming

Department of Computer Science
Wellesley College

Recall: fruitful functions and side effects



Functions can return a value using the keyword **return**. That value is returned to whoever called the function.

Functions can have **side effects** such as printing (remember, this is not the same as returning). Regardless of whether a function has a side effect, it will always return a value. **None** will be provided as a default. Generally, the function caller will not use the returned **None**. Thus, we write this situation without an out as shown in second picture. The last picture depicts a scenario in which a function has a side effect AND returns a value other than **None**.

Sum of numbers from 1 to n

- Recall `countUp(n)` for printing integers from 1 up to n:

```
def countUp(n):
    if n <= 0:
        pass
    else:
        countUp(n-1)
        print(n)
```

- How would we define a function `sumUp(n)` that **returns** the sum of integers from 1 through n?

Thinking Box
In a normal function, we would use an accumulator variable that starts at 0 to keep track of the amount being accumulated (e.g., a sum). Would it make sense to have such a variable in a recursive function? Explain. Use the call frame model to verify your answer.

How to write recursive functions? Wishful thinking! (for the recursive case)

1. Consider a relatively small **concrete example** of the function, typically of size n = 3 or n = 4. What should it return?

   In this case, `sumUp(4)` should return 4 + 3 + 2 + 1 = 10

2. Without even thinking, apply the function to a smaller version of the problem. By *wishful thinking*, assume this "just works".

   In this case, `sumUp(3)` should return 3 + 2 + 1 = 6.

3. What glue can be used to combine the arguments of the big problem and the result of the smaller problem to yield the result for the big problem?.

   In this case, `sumUp(4)` should return `4 + sumUp(3)`

4. Generalize the concrete example into the general case:

   In this case, `sumUp(n)` should return `n + sumUp(n-1)`

What about the base case?
Use the recursive case for the penultimate input

For example, what should **`sumUp(0)`** return?

1. According to the recursive case:

   `sumUp(n)` should return **`n + sumUp(n-1)`**

2. Specialize the recursive case to the penultimate (next to last) input:

   `sumUp(1)` should return **`1 + sumUp(0)`**

3. Decide what should be returned for the penultimate input.
   In this case, **`sumUp(1)`** should clearly return 1.

4. Deduce what should be returned for the base case.

   `sumUp(1)` equals **`1`** equals **`1 + sumUp(0)`**,
   so **`sumUp(0)`** should return **`0`**

Here, 0 is the **identity value** for **+**. **Fruitful base cases are often identity values.**

Defining `sumUp`

```
def sumUp(n):
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

Compare this to **`countUp`**`(n)`:

```
def countUp(n):
    if n <= 0:
        pass
    else:
        countUp(n-1)
        print(n)
```

Thinking Box
The solution didn't use an accumulator variable that started at 0 to store the sum. Does that mean that we cannot use local variables in a recursive function? Do you think the following function will work? Explain.

```
def sumUp(n):
    if n <= 0:
        return 0
    else:
        sumSoFar = n + sumUp(n-1)
        return sumSoFar
```

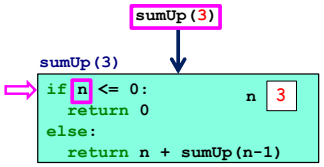

Call frame model for `sumUp(3)`

`sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

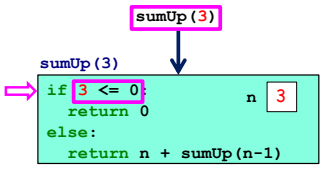

Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



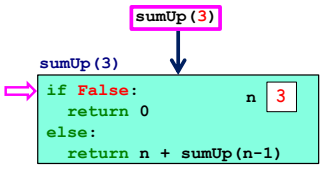
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



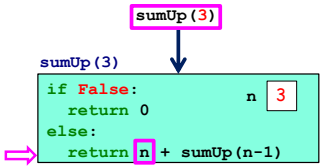
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



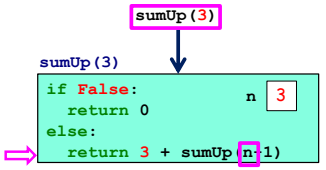
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



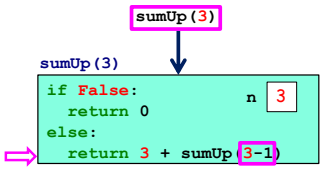
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



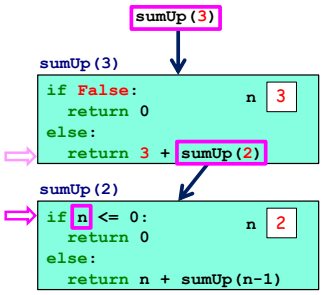
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



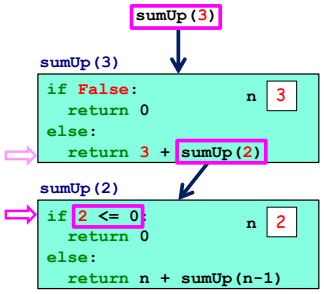
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



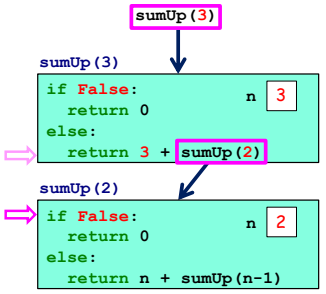
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



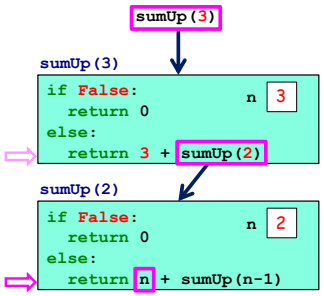
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



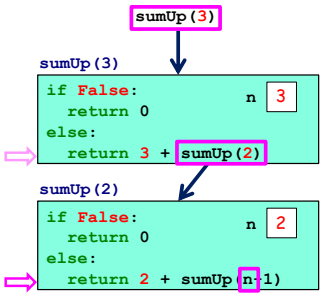
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



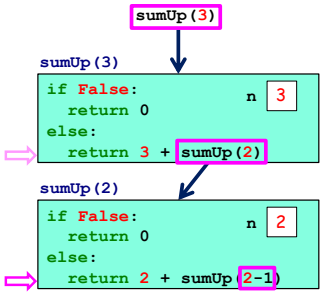
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



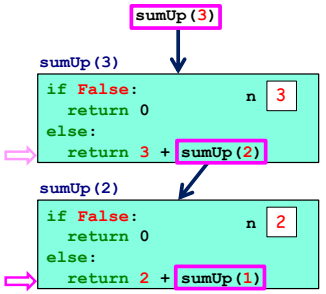
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



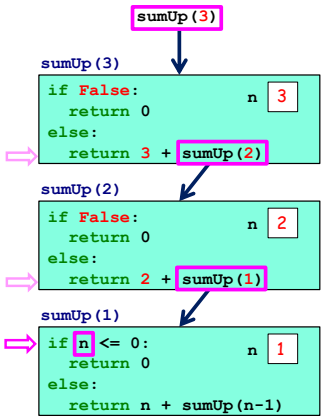
Call frame model for `sumUp(3)`

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



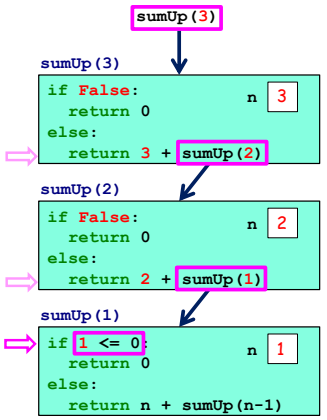
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



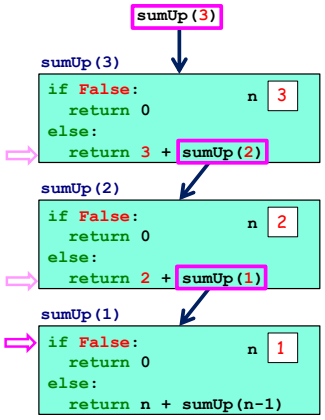
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



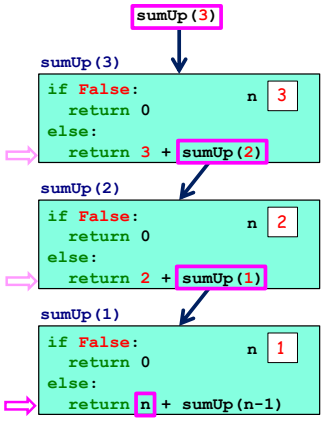
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



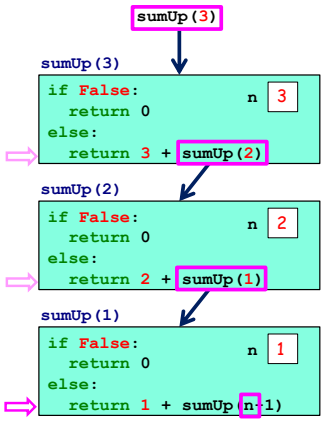
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



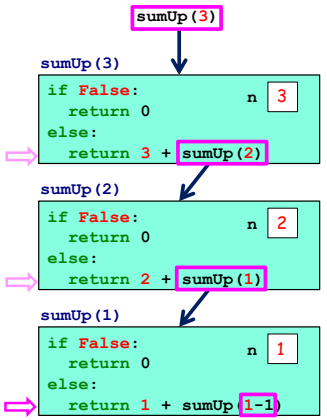
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



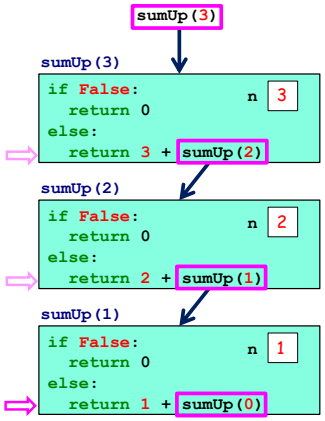
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



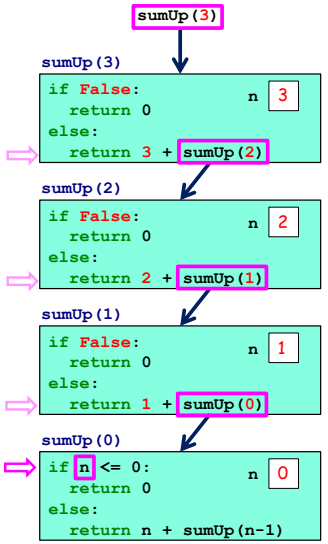
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



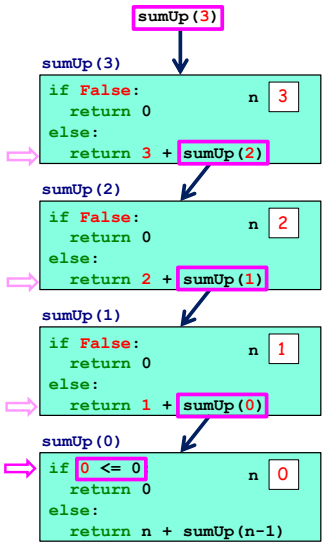
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



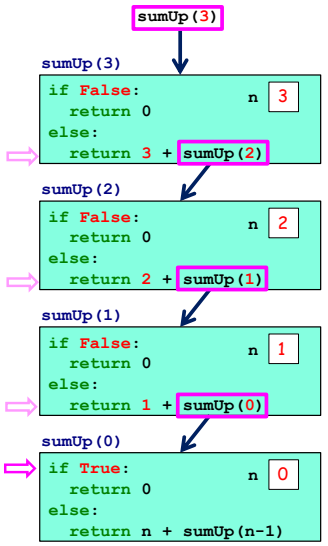
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



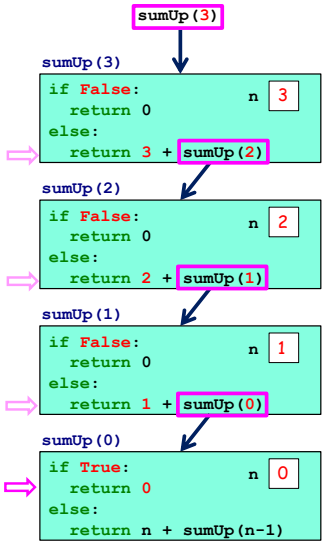
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



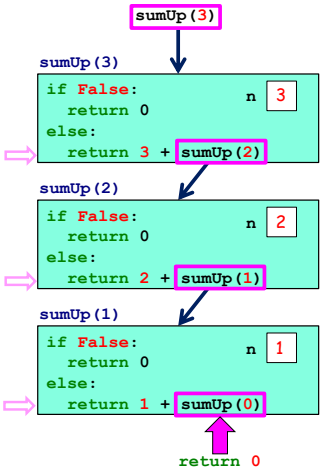
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



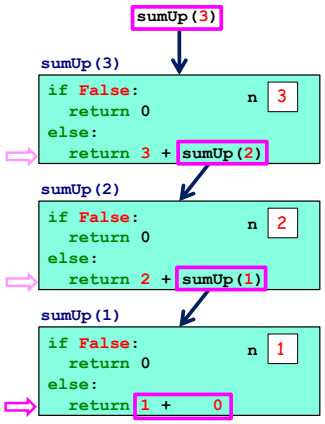
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



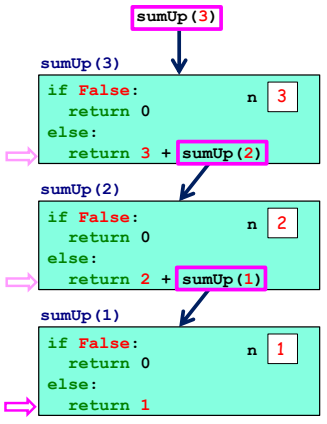
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



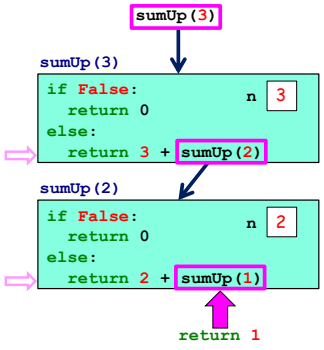
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



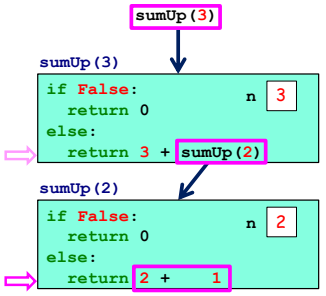
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



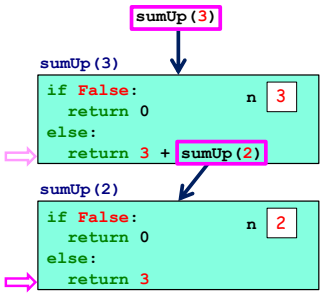
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



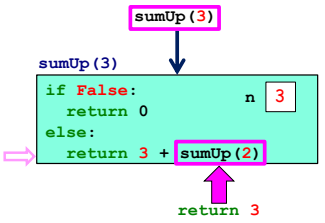
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



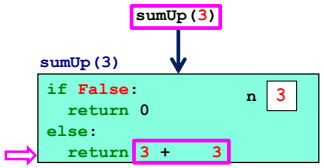
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



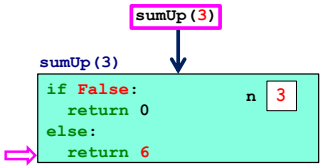
Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

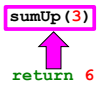


Call frame model for sumUp(3)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```



Call frame model for sumUp(3)




```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```


Call frame model for sumUp(3)

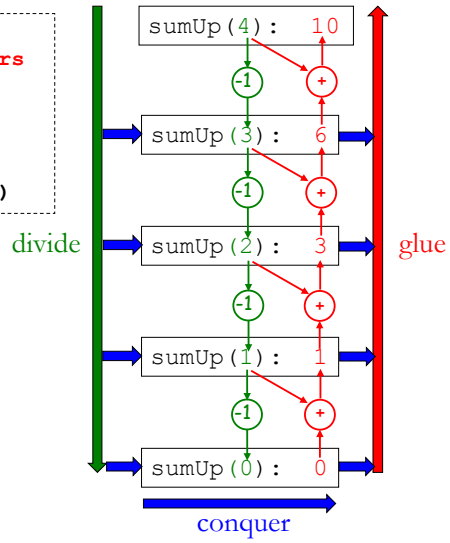
6

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```


Another view: sumUp(4)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

pending addition
is nontrivial glue step



Yet Another view: sumUp(4)

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

```
sumUp(4)
⇒ 4 + sumUp(3)
⇒ 4 + (3 + sumUp(2))
⇒ 4 + (3 + sumUp(2))
⇒ 4 + (3 + (2 + sumUp(1)))
⇒ 4 + (3 + (2 + (1 + sumUp(0))))
⇒ 4 + (3 + (2 + (1 + 0)))
⇒ 4 + (3 + (2 + 1))
⇒ 4 + (3 + 3)
⇒ 4 + 6
⇒ 10
```

In Fruitful Recursion, Base Case(s) are Required

countUp and **sumUp** have similar structure:

```
def countUp(n):
    if n <= 0:
        pass
    else:
        countUp(n-1)
        print(n)
```

```
def sumUp(n):
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

```
def countUp(n):
    if n > 0:
        countUp(n-1)
        print(n)
```

For nonfruitful recursive functions like **countUp**, it's possible to eliminate the **pass** base case by rewriting the conditional, because **else: pass** does nothing.

But for fruitful recursive functions like **sumUp**, no conditional branch can be eliminated, because a return value must be specified for the base case. Often it's an **identity value** for the glue.

```
def sumUp(n):
    if n > 0:
        return n + sumUp(n-1)
    else:
        return 0
```


Practice: Factorial

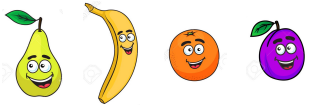
How many ways can you arrange 3 items in a sequence?

Factorial

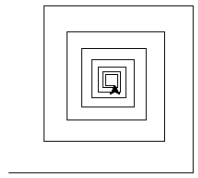
3 items were arranged in 6 different ways. Or 3x2x1. 4 items are arranged 4x3x2x1 ways.

What is the general formula for calculating the arrangements of n items (or n!)?

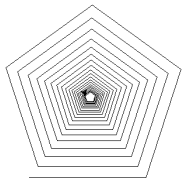
How about 4 items?



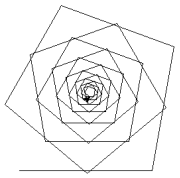
Review: Spiraling Turtles



spiral(200,90,0.9,10)



spiral(200,72,0.97,10)

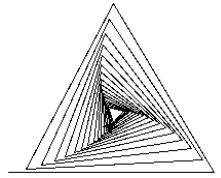


spiral(200,80,0.95,10)

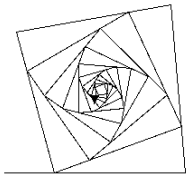
Answer this:

How would you create these shapes using loops?

Recursion makes easier solving certain problems that involve a repeating pattern.



spiral(200,121,0.95,15)



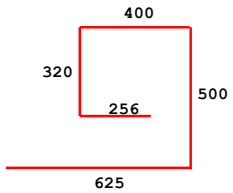
spiral(200,95,0.93,10)

Review: Spiraling Turtles

```
def spiral(sideLen, angle,
           scaleFactor, minLength):
    """Draw a spiral recursively."""

    if sideLen >= minLength:
        fd(sideLen)
        lt(angle)
        spiral(sideLen*scaleFactor,
               angle,
               scaleFactor,
               minLength)
```

- **sideLen** is the length of the current side
- **angle** is the amount the turtle turns left to draw the next side
- **scaleFactor** is the multiplicative factor (between 0.0 and 1.0) by which to scale the next side
- **minLength** is the smallest side length that the turtle will draw



spiral(625, 90, 0.8, 250)

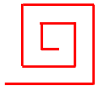
New: How to do Fruitful Spiraling?

Recall the definition for having a turtle draw a spiral and return to its original position and orientation:

```
def spiralBack(sideLen, angle, scaleFactor, minLength):
    """Draws a spiral based on the given parameters and
       brings the turtle back to its initial location and
       orientation."""
    if sideLen < minLength:
        pass
    else:
        fd(sideLen); lt(angle) # Put 2 stmts on 1 line with ;
        spiralBack(sideLen*scaleFactor, angle,
                   scaleFactor, minLength)
        rt(angle); bk(sideLen)
```

How can we modify this function to return
(1) the total length of lines in the spiral;
(2) the number of lines in the spiral;
(3) both of the above numbers in a pair?

spiralLength



```
def spiralLength(sideLen, angle, scaleFactor, minLength)
    """Draws a spiral and returns the total length
       of the lines drawn."""
    if sideLen < minLength:
        return 0
    else:
        fd(sideLen); lt(angle)
        subLen = spiralLength(sideLen*scaleFactor, angle,
                              scaleFactor, minLength)
        rt(angle); bk(sideLen)
        return sideLen + subLen
```

```
spiralLength(100, 90, 0.5, 5) ➔ 193.7

spiralLength(120, 60, 0.5, 5) ➔ 578.8893767467009

spiralLength(512, 90, 0.5, 5) ➔ 1016
```


Exercise: spiralCount

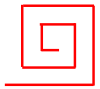

```
def spiralCount(sideLen, angle, scaleFactor, minLength)
    """Draws a spiral and returns the total number
       of lines drawn. """
    if sideLen < minLength:
        return ?? # What goes here?
    else:
        fd(sideLen); lt(angle)
        subCount = spiralCount(sideLen*scaleFactor, angle,
                               scaleFactor, minLength)
        rt(angle); bk(sideLen)
        return ?? # What goes here?
```

```
spiralCount(100, 90, 0.5, 5) ➔ 5

spiralTuple(120, 60, 0.5, 5) ➔ 15

spiralTuple(512, 90, 0.5, 5) ➔ 7
```


Exercise: spiralTuple




```
def spiralTuple(sideLen, angle, scaleFactor, minLength)
    """Draws a spiral and returns a pair of (1) the total length
       of the lines drawn and (2) the number of lines."""
    if sideLen < minLength:
        return ?? # What goes here?
    else:
        fd(sideLen); lt(angle)
        ??                    = spiralTuple(sideLen*scaleFactor, angle,
                                            scaleFactor, minLength)
        rt(angle); bk(sideLen)
        return ?? # What goes here?
```

```
spiralTuple(100, 90, 0.5, 5) ➔ (193.75, 5)

spiralTuple(120, 60, 0.5, 5) ➔ (578.8893767467009, 15)

spiralTuple(512, 90, 0.5, 5) ➔ (1016, 7)
```

Fruitful Trees

As with spirals, we can return counts of the drawings we make using fruitful recursion. Try this example below in the notebook and check the notebook solution for answers.

```
def branchCount(levels, trunkLen, angle, shrinkFactor):
    """Draw a 2-branch tree recursively and returns a
    count of the branches.
    levels: number of branches on any path
            from the root to a leaf
    trunkLen: length of the base trunk of the tree
    angle: angle from the trunk for each subtree
    shrinkFactor: shrinking factor for each subtree
    """
    # your code here
```


List of numbers from n down to 1

Define a function **countDownList** to return the list of numbers from n down to 1

```
countDownList(0) → [ ]
countDownList(5) → [5, 4, 3, 2, 1]
countDownList(8) → [8, 7, 6, 5, 4, 3, 2, 1]
```

Apply the wishful thinking strategy on n = 4:

- `countDownList(4)` should return `[4, 3, 2, 1]`
- By wishful thinking, assume `countDownList(3)` returns `[3, 2, 1]`
- How to combine `4` and `[3, 2, 1]` to yield `[4, 3, 2, 1]`?
  `[4] + [3, 2, 1]`
- Generalize: `countDownList(n)` = `[n] + countDownList(n-1)`

countDownList(n)

```
def countDownList(n):
    """Returns a list of numbers from n down to 1.
       For example, countDownList(5) returns
       [5,4,3,2,1].
    """
    if n <= 0:
        return []
    else:
        return [n] + countDownList(n-1)
```

To remember
When the glue operation in a recursive function involves lists, **the identity value** is the empty list.

Define countDownListPrintResults(n)

```
def countDownListPrintResults(n):
    """Returns a list of numbers from n down to 1
       and also prints each recursive result along
       the way."""
    if n <= 0:
        # add a print statement here
        result = []
    else:
        result = [n] + countDownListPrintResults(n-1)
        # add a print statement here
        return result
```

Exercise: Define `countUpList(n)`

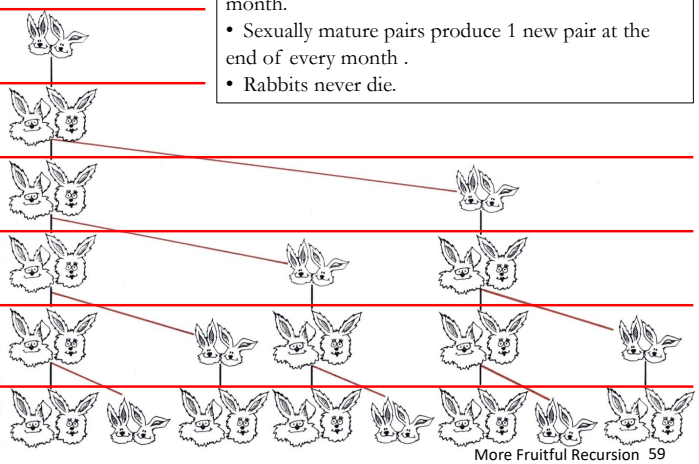

```
def countUpList(n):
    """Returns a list of numbers from 1 up to n.
       For example, countUpList(5) returns
       [1,2,3,4,5]."""
    if n <= 0:
        return ?? # What goes here?
    else:
        return ?? # What goes here?
```


Extra: Fibonacci numbers

Leonardo Pisano Fibonacci counts Rabbits

Month	# Pairs
0	0
1	1
2	1
3	2
4	3
5	5
6	8

Assume:
- Start with one pair of newborn rabbits in month 1.
- Newborn rabbits become sexually mature after 1 month.
- Sexually mature pairs produce 1 new pair at the end of every month .
- Rabbits never die.



Exercise: Fibonacci Numbers `fib(n)`

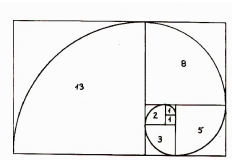
The **n**th Fibonacci number fib(n) is the number of pairs of rabbits alive in the **n**th month.

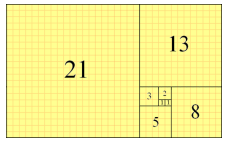
Formula:

fib(0) = 0 ; no pairs initially
fib(1) = 1 ; 1 pair introduced the first month
fib(n) = fib(n-1) ; pairs never die, so live to next month
+ fib(n-2) ; all sexually mature pairs produce
; a pair each month

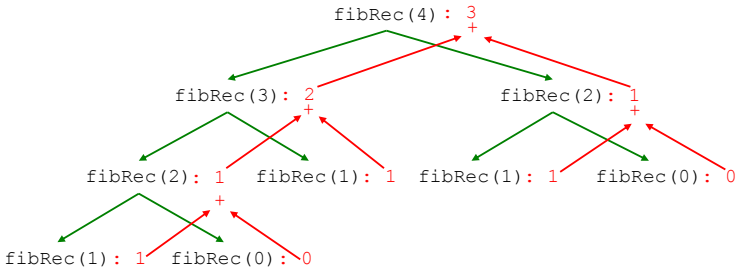
Now write the program:

```
def fibRec(n):
    '''Returns the nth Fibonacci number.'''
    if n <= 1:
        return n
    else:
        return fibRec(n-1) + fibRec(n-2)
```





Fibonacci: Efficiency



How long would it take to calculate `fibRec(100)`?

Is there a better way to calculate Fibonacci numbers?

Iteration leads to a more efficient `fib(n)`

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

Iteration table for calculating the 8th Fibonacci number:

i	fibi	fibi_next
0	0	1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13
7	13	21
8	21	34

Exercise: `fibLoop(n)`

Use iteration to calculate Fibonacci numbers more efficiently:

i	fibi	fibi_next
0	0	1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13
7	13	21
8	21	34

```
def fibLoop(n):
    '''Returns the nth Fibonacci number.'''
    fibi = 0
    fibi_next = 1
    for i in range(1, n+1):
        # flesh out this loop body


    return ?? # What goes here?
```