Fruitful Recursion
(recursion that returns a value)

Sum of numbers from 1 to n
- Recall `countUp(n)` for printing integers from 1 up to n:
  ```python
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?

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(n)`**

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

Compare this to `countUp(n)`:  

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

---

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

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

Call frame model for sumUp(3)

Call frame model for sumUp(3)

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

def sumUp(n):
    """returns sum of integers from 1 up to n""
    if n <= 0:
        return 0
    else:
        return 3 + sumUp(n-1)
def sumUp(n):
    """returns sum of integers from 1 up to n""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)

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

sumUp(3): 6
sumUp(2): 3
sumUp(1): 1
sumUp(0): 0

sumUp(4): 10
sumUp(3): 6
sumUp(2): 3
sumUp(1): 1
sumUp(0): 0

Another view: sumUp(4)

Yet Another view: sumUp(4)
In Fruitful Recursion, Base Case(s) are Required

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

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

For nonfruitful recursive functions like `countUp(n)`, 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(n)`, 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.

Exercise 1: Factorial

\[ n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \]

Write a function that computes the factorial of `n` (using the same logic as `sumUp`).

General case by wishful thinking: \[ n! = n \cdot (n-1)! \]

```python
def factorial(n):
    """Returns n! using recursion""
    if n <= 0:
        return 1 # use penultimate strategy for this
    else:
        return n * factorial(n-1)
```

Factorial

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

How about 4 items?

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)`
**Exercise 2: Define `countDownList(n)`**

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

**Exercise 3: Define `countDownListPrintResults(n)`**

```python
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:
        result = []
    else:
        result = [n] + countDownListPrintResults(n-1)
        print result
    return result
```

**Exercise 4: Define `countUpList(n)`**

```python
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 []
    else:
        return countDownList(n-1) + [n]
```

**Fruitful Spiraling**

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

```python
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?
Exercise 5: `spiralLength`

```python
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 6: `spiralCount`

```python
def spiralCount(sideLen, angle, scaleFactor, minLength):
    """Draws a spiral and returns the total number of lines drawn."""
    if sideLen < minLength:
        return 0
    else:
        fd(sideLen); lt(angle)
        subCount = spiralCount(sideLen*scaleFactor, angle, scaleFactor, minLength)
        rt(angle); bk(sideLen)
        return 1 + subCount
```

- `spiralCount(100, 90, 0.5, 5) ➞ 5`
- `spiralCount(120, 60, 0.5, 5) ➞ 15`
- `spiralCount(512, 90, 0.5, 5) ➞ 7`

Exercise 7: `spiralTuple`

```python
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 (0, 0)
    else:
        fd(sideLen); lt(angle)
        (subLength, subCount) = spiralCount(sideLen*scaleFactor, angle, scaleFactor, minLength)
        rt(angle); bk(sideLen)
        return (sideLen + subLength, 1 + subCount)
```

- `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)`
Exercise 8: **upperRightRepeat**

```python
def upperRightRepeat(levels, pic):
    """Returns a picture containing the specified picture
    nested in the upper right corner of itself, the
    specified number of levels.""
    if levels <= 0:
        return empty()
    else:
        return upperRightNest(pic, upperRightRepeat(levels-1, pic))
```

---

upperRightNest in PictureWorld

We have defined a function, `upperRightNest`, in `picture.py` that takes two pictures (200 x 200 graphics objects) and returns a picture with the second one overlaid in the upper right quadrant of the first.

```python
def upperRightNest(pic1, pic2):
    """Returns a new picture in which pic2 is overlaid on
    the upper right quadrant of pic1""
    return overlay(fourPics(empty(), pic2, empty(), empty()), pic1)
```

---

Leonardo Pisano Fibonacci counts Rabbits

<table>
<thead>
<tr>
<th>Month</th>
<th># Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

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 9: **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:**

```python
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 \( \text{fibRec}(100) \)?

Is there a better way to calculate Fibonacci numbers?

Iteration leads to a more efficient \( \text{fib}(n) \)

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

Iteration table for calculating the 8th Fibonacci number:

<table>
<thead>
<tr>
<th>i</th>
<th>fibi</th>
<th>fibi_next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>34</td>
</tr>
</tbody>
</table>

Exercise 10: \text{fibLoop}(n)

Use iteration to calculate Fibonacci numbers more efficiently:

```python
def fibLoop(n):
    '''Returns the nth Fibonacci number.'''
    fibi = 0
    fibi_next = 1
    for i in range(1, n+1):
        fibi, fibi_next = fibi_next, fibi+fibi_next
        # tuple assignment simultaneously updates state vars
    return fibi
```

"it's your turn"