Fruitful Recursion
(recursion that returns a value)

Recall: fruitful functions and side effects

Recall: fruitful functions and side effects

Recursive functions today:
- sumUp
- factorial
- countDown
- print
- len
- int

Recursive functions in last two lectures:
- countDown
- spiral
- countUp
- spiralBack
- countDownUp
- tree
- drawTarget
- upperRight
- repeat
- drawNestedCircles
- spiralLength
- spiralCount
- spiralTuple

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?

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, \( \text{sumUp}(4) \) 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, \( \text{sumUp}(3) \) 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, \( \text{sumUp}(4) \) should return \( 4 + \text{sumUp}(3) \)

4. Generalize the concrete example into the general case:
   
   In this case, \( \text{sumUp}(n) \) should return \( n + \text{sumUp}(n-1) \)
What about the base case?
Use the recursive case for the penultimate input

For example, what should \( \text{sumUp}(0) \) return?

1. According to the recursive case:
   \( \text{sumUp}(n) \) should return \( n + \text{sumUp}(n-1) \)

2. Specialize the recursive case to the penultimate (next to last) input:
   \( \text{sumUp}(1) \) should return \( 1 + \text{sumUp}(0) \)

3. Decide what should be returned for the penultimate input.
   In this case, \( \text{sumUp}(1) \) should clearly return 1.

4. Deduce what should be returned for the base case.
   \( \text{sumUp}(1) \) equals 1 equals \( 1 + \text{sumUp}(0) \),
   so \( \text{sumUp}(0) \) 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 \( \text{countUp}(n) \):

```
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)**

```
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)
```
Another view: \( \text{sumUp}(4) \)

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

Yet Another view: \( \text{sumUp}(4) \)

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

In Fruitful Recursion, Base Case(s) are Required

\( \text{countUp} \) and \( \text{sumUp} \) have similar structure:

```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 \( \text{countUp} \), it's possible to eliminate the \texttt{pass} base case by rewriting the conditional, because \texttt{else: pass} does nothing.

But for fruitful recursive functions like \( \text{sumUp} \), 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.

Factorial

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

3 items were arranged in 6 different ways. Or 3x2x1. What is the general formula for calculating the arrangements of \( n \) items (or \( n! \))?

We will write the code to calculate the factorial with code in the Notebook.
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)`

Fruitful Recursion 20-13

Fruitful Spiraling

Recall the definition for having a turtle draw a spiral and return to its initial location 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?

Fruitful Recursion 20-14

Exercises for the notebook

```python
def spiralLength(sideLen, angle, scaleFactor, minLength):
    """Draws a spiral and returns the total length of the lines drawn."""
    spiralLength(100, 90, 0.5, 5) → 193.75

def spiralCount(sideLen, angle, scaleFactor, minLength):
    """Draws a spiral and returns the total number of lines drawn."""
    spiralCount(100, 90, 0.5, 5) → 5

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."""
    spiralTuple(100, 90, 0.5, 5) → (193.75, 5)
```

Fruitful Recursion 20-15

upperRightRepeat

```python
def upperRightRepeat(5, redLeaves):

def upperRightRepeat(5, triangles):

def upperRightRepeat(5, angryBird):
```

Fruitful Recursion 20-16
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: Fibonacci Numbers `fib`

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
- $fib(n) = fib(n-1)$; all sexually mature pairs produce a pair each month
- $fib(n) = fib(n-2)$; a pair each month

Now write the program:

```python
def fibRec(n):
    """Returns the nth Fibonacci number""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    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?
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>

Iteration leads to more efficient `fib`

Exercise: `fibLoop(n)`

Use iteration to calculate Fibonacci numbers more efficiently:

```python
def fibLoop(n):
    '''Returns the nth 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>