RECURSION

LECTURE 06-2

JIM FIX, REED COLLEGE CSCI 121

AN INTERESTING PROCEDURE

- Consider this procedure:

```
def outputCount(count)
    print(count)
    outputCount(count - 1)
```

- What does this do?

```
>>> outputCount(10)
????
```

AN INTERESTING PROCEDURE

- Consider this procedure:

```
def outputCount(count)
    print(count)
    outputCount(count - 1)
```

- It counts down from 10

```
>>> outputCount(10)
10
9
8
...
```

- But then it keeps going!

AN INTERESTING PROCEDURE

- Consider this procedure:

```
def outputCount(count)
    print(count)
    outputCount(count - 1)
```

- It counts down from 10

```
>>> outputCount(10)
10
9
8
...
```

- But then it keeps going!
- The calls stack up, deeper and deeper, until Python's "maximum recursion depth" gets reached and Python bails with an error.

AN INTERESTING PROCEDURE

- Consider this procedure:

```
def outputCount(count)
    print(count)
    outputCount(count - 1)
```

- It counts down from 10

```
>>> outputCount(10)
10
9
8
...
```

- But then it keeps going!
- Can we re-write it so it only counts down to 1?

COUNTING DOWN TO 1

- Yes. Here is the *rewrite*:

```
def outputCount(count)
    print(count)
    if count > 1:
        outputCount(count - 1)
```

- It counts down from 10 and stops.

```
>>> outputCount(10)
10
9
8
7
6
5
4
3
2
1
>>>
```

COUNTING DOWN TO 1

- This code counts from **count** down to **1**.

```
def outputCount(count)
    print(count)
    if count > 1:
        outputCount(count - 1)
```

- Its procedure to count from 10, down, relies on the procedure to count from 9.

```
>>> outputCount(10)
10
9
8
7
6
5
4
3
2
1
>>>
```

These are just the lines of **outputCount(9)**.

THE SAME, WITH A SMALL TWEAK

- How about this procedure, *with just one change*:

```
def outputCountTweaked(count):
    print(count)
    if count > 1:
        outputCountTweaked(count - 1)
    print(count)
```

- What does it do?

```
>>> outputCountTweaked(5)
????
```

THE SAME, WITH A SMALL TWEAK

- How about this procedure, *with just one change*:

```
def outputCountTweaked(count):
    print(count)
    if count > 1:
        outputCountTweaked(count - 1)
    print(count)
```

- It prints the numbers from 5 down to 1, then counts back up again:

```
>>> outputCountTweaked(5)
5
4
3
2
1
1
2
3
4
5
>>>
```

THE SAME, WITH A SMALL TWEAK

- How about this procedure, *with just one change*:

```
def outputCountTweaked(count)
    print(count)
    if count > 1:
        outputCountTweaked(count - 1)
    print(count)
```

- It prints the numbers from 5 down to 1, then counts back up again:

```
>>> outputCountTweaked(5)
5
4
3
2
1
1
2
3
4
5
>>>
```

This is just the lines of **`outputCountTweaked(4)`**.

THE SAME, WITH A SMALL TWEAK

- How about this procedure, *with just one change*:

```
def outputCountTweaked(count)
    print(count)
    if count > 1:
        outputCountTweaked(count - 1)
    print(count)
```

- Why? Recall that function and procedure calls "stack up"...

```
>>> outputCountTweaked(5)
5
4
3
2
1
1
2
3
4
5
>>>
```

This is just the lines of `outputCountTweaked(4)`.

- ... and so the call with `count` of 5 waits for the call for 4 to finish, then prints it's second 5.

RECURSION

- A procedure or function that calls itself is ***recursive***.
  - Some clever algorithms are naturally expressed this way.
  - The general programming technique is ***recursion***.
- **Reading** on recursion:
  - TP 4.9-4.11, 5.5
  - CP 1.7

RECURSIVE PROCEDURES

- Recursive procedures are very common in computer science. They are sometimes a natural way of expression an algorithm.

- Here is a procedure for sorting a collection of items:

"Pick an item from that collection. Divide the rest into a collection of things that come before that item, and a collection of things that come after it. ***Follow this same procedure*** *to sort each of those collections into an order, and put the chosen item between those two orderings*."

RECURSIVE PROCEDURES

- Recursive procedures are very common in computer science. They are sometimes a natural way of expression an algorithm.

- Here is a procedure for sorting a collection of items:

"Pick an item from that collection. Partition the other items into a collection of things that come before that item, and a collection of things that come after it. ***Follow this same procedure*** *to sort each of those collections into an order, and put the chosen item between those two orderings*."

- Here is a procedure for sorting a collection of items:

"Split the collection into two collections, arbitrarily. ***Follow this same procedure*** *to sort each of those collections into an order*. Merge those two orderings."

TWO FAMOUS SORTING ALGORITHMS IN PYTHON

- Both of these sorting algorithms are famous. The first is called "quick sort."
- The **quick sort** procedure for sorting a collection of items:

"Pick an item from that collection. Partition the other items into a collection of things that come before that item, and a collection of things that come after it. ***Follow this same* quick sort *procedure*** *to sort each of those collections into an order, and put the extra item between those two orderings*."

- Here is a Python procedure that mimics the above:

```
def quick_sort(items):
    item = choose_item(items)
    befores, afters = partition(items,item)
    sorted_befores = quick_sort(befores)
    sorted_afters = quick_sort(afters)
    return sorted_befores + [item] + sorted_afters
```

- It is not quite correct! It doesn't properly recognize the "bottom case".

TWO FAMOUS SORTING ALGORITHMS IN PYTHON

- Both of these sorting algorithms are famous. The first is called "quick sort."
- The **quick sort** procedure for sorting a collection of items:

"Pick an item from that collection. Partition the other items into a collection of things that come before that item, and a collection of things that come after it. ***Follow this same* quick sort *procedure*** *to sort each of those collections into an order, and put the extra item between those two orderings*."

- Here is the correct Python procedure that mimics the above:

```
def quick_sort(items):
    if len(items) == 0: # nothing to sort
        return []
    item = choose_item(items)
    befores, afters = partition(items,item)
    sorted_befores = quick_sort(befores)
    sorted_afters = quick_sort(afters)
    return sorted_befores + [item] + sorted_afters
```

A RECURSIVE PROCEDURE

- Handling an empty list is called a ***base case*** of this recursive algorithm.
  - A base case is typically an "easy enough to handle" case.
- The other kind of case is called a ***recursive case***.
  - It is any case that is handled by procedure calling itself.
  - When it calls itself, it typically hands itself an "easier" case to handle.

BASE CASE VERSUS RECURSIVE CASE

- Note that we could have used 0 as the base case for our count down code:

```
def countDownFrom(start)
    if start == 0:
        return
    else:
        print(start)
        countDownFrom(start - 1)
```

BASE CASE VERSUS RECURSIVE CASE

- The ***base case*** is when someone *counts down from 0*:

```
def countDownFrom(start)
    if start == 0:
        return # Do nothing when start is 0
    else:
        print(start)
        countDownFrom(start - 1)
```

BASE CASE VERSUS RECURSIVE CASE

- Our ***recursive case*** is when someone counts down from a positive number

```
def countDownFrom(start)
    if start == 0:
        return
    else:
        print(start)              # Print the number then...
        countDownFrom(start - 1) # count from one below it.
```

THE SECOND RECURSIVE SORT: MERGE SORT

- Both of these sorting algorithms are famous. The second is called "merge sort."

- Here is the **merge sort** procedure for sorting a collection of items:

"Split the collection into two collections. ***Follow this same*** **merge sort** ***procedure*** *to sort each of those collections into an order*. Merge these two orderings"

- Here is a Python procedure that mimics the above:

```
def merge_sort(items):
    if len(items) == 0: # nothing to sort
        return []
    part1, part2 = split(items)
    sorted1 = merge_sort(part1)
    sorted1 = merge_sort(part2)
    return merge(sorted1, sorted2)
```

THE SECOND RECURSIVE SORT: MERGE SORT

- Both of these sorting algorithms are famous. The second is called "merge sort."

- Here is the **merge sort** procedure for sorting a collection of items:

"Split the collection into two collections. ***Follow this same* merge sort *procedure to sort each of those collections into an order***. Merge these two orderings"

- Here is a Python procedure that mimics the above:

```
def merge_sort(items):
    if len(items) == 0: # nothing to sort
        return []
    part1, part2 = split(items)
    sorted1 = merge_sort(part1)
    sorted1 = merge_sort(part2)
    return merge(sorted1, sorted2)
```

- We will look at these sorts more carefully in the second half of the course.

RECURSIVE FUNCTIONS

- Let's invent a recursive function.
- Suppose we wanted to write Python code that computes this sum:

```
1 + 2 + 3 + ... + 99 + 100 == ????
```

- And we want it to work for any value of `n`, not just up to `100`.

RECURSIVE FUNCTIONS

- Let's invent a recursive function.
- Suppose we wanted to write Python code that computes this sum:

```
1 + 2 + 3 + ... + (n-1) + n == ????
```

- And we want it to work for any value of `n`, not just up to `100`.

```
def sumUpTo(n):
    ????
```

RECURSIVE FUNCTIONS

- Let's invent a recursive function.
- Suppose we wanted to write Python code that computed this sum:

```
(1 + 2 + 3 + ... + (n-1)) + n == ????
```

- We see that the sum up to **n** relies on computing the sum up to **n-1**
- So we try this:

```
def sumUpTo(n):
    return sumUpTo(n-1) + n
```

- But this turns out to have the same problem as our first count code.
  - There's no base case to stop the "unwinding" of the sum.

RECURSIVE FUNCTIONS

- Let's invent a recursive function.
- Suppose we wanted to write Python code that computed this sum:

```
(1 + 2 + 3 + ... + (n-1)) + n == ????
```

- Here is working code that has 1 as the base case

```
def sumUpTo(n):
    if n == 1:
        return 1
    else:
        return sumUpTo(n-1) + n
```

RECURSIVE FUNCTIONS

- Let's invent a recursive function.
- Suppose we wanted to write Python code that computed this sum:

```
(1 + 2 + 3 + ... + (n-1)) + n == ????
```

- This one considers non-positive sums as "trivially 0":

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

RECURSION AS SUBSTITUTION

- Defined recursively how are we to think of an expression like what's below?

```
>>> sumUpTo(5)
```

RECURSION AS SUBSTITUTION

- Defined recursively how are we to think of an expression like what's below?

```
>>> sumUpTo(5)
```

- I imagine some series of rewriting steps, or substitutions, so this is like:

```
>>> sumUpTo(4) + 5
```

RECURSION AS SUBSTITUTION

- Defined recursively how are we to think of an expression like what's below?

```
>>> sumUpTo(5)
```

- I imagine some series of rewriting steps, or substitutions, so this is like:

```
>>> sumUpTo(4) + 5
```

- which is like

```
>>> (sumUpTo(3) + 4) + 5
```

RECURSION AS SUBSTITUTION

- Defined recursively how are we to think of an expression like what's below?

```
>>> sumUpTo(5)
```

- I consider some series of rewriting steps, or substitutions, so this is like:

```
>>> sumUpTo(4) + 5
```

- which is like

```
>>> (sumUpTo(3) + 4) + 5
```

- which is like

```
>>> ((sumUpTo(2) + 3) + 4) + 5
```

- And so on...

RECURSION AS SUBSTITUTION

- Defined recursively how are we to think of an expression like what's below?

```
>>> sumUpTo(5)
```

- I consider some series of rewriting steps, or substitutions, so this is like:

```
>>> sumUpTo(4) + 5
```

- which is like

```
>>> (sumUpTo(3) + 4) + 5
```

- which is like

```
>>> ((sumUpTo(2) + 3) + 4) + 5
```

- And so on. So `sumUpTo(5)` is this sum:

```
>>> (((((0) + 1) + 2) + 3) + 4) + 5
```

- It is the recursion, unwound down to the base case. And so:

```
>>> sumUpTo(5)
15
```

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

number = int(input("Number? "))
print(sumUpTo(number))
```

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

number = int(input("Number? "))
print(sumUpTo(number))
```

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

number = int(input("Number? "))
print(sumUpTo(number))
```

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(3) frame

n: 3

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(3) frame

n: 3

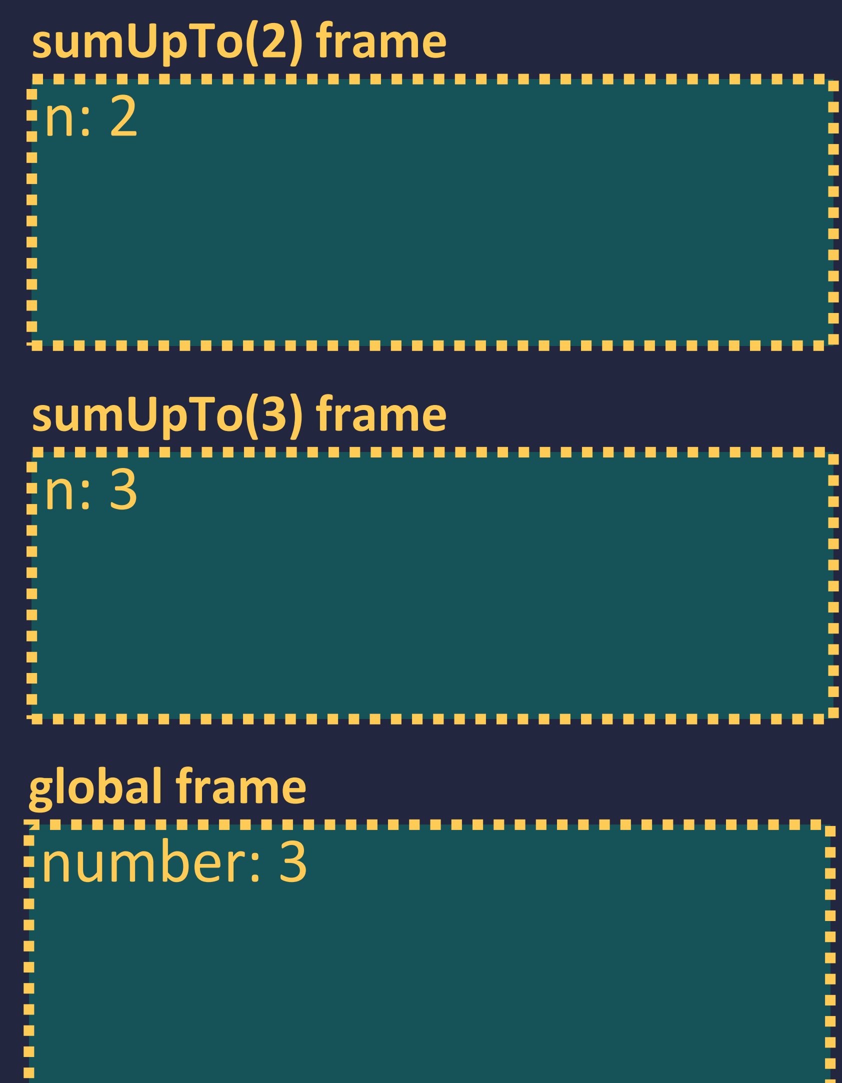
global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

number = int(input("Number? "))
print(sumUpTo(number))
```



PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(2) frame

n: 2

sumUpTo(3) frame

n: 3

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(1) frame

n: 1

sumUpTo(2) frame

n: 2

sumUpTo(3) frame

n: 3

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(1) frame

n: 1

sumUpTo(2) frame

n: 2

sumUpTo(3) frame

n: 3

global frame

number: 3

PYTHON'S EXECUTION OF A REC ON

- Let's take a look at Python's execution of this script:

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

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(0) frame

n: 0

sumUpTo(1) frame

n: 1

sumUpTo(2) frame

n: 2

sumUpTo(3) frame

n: 3

global frame

number: 3

PYTHON'S EXECUTION OF A REC ON

- Let's take a look at Python's execution of this script:

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

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(0) frame

n: 0
returning 0

sumUpTo(1) frame

n: 1

sumUpTo(2) frame

n: 2

sumUpTo(3) frame

n: 3

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(0) frame

n: 0
returning 0

sumUpTo(1) frame

n: 1
returning 1

sumUpTo(2) frame

n: 2

sumUpTo(3) frame

n: 3

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

def sumUpTo(n):
    if n <= 0:
        return 0
    return sumUpTo(n-1) + n

number = int(input("Number? "))
print(sumUpTo(number))
```

sumUpTo(1) frame

n: 1
returning 1

sumUpTo(2) frame

n: 2
returning 3

sumUpTo(3) frame

n: 3

global frame

number: 3

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

sumUpTo(2) frame

n: 2
returning 3

sumUpTo(3) frame

n: 3
returning 6

global frame

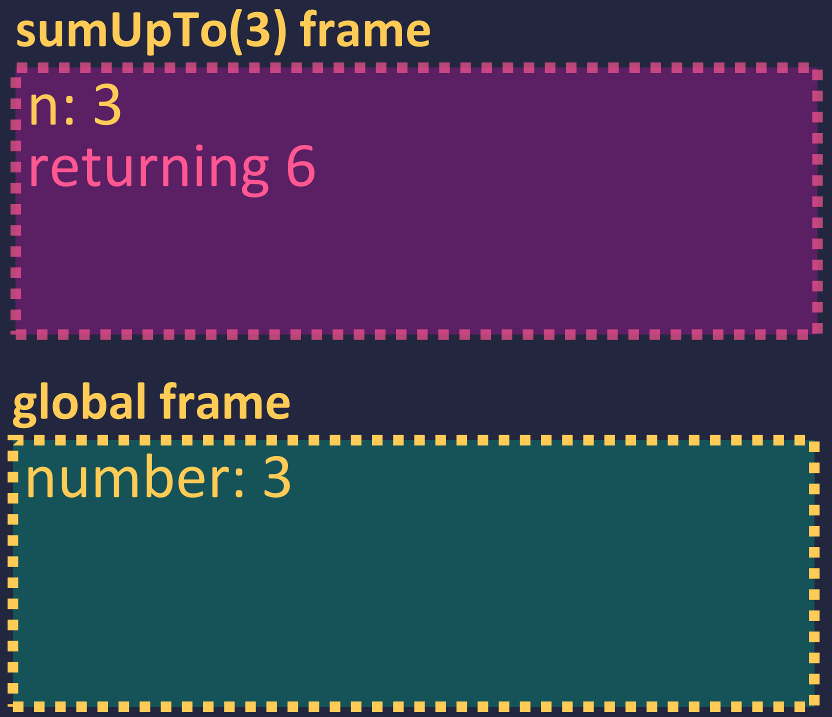
number: 3

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

number = int(input("Number? "))
print(sumUpTo(number))
```

PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

number = int(input("Number? "))
print(sumUpTo(number))
```



PYTHON'S EXECUTION OF A RECURSIVE FUNCTION

- Let's take a look at Python's execution of this script:

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

number = int(input("Number? "))
print(sumUpTo(number))
```

Outputs 6 to the console.

global frame

number: 3

THE FIBONACCI FUNCTION

- Consider the following integer sequence:
  - It starts with a 1.
  - The second number is also a 1.

1, 1,

THE FIBONACCI FUNCTION

- Consider the following integer sequence:
  - It starts with a 1.
  - The second number is also a 1.
  - The next number is the sum of the previous two.

1, 1, 2,

THE FIBONACCI FUNCTION

- Consider the following integer sequence:
  - It starts with a 1.
  - The second number is also a 1.
  - The next number is the sum of the previous two.
  - And so are the rest of the numbers.

1, 1, 2, 3,

THE FIBONACCI FUNCTION

- Consider the following integer sequence:
  - It starts with a 1.
  - The second number is also a 1.
  - The next number is the sum of the previous two.
  - And so are the rest of the numbers.

1, 1, 2, 3, 5,

THE FIBONACCI FUNCTION

- Consider the following integer sequence:
  - It starts with a 1.
  - The second number is also a 1.
  - The next number is the sum of the previous two.
  - And so are the rest of the numbers.

1, 1, 2, 3, 5, 8,

THE FIBONACCI FUNCTION

- Consider the following integer sequence:
  - It starts with a 1.
  - The second number is also a 1.
  - The next number is the sum of the previous two.
  - And so are the rest of the numbers.

    1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- This is the Fibonacci sequence, and it has lots of interesting properties.
- Let's just write its code.

THE FIBONACCI FUNCTION

- Consider the following integer sequence:
  - It starts with a 1.
  - The second number is also a 1.
  - The next number is the sum of the previous two.
  - And so are the rest of the numbers.

  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- This is the Fibonacci sequence, and it has lots of interesting properties.
- Let's just write its code as a Python function:

```
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)
```

SUMMARY

- Functions and procedures can call other functions and procedures.
  - They can also *call themselves*. This makes them **recursive**.
- Each active function has its local variables stored in its **call frame**.
  - With recursion, several call frames for the same-named function ***stack*** up.
  - Each call has a different value for the parameter in each frame.
- Recursive functions are designed to handle two cases:
  - a **recursive case**: this leads the function to call itself
    - usually a (slightly) simpler case
  - a **base case**: this stops the "unwinding" or "deepening" of the recursive calls
    - they are (usually) easy cases; immediately return a result
- The tricky part is learning to express algorithms in this way.