

RECURSION

LECTURE 05-2

JIM FIX, REED COLLEGE CSC1 121

AN INTERESTING FUNCTION

- ▶ Consider this function:

```
def list_using(number)
  if number == 1
    return [1]
  else:
    return [number] + list_using(number-1) + [number]
```

- ▶ What does this compute?

```
>>> list_using(5)
????
```

AN INTERESTING FUNCTION

- ▶ Consider this function:

```
def list_using(number)
    if number == 1
        return [1]
    else:
        return [number] + list_using(number-1) + [number]
```

- ▶ What does this compute?

```
>>> list_using(5)
[5, 4, 3, 2, 1, 2, 3, 4, 5]
```

AN INTERESTING FUNCTION

- ▶ Consider this function:

```
def list_using(number)
    if number == 1
        return [1]
    else:
        return [number] + list_using(number-1) + [number]
```

- ▶ What does this compute?

```
>>> list_using(5)
[5, 4, 3, 2, 1, 2, 3, 4, 5]
>>> list_using(4)
[4, 3, 2, 1, 2, 3, 4]
>>> list_using(1)
[1]
```

AN INTERESTING FUNCTION

- ▶ Consider this function:

```
def list_using(number)
  if number == 1
    return [1]
  else:
    return [number] + list_using(number-1) + [number]
```

- ▶ What's interesting about it?

AN INTERESTING FUNCTION

- ▶ Consider this function:

```
def list_using(number)
  if number == 1
    return [1]
  else:
    return [number] + list_using(number-1) + [number]
```

- ▶ What's interesting about it? ***It calls itself in order to compute its result.***
- ▶ Functions that do this are ***recursive functions.***

AN INTERESTING FUNCTION

- ▶ Consider this function:

```
def list_using(number)
    if number == 1
        return [1]
    else:
        return [number] + list_using(number-1) + [number]
```

- ▶ What's interesting about its result?
- ▶ It handles "larger cases" by building from its "answers" for smaller cases:

```
>>> list_using(5)
[5, 4, 3, 2, 1, 2, 3, 4, 5]
>>> list_using(4)
[4, 3, 2, 1, 2, 3, 4]
>>> list_using(1)
[1]
```

AN INTERESTING FUNCTION

- ▶ Consider this function:

```
def list_using(number)
  if number == 1
    return [1]
  else:
    return [number] + list_using(number-1) + [number]
```

- ▶ What's interesting about its result?
- ▶ It handles "larger cases" by building from its "answers" for smaller cases:

```
>>> list_using(5)
[5, 4, 3, 2, 1, 2, 3, 4, 5]
>>> list_using(4)
[4, 3, 2, 1, 2, 3, 4]
>>> list_using(1)
[1]
```



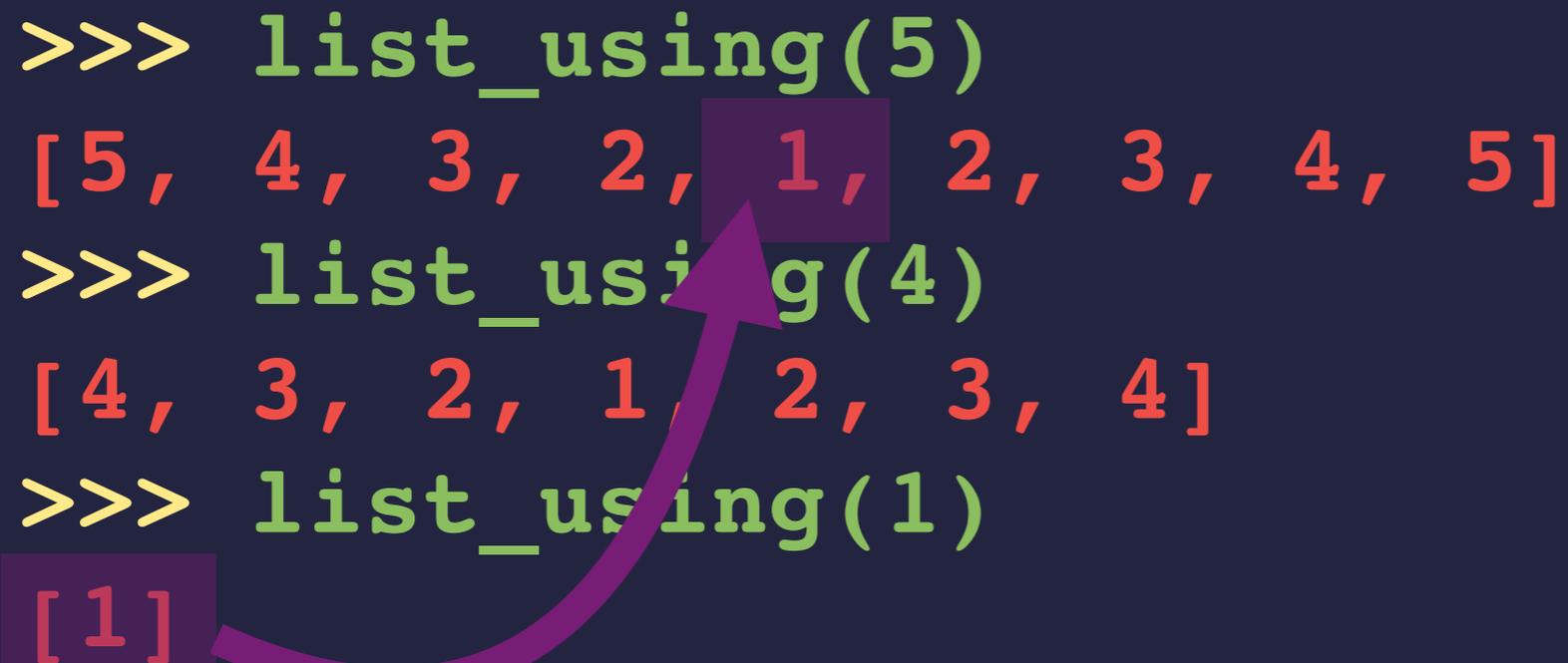
AN INTERESTING FUNCTION

- ▶ Consider this function:

```
def list_using(number)
  if number == 1
    return [1]
  else:
    return [number] + list_using(number-1) + [number]
```

- ▶ What's interesting about its result?
- ▶ It handles "larger cases" by building from its "answers" for smaller cases:

```
>>> list_using(5)
[5, 4, 3, 2, 1, 2, 3, 4, 5]
>>> list_using(4)
[4, 3, 2, 1, 2, 3, 4]
>>> list_using(1)
[1]
```



AN INTERESTING PROCEDURE

- ▶ Consider this procedure:

```
def output_using(number)
  if number == 1
    print(1)
  else:
    print(n)
    output_using(number-1)
    print(n)
```

- ▶ What does this do?

```
>>> output_using(4)
??????
```

AN INTERESTING PROCEDURE

- ▶ Consider this procedure:

```
def output_using(number)
  if number == 1
    print(1)
  else:
    print(n)
    output_using(number-1)
    print(n)
```

- ▶ What does this do?

```
>>> output_using(4)
```

```
4
3
2
1
2
3
4
```

AN INTERESTING PROCEDURE

- ▶ Consider this procedure:

```
def output_using(number)
  if number == 1
    print(1)
  else:
    print(n)
    output_using(number-1)
    print(n)
```

- ▶ It is also ***recursive***.

AN INTERESTING PROCEDURE

- ▶ Consider this procedure:

```
def output_using(number)
  if number == 1
    print(1)
  else:
    print(n)
    output_using(number-1)
    print(n)
```

- ▶ It contains the result of the activity of simpler cases...

```
>>> output_using(4)
```

4

3

2

1

2

3

4



the activity of `output_using(3)`

AN INTERESTING PROCEDURE

- ▶ Consider this procedure:

```
def output_using(number)
  if number == 1
    print(1)
  else:
    print(n)
    output_using(number-1)
    print(n)
```

- ▶ It contains the result of the activity of simpler cases...

```
>>> output_using(4)
```

4

3

2

1

2

3

4

← the activity of `output_using(1)`

ANOTHER RECURSIVE 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!

ANOTHER RECURSIVE 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.

ANOTHER RECURSIVE 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
>>>
```

A RECURSIVE PROCEDURE

- ▶ A function or procedure that "calls itself" is **recursive**.
- ▶ Here, I've rewritten it for (perhaps) a decent explanation

```
def countdownFrom(start)
    if start == 1:
        print(1)
    elif start > 1:
        print(start)
        countdownFrom(start - 1)
```

- ▶ You can think of it this way, describing **the procedure for counting down to 1**:
*"When asked to count down from 1 down to 1, just say "one." When asked to count down from a number larger than 1, just say the starting number. And **then follow this same procedure** for counting down from its predecessor."*

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:
        outputCount(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:
        outputCount(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*:

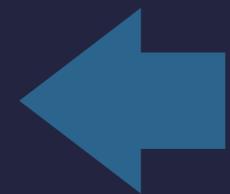
```
def outputCountTweaked(count)
    print(count)
    if count > 1:
        outputCount(count - 1)
    print(count)
```

- ▶ 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 its 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*.

SOME OTHER RECURSIVE PROCEDURES

- ▶ Recursive procedures are very common in computer science. They are sometimes a natural way of expressing 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.*"

SOME OTHER RECURSIVE PROCEDURES

- ▶ Recursive procedures are very common in computer science. They are sometimes a natural way of expressing 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."

QUICK SORT

- ▶ 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)  
    before_after = partition(items, item)  
    before = before_after[0]  
    after = before_after[1]  
    sorted_before = quick_sort(before)  
    sorted_after = quick_sort(after)  
    return sorted_before + [item] + sorted_after
```

QUICK SORT CODE ATTEMPT

- ▶ Here is an attempt at quick-sort code in Python:

```
def quick_sort(items):  
    item = choose_item(items)  
    befores_afters = partition(items, item)  
    befores = befores_afters[0]  
    afters = befores_afters[1]  
    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".

QUICK SORT CODE

- ▶ Here is the correct quick-sort code in Python:

```
def quick_sort(items):  
    if len(items) == 0: # nothing to sort  
        return []  
    item = choose_item(items)  
    befores_afters = partition(items, item)  
    befores = befores_afters[0]  
    afters = befores_afters[1]  
    sorted_befores = quick_sort(befores)  
    sorted_afters = quick_sort(afters)  
    return sorted_befores + [item] + sorted_afters
```

QUICK SORT CODE

- ▶ Here is the correct quick-sort code in Python:

```
def quick_sort(items):  
    if len(items) == 0: # nothing to sort  
        return []  
    item = choose_item(items)  
    befores_afters = partition(items, item)  
    befores = befores_afters[0]  
    afters = befores_afters[1]  
    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 OTHER RECURSIVE SORT: MERGE SORT

- ▶ The second sort we described 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)  
    sorted2 = merge_sort(part2)  
    return merge(sorted1, sorted2)
```



THE OTHER RECURSIVE SORT: MERGE SORT

- ▶ The second sort we described 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)  
    sorted2 = merge_sort(part2)  
    return merge(sorted1, sorted2)
```
- ▶ We will look at these sorts more carefully in the second half of the course.

RECURSION IN OTHER SETTINGS

- ▶ Recursion occurs in logic and other formal analyses, including the study of natural languages. Here is a possible *grammar* for English sentences:

sentence ::= noun-phrase intransitive-verb

| noun-phrase transitive-verb noun-phrase

subject-phrase ::= noun-phrase | pronoun | proper-noun

noun-phrase ::= article maybe-adjective noun maybe-pp

article ::= the | a

noun ::= dog | cat | couch | boat | man | ...

maybe-adjective ::= adjective maybe-adjective | •

adjective ::= brown | black | couch | boat | ...

intransitive-verb ::= sat | moved | ...

maybe-pp ::= prep noun-phrase

prep ::= on | by | near | ...

...

GRAMMAR FOR ENGLISH

sentence ::= subject-phrase intransitive-verb maybe-adverb maybe-pp

| subject-phrase transitive-verb object-phrase

subject-phrase ::= noun-phrase | pronoun | proper-noun

noun-phrase ::= article maybe-adjective noun maybe-pp

...

GRAMMAR FOR ENGLISH

sentence ::= subject-phrase intransitive-verb maybe-adverb maybe-pp

| *subject-phrase transitive-verb object-phrase*

subject-phrase ::= noun-phrase | pronoun | proper-noun

noun-phrase ::= article maybe-adjective noun maybe-pp

article ::= the | a

...

GRAMMAR FOR ENGLISH

sentence ::= subject-phrase intransitive-verb maybe-adverb maybe-pp

| *subject-phrase transitive-verb object-phrase*

subject-phrase ::= noun-phrase | pronoun | proper-noun

noun-phrase ::= article maybe-adjective noun maybe-pp

article ::= the | a

maybe-adjective ::= adjective maybe-adjective | •

adjective ::= brown | black | chest | boat | ...

...

GRAMMAR FOR ENGLISH

sentence ::= subject-phrase intransitive-verb maybe-adverb maybe-pp

| *subject-phrase transitive-verb object-phrase*

subject-phrase ::= noun-phrase | pronoun | proper-noun

noun-phrase ::= article maybe-adjective noun maybe-pp

article ::= the | a

maybe-adjective ::= adjective maybe-adjective | •

adjective ::= brown | black | chest | boat | ...

noun ::= dog | cat | couch | boat | man | ...

...

GRAMMAR FOR ENGLISH

sentence ::= subject-phrase intransitive-verb maybe-adverb maybe-pp

| *subject-phrase transitive-verb object-phrase*

subject-phrase ::= noun-phrase | pronoun | proper-noun

noun-phrase ::= article maybe-adjective noun maybe-pp

article ::= the | a

maybe-adjective ::= adjective maybe-adjective | •

adjective ::= brown | black | chest | boat | ...

noun ::= dog | cat | couch | boat | man | ...

maybe-pp ::= prep noun-phrase

prep ::= on | by | near | ...

...

GRAMMAR FOR ENGLISH

sentence ::= subject-phrase intransitive-verb maybe-adverb maybe-pp

| subject-phrase transitive-verb object-phrase

subject-phrase ::= noun-phrase | pronoun | proper-noun

noun-phrase ::= article maybe-adjective noun maybe-pp

article ::= the | a

maybe-adjective ::= adjective maybe-adjective | •

adjective ::= brown | black | chest | boat | ...

noun ::= dog | cat | couch | boat | man | ...

maybe-pp ::= prep noun-phrase

prep ::= on | by | near | ...

intransitive-verb ::= sat | moved | ...

maybe-adverb ::= silently | nervously | ...

...

GRAMMAR FOR ENGLISH

sentence ::= subject-phrase intransitive-verb maybe-adverb maybe-pp

| subject-phrase transitive-verb object-phrase

subject-phrase ::= noun-phrase | pronoun | proper-noun

noun-phrase ::= article maybe-adjective noun maybe-pp

article ::= the | a

maybe-adjective ::= adjective maybe-adjective | •

adjective ::= brown | black | chest | boat | ...

noun ::= dog | cat | couch | boat | man | ...

maybe-pp ::= prep noun-phrase

prep ::= on | by | near | ...

intransitive-verb ::= sat | moved | ...

maybe-adverb ::= silently | nervously | ...

...

Can produce:

the cat on the boat moved nervously near the dog by the brown chest

GRAMMAR FOR ENGLISH

sentence ::= *subject-phrase intransitive-verb maybe-adverb maybe-pp*
| *subject-phrase transitive-verb object-phrase*
| *sentence and sentence*

subject-phrase ::= *noun-phrase | pronoun | proper-noun*

noun-phrase ::= *article maybe-adjective noun maybe-pp*

article ::= *the | a*

maybe-adjective ::= *adjective maybe-adjective | •*

adjective ::= *brown | black | chest | boat | ...*

noun ::= *dog | cat | couch | boat | man | ...*

maybe-pp ::= *prep noun-phrase*

prep ::= *on | by | near | ...*

intransitive-verb ::= *sat | moved | ...*

maybe-adverb ::= *silently | nervously | ...*

...

Can produce: *the cat on the boat moved nervously near the dog
by the brown chest and a man sat silently*

GRAMMAR FOR PYTHON

statements ::= statements statement | statement

statement ::= name = expression

| **print(expression)**

| **if expression:**

... statements

else:

... statements

| **while expression:**

... statements

| **def (names):**

... statements

| ...

expression ::= value | name | (expression) | expression op expression | name (expressions) | ...

op ::= + | * | // | - | % | ...

name ::= x | y | count | xs | % | ...

value ::= 42 | True | "hello" | [expressions] | ...

RECURSIVE FUNCTIONS

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

$$1 + 2 + 3 + \dots + 99 + 100 == ????$$

- ▶ 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:

$$1 + 2 + 3 + \dots + (n-1) + n == \text{????}$$

- ▶ 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:

$$(1 + 2 + 3 + \dots + (n-1)) + n == ????$$

- ▶ 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
```

RECURSIVE FUNCTIONS

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

$$(1 + 2 + 3 + \dots + (n-1)) + n == ????$$

- ▶ 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:

$$(1 + 2 + 3 + \dots + (n-1)) + n == ????$$

- ▶ 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:

$$(1 + 2 + 3 + \dots + (n-1)) + n == ????$$

- ▶ 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 imagine 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 imagine 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))
```

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(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 RECURSIVE

- ▶ 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 RECURSIVE

- ▶ 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:

```
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
returning 3

sumUpTo(3) frame

n: 3
returning 6

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
returning 6

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

Outputs 6 to the console.

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)
```

DEMO OF "NOISY" RECURSIVE FUNCTIONS

▶ (in Terminal)

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. **Homework 7**.