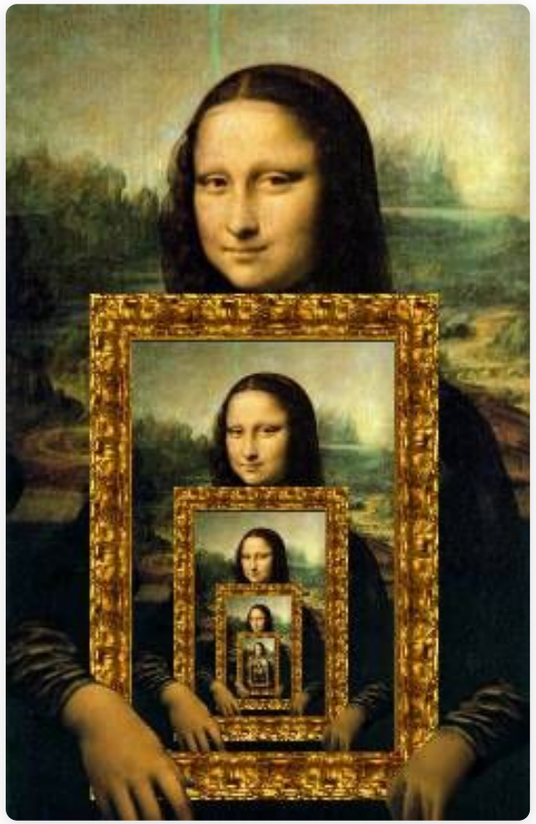
CSCI 145 · Introduction to Computing · Fall 2026

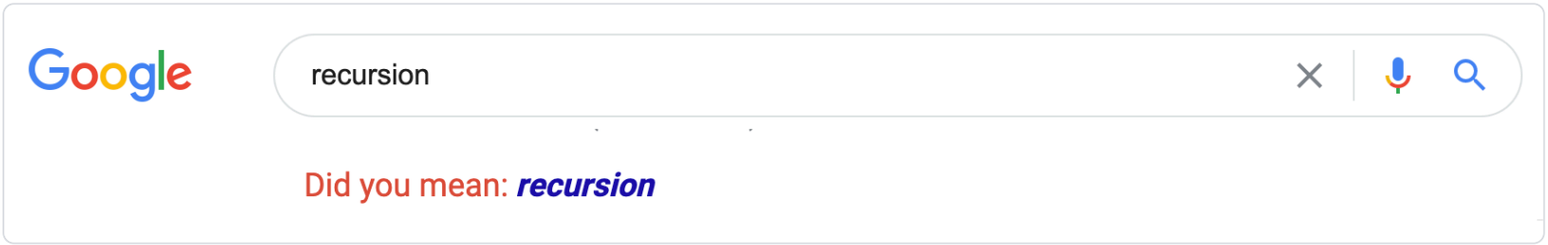
Recursion

Lecture 10 — functions that call themselves

Recursion



A picture that contains
a smaller copy of itself



Today: functions that call
a smaller copy of themselves

Outline

- A function that calls itself
- Drawing with the turtle
- Recursion that returns an answer
- When recursion goes wrong

Part 1 of 4

A function that calls itself

A recursive function

A **recursive function** calls itself

Each call works on a smaller version of the problem

```
def countdown(n):
    if n == 0:
        print("Liftoff!")
    else:
        print(n)
        countdown(n - 1)

countdown(3)
```

OUTPUT

```
3
2
1
Liftoff!
```

- `countdown(3)` prints 3
  Then it calls `countdown(2)`
- `countdown(2)` prints 2
  Then it calls `countdown(1)`, and so on
- `countdown(0)` prints Liftoff!
  It doesn't call itself, so the calls stop

Base case and recursive case

```
def countdown(n):
    if n == 0:
        print("Liftoff!")
    else:
        print(n)
        countdown(n - 1)
```

Base case Recursive case

- The **base case** is handled directly
  No recursive call
- The **recursive case** does one step
  Then it calls itself on a smaller input
- Every recursive function needs both

Tracing `countdown(3)`

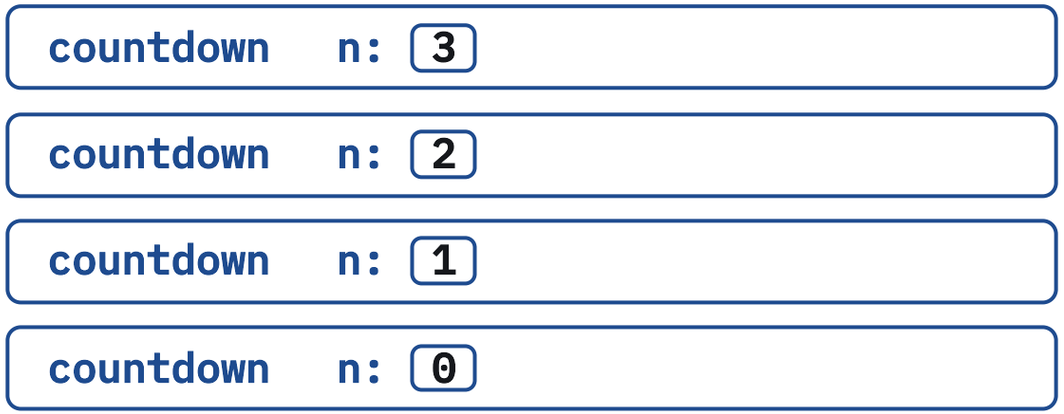
```
def countdown(n):
    if n == 0:
        print("Liftoff!")
    else:
        print(n)
        countdown(n - 1)

countdown(3)
```

OUTPUT

```
3
2
1
Liftoff!
```

CALL STACK



- Each call gets its own frame with its own `n`
- A gray frame has finished
  Nothing is left to do
  after the recursive call

What "smaller" means

Each recursive call gets an input closer to the base case

A smaller number

`n - 1`

3
↓
2
↓
1
↓
0

Base case

`n == 0`

A shorter string

`word[1:]`

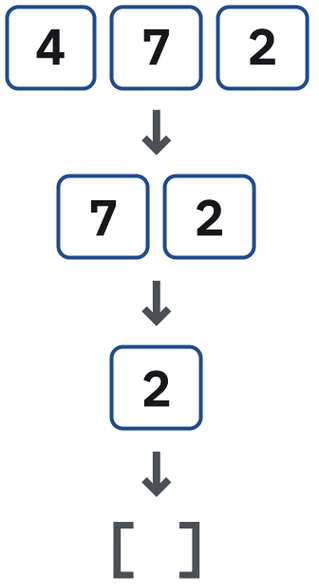
"cat"
↓
"at"
↓
"t"
↓
""

Base case

`word == ""`

A shorter list

`nums[1:]`



Base case

`len(nums) == 0`

What does this print?

```
def show(word):
    if word == "":
        print("Done!")
    else:
        print(word)
        show(word[1:])

show("cat")
```

Go to **slido.com** and enter the code: 3674841

Or join directly here



What does this print?

```
def show(word):
    if word == "":
        print("Done!")
    else:
        print(word)
        show(word[1:])

show("cat")
```

OUTPUT

```
cat
at
t
Done!
```

It prints cat, at, t, Done!

`word[1:]` drops the first letter

Each call prints a shorter word

The empty string is the base case

`show("")` prints Done!

It doesn't call itself, so the calls stop

Part 2 of 4

Drawing with the turtle

The `turtle` module

```
import turtle as t
```

Command	What it does
`t.forward(distance)`	Move forward
`t.backward(distance)`	Move backward
`t.left(angle)`	Turn left by angle degrees
`t.right(angle)`	Turn right by angle degrees
`t.up()`	Lift the pen: moving won't draw
`t.down()`	Put the pen down: moving draws
`t.goto(x, y)`	Move straight to the point `(x, y)`
`t.pencolor(color)`	Change the pen color: `"red"`
`t.done()`	Finish: keep the window open

- A turtle draws as it moves
- It starts at `(0, 0)` the center of the window facing right
- `t.forward(100)` moves 100 pixels

Turtle documentation: docs.python.org/3/library/turtle.html

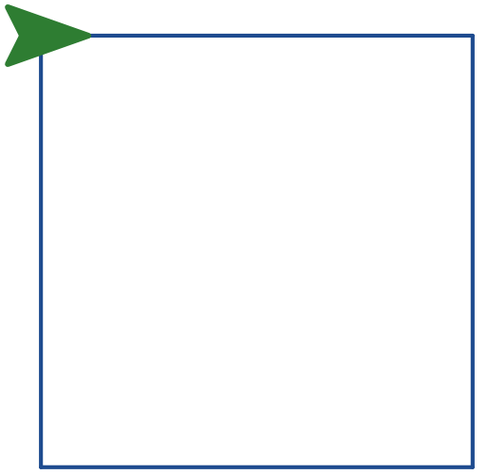
Warm-up: draw a square

```
import turtle as t

def draw_square(side_len):
    for _ in range(4):
        t.forward(side_len)
        t.right(90)

draw_square(100)
t.done()
```

TURTLE WINDOW



- `_` is a loop variable we never use
- Four sides, four right turns

  The turtle ends where it started, facing the same way

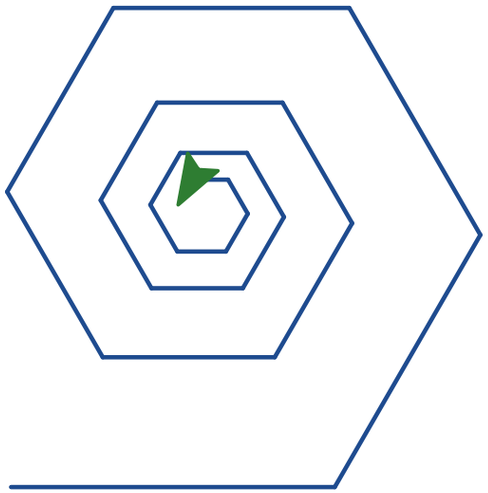
A recursive spiral

```python
import turtle as t

def spiral(length):
    if length > 10:
        t.forward(length)
        t.left(60)
        spiral(length * 0.9)

spiral(100)
t.done()
```

TURTLE WINDOW



CALLS

```
spiral(100)
spiral(90.0)
spiral(81.0)
spiral(72.9)
⋮
spiral(9.84…)  draws nothing
```

- Each call draws one side, turns, then calls `spiral` on a shorter length
- No `else`: when `length` is 10 or less, `spiral` does nothing: the base case

Predict the drawing

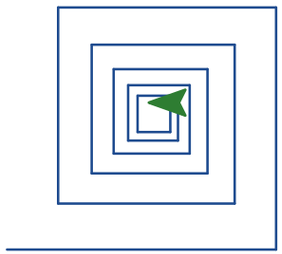
```
def spiral(length):
    if length > 10:
        t.forward(length)
        t.left(    )
        spiral(length * 0.9)
```

Each version puts a different angle in the box

Sketch it, then we'll run it

Same recursion, different turn

`t.left(90)`



`t.left(120)`



`t.left(144)`



Part 3 of 4

Recursion that returns an answer

Writing a recursive function

- Write the docstring first
  What goes in, and what comes out
- Base case: an input small enough to answer directly
- Recursive case: call the function on a smaller input
- **Trust the recursive call**
  Assume it does what the docstring says
- Don't trace every call in your head while you write

Searching a list: `contains`

```
def contains(nums, target):
    """
    Returns True if target is in nums
    Otherwise, returns False
    """
```

```
>>> contains([4, 7, 2], 2)
True
>>> contains([4, 7, 2], 5)
False
```

What are the base cases?

`nums` is empty: False

`nums[0] == target` : True

What is the recursive case?

Search the rest of the list, `nums[1:]`

Return what that call returns

`contains` : two base cases

```
def contains(nums, target):
    if len(nums) == 0:
        return False
    elif nums[0] == target:
        return True
    else:
        return contains(nums[1:], target)
```

Base cases Recursive case

- A function can have more than one base case
  Here, one for each answer
- The recursive case returns whatever the smaller call returns
- The `return` matters
  Without it, the answer is lost

Tracing `contains`

```
def contains(nums, target):
    if len(nums) == 0:
        return False
    elif nums[0] == target:
        return True
    else:
        return contains(nums[1:], target)

found = contains([4, 7, 2], 2)
print(found)
```

OUTPUT

```
True
```

CALL STACK

main	found: True	
contains	nums: [4, 7, 2] target: 2	returns True
contains	nums: [7, 2] target: 2	returns True
contains	nums: [2] target: 2	returns True

- Each call searches a shorter list
- Each call returns
  what the smaller call returned

What does this print?

```
def contains(nums, target):
    if len(nums) == 0:
        return False
    elif nums[0] == target:
        return True
    else:
        contains(nums[1:], target)

print(contains([4, 7, 2], 2))
```

It prints None

The last call returns True
The first call never returns it

```
def contains(nums, target):
    if len(nums) == 0:
        return False
    elif nums[0] == target:
        return True
    else:
        return contains(nums[1:], target)

print(contains([4, 7, 2], 2))
```

✓ Fix: return the smaller call's answer

Now it prints True

Is `word` a palindrome?

```
def is_palindrome(word):
    """
    Returns True if word reads the same
    forward and backward
    Otherwise, returns False
    """
```

```
>>> is_palindrome("racecar")
True
>>> is_palindrome("frog")
False
```

What are the base cases?

Fewer than 2 letters: True

First and last letters differ: False

What is the recursive case?

Check the middle, `word[1:-1]`

Return what that call returns

`is_palindrome`: a loop vs. recursion

LECTURE 7: A LOOP

```
def is_palindrome(word):
    for i in range(len(word) // 2):
        if word[i] != word[len(word) - 1 - i]:
            return False
    return True
```

RECURSION

```
def is_palindrome(word):
    if len(word) < 2:
        return True
    elif word[0] != word[-1]:
        return False
    else:
        return is_palindrome(word[1:-1])
```

Base cases | Recursive case

- `word[1:-1]` drops the first and last letters
- No loop variable and no index arithmetic

  The recursive call checks the middle

Tracing `is_palindrome`

```
def is_palindrome(word):
    if len(word) < 2:
        return True
    elif word[0] != word[-1]:
        return False
    else:
        return is_palindrome(word[1:-1])

print(is_palindrome("racecar"))
```

OUTPUT

```
True
```

CALL STACK

`is_palindrome` **word:** `"racecar"` **returns True**
"r" and "r" match

`is_palindrome` **word:** `"aceca"` **returns True**
"a" and "a" match

`is_palindrome` **word:** `"cec"` **returns True**
"c" and "c" match

`is_palindrome` **word:** `"e"` **returns True**
Fewer than 2 letters

- For `"frog"`, the first call returns False
  `"f"` and `"g"` differ

Part 4 of 4

When recursion goes wrong

When the calls never stop

NO BASE CASE

```
def countdown(n):
    print(n)
    countdown(n - 1)

countdown(3)
```

OUTPUT

```
3
2
1
0
⋮
RecursionError
```

After about 1,000 frames, Python stops: **RecursionError**

A BASE CASE IT SKIPS

```
def countdown(n):
    if n == 0:
        print("Liftoff!")
    else:
        print(n)
        countdown(n - 2)

countdown(3)
```

OUTPUT

```
3
1
-1
⋮
RecursionError
```

`n` jumps from 1 to -1
It never equals 0

FIXED

```
def countdown(n):
    if n <= 0:
        print("Liftoff!")
    else:
        print(n)
        countdown(n - 2)

countdown(3)
```

OUTPUT

```
3
1
Liftoff!
```

`n <= 0` catches every number that's too small

Recursion vs. loops

- A loop repeats with a loop variable

  Recursion repeats by calling the function again

- Both need a way to stop

  A loop condition or a base case

- Today, nothing was left to do after the recursive call

  This is called **tail recursion**

- Next time, work that waits for the recursive call

  `n * factorial(n - 1)`

Activity

1. Download day10_recursion.py and save it in your `cs145/class` folder.
2. Launch Thonny. From the File menu, open `cs145/class/day10_recursion.py`.
3. For each function, decide on the base case and the recursive case first.
4. Write `has_vowel`, `all_even`, and `is_sorted`.
5. Play with the turtle: uncomment `turtle_play()` at the bottom and run the file. Try other angles, lengths, and colors.
6. If you have time, write `staircase` and draw it from `turtle_play`.

To do

- Review this slide deck and the corresponding notes: Notes 9
- Homework 3 is due Thursday 10/15
- No lab this Friday: fall break
- Quiz 5 (Lectures 9–10) is at the start of next class
- Exam 2 is Friday 10/23, covering Lectures 5–9

See practice materials

Next time

Lecture 11 — recursion with pending operations