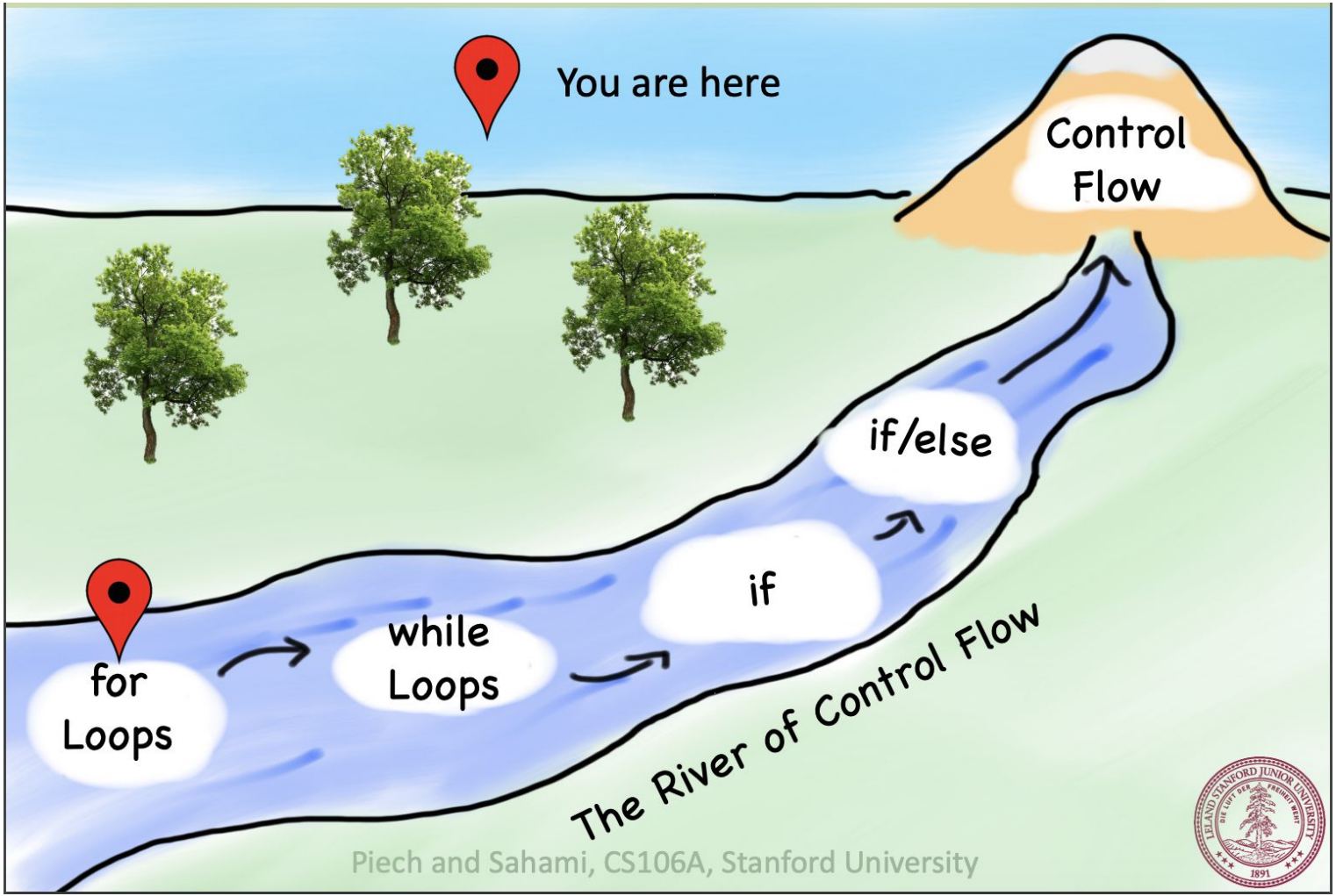
Control Flow Review Session

Will Kenney and Juliette Woodrow

Todays ~Flow~

- if, if/else, and if/elif/else
- while loops
- for loop variations
- range function
- printing vs. returning
- Top Down Decomposition
- Incremental Testing
  - Doctests
- Answer any of your questions
- Practice Problem


You are here
Control Flow
if/else
if
while Loops
for Loops
The River of Control Flow
Piech and Sahami, CS106A, Stanford University

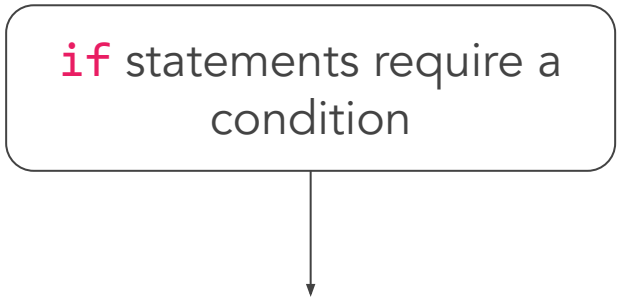
Control Flow Review

Thanks to Brahm Capoor for these awesome slides

Control flow: the steps our program takes

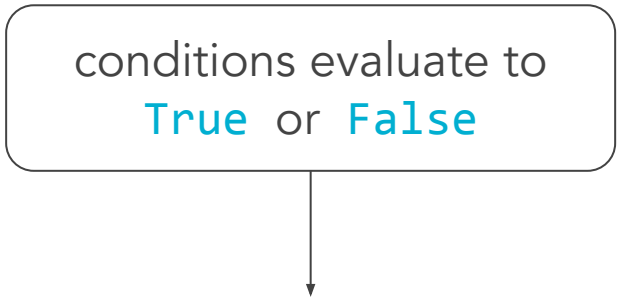
```
if front_is_clear():
  # sick code here
```

Control flow: the steps our program takes



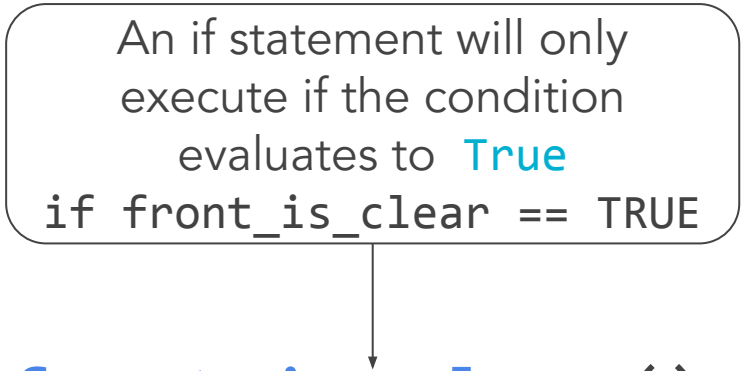

```
if front_is_clear():
    # sick code here
```

Control flow: the steps our program takes




```
if front_is_clear():
    # sick code here
```

Control flow: the steps our program takes



Control flow: the steps our program takes

If the condition is True, the code inside the if statement will happen exactly *once*

```
if front_is_clear():
    # sick code here
```

Control flow: the steps our program takes

Once the code inside the if statement has completed, the program moves on, *even if the condition is still* True

```
if front_is_clear():
    # sick code here
# more sick code here
```

Control flow: the steps our program takes

```
if front_is_clear():
    # sick code here
else:
    # different sick code here
```

Control flow: the steps our program takes

Sometimes we want to do one thing when a condition is True and something else when that condition is False

```
if front_is_clear():
    # sick code here
else:
    # different sick code here
```

Control flow: the steps our program takes

```
if front_is_clear():
    # sick code here
elif beepers_present():
    # other sick code here
else:
    # even more sick code here
```

Control flow: the steps our program takes

Sometimes we want to do one thing when one condition is True and something else when another that condition is True

```
if front_is_clear():
    # sick code here
elif beepers_present():
    # other sick code here
else:
    # even more sick code here
```

Control flow: the steps our program takes

Important Note: If you only use if/elifs, make sure you consider all cases.

```
if front_is_clear():
    # sick code here
elif beepers_present():
    # other sick code here
elif beepers_not_present():
    # even more sick code here
```

Control flow: the steps our program takes

```
while front_is_clear():
    # sick code here
```

Control flow: the steps our program takes

`while` loops also require a condition, which behaves in exactly the same way

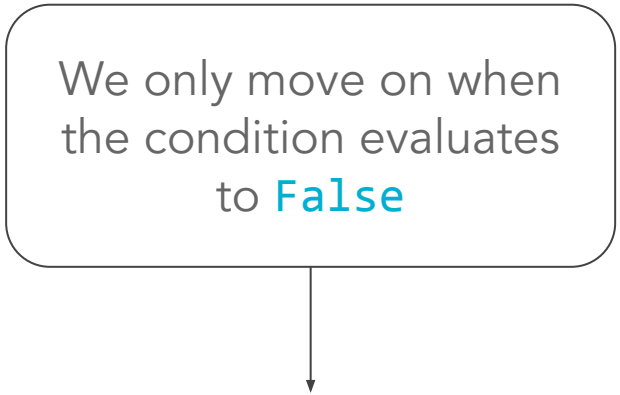
```
while front_is_clear():
    # sick code here
```

Control flow: the steps our program takes

The difference is that the `while` loop repeats so long as the condition is `True`

```
while front_is_clear():
    # sick code here
```

Control flow: the steps our program takes




```
While front_is_clear():
    # sick code here
# more sick code here
```

Control flow: the steps our program takes

```
for i in range(42):
    # sick code here
```

Control flow: the steps our program takes

A `for` loop goes through each of the elements of some collection of things

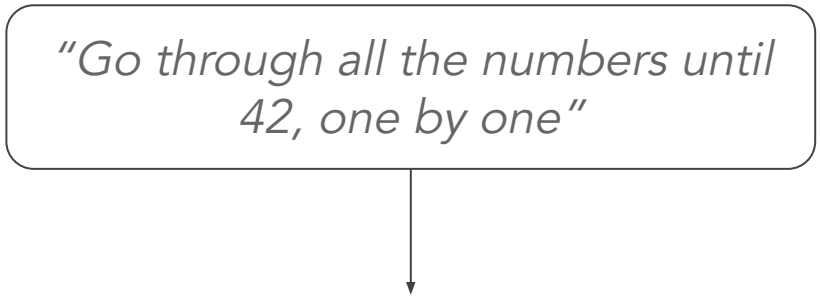
```
for i in range(42):
    # sick code here
```

Control flow: the steps our program takes

The range function gives us an ordered collection of all the non-negative integers below a particular number

```
for i in range(42):
    # sick code here
```

Control flow: the steps our program takes




```
for i in range(42):
    # sick code here
```

Control flow: the steps our program takes

"Go through all the pixels in `image`*, one by one"*

```
for pixel in image:
    # sick code here
```

Control flow: the steps our program takes

The `for` loop ends when we've gone through all the things in the collection

```
for pixel in image:
    # sick code here
# more sick code here
```

Other useful things to know about control flow

`range(42)` - all the numbers between 0 (inclusive) and 42 (exclusive)

`range(10, 42)` - all the numbers between 10 (inclusive) and 42 (exclusive)

`range(10, 42, 2)` - all the numbers between 10 (inclusive) and 42 (exclusive), going up by 2 each time

`range(42, 10, -2)` - all the numbers between 42 (inclusive) and 10 (exclusive), going down by 2 each time.

Printing vs Returning

Programs have a information flow, and a text output area, and those are separate.

- When a function returns something, that's information flowing out of the function to another function
- When a function prints something, that's information being displayed on the text output area (which is usually the terminal)

A useful metaphor is viewing a function as a painter inside a room

- Returning is like the painter leaving the room and telling you something
- Printing is like the painter hanging a painting inside the room
- The painter can do either of those things without affecting whether they do the other thing

Printing is sometimes described as a *side effect*, since it doesn't directly influence the flow of information in a program

Top Down Decomposition

- When faced with a new problem, we want to think about our large, overall problem by breaking it down into smaller and smaller problems
  - Think about the milestones in the assignments!

- Think about making a cake: while the overall outcome is one, cohesive structure, there were various individual steps along the way
  - The icing and the batter are made separately with their own unique components and sub-steps (mixing in various ingredients at various times.)
  - When we code, we can see the end goal (red velvet cake!) but need to break down the problem into smaller, manageable subproblems.

Top Down Decomposition

Think about our Ghost assignment....

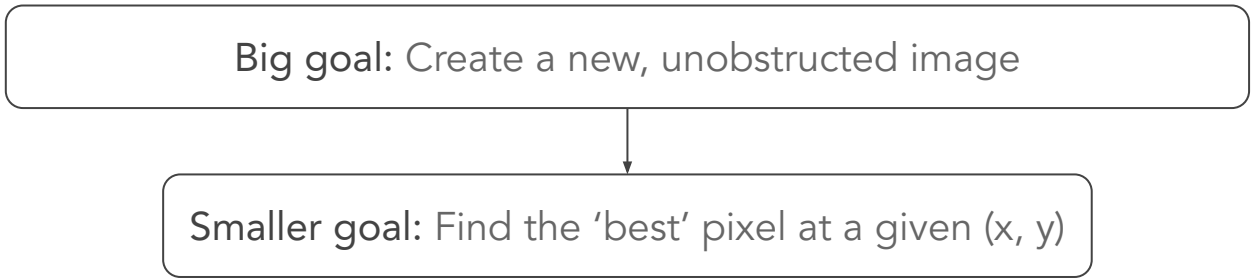
Top Down Decomposition

Think about our Ghost assignment....

Big goal: Create a new, unobstructed image

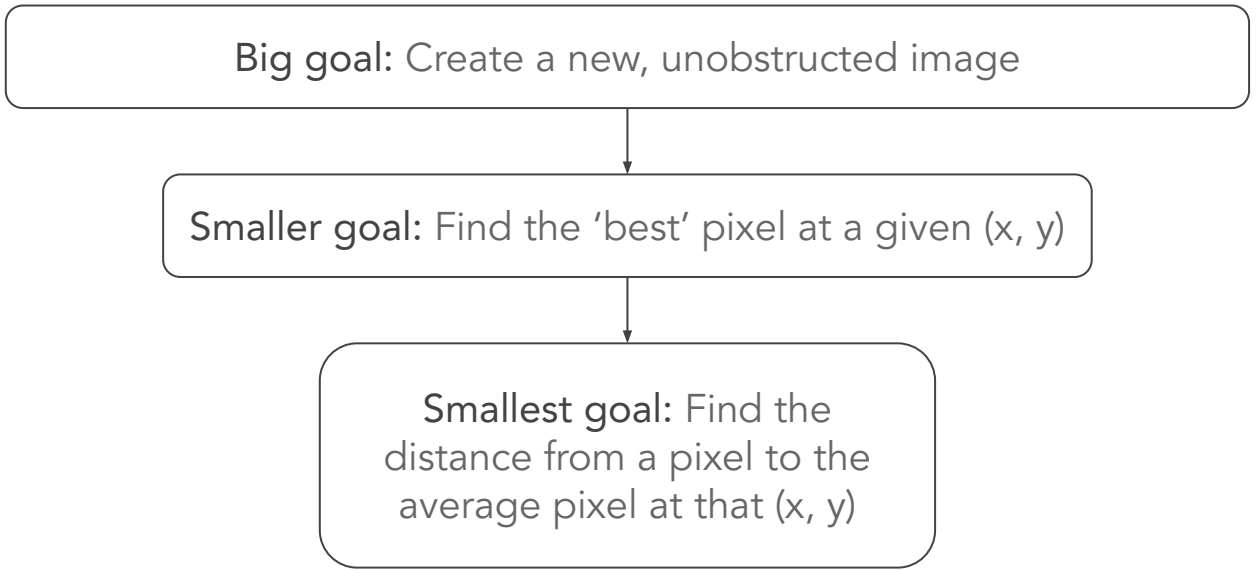
Top Down Decomposition

Think about our Ghost assignment....



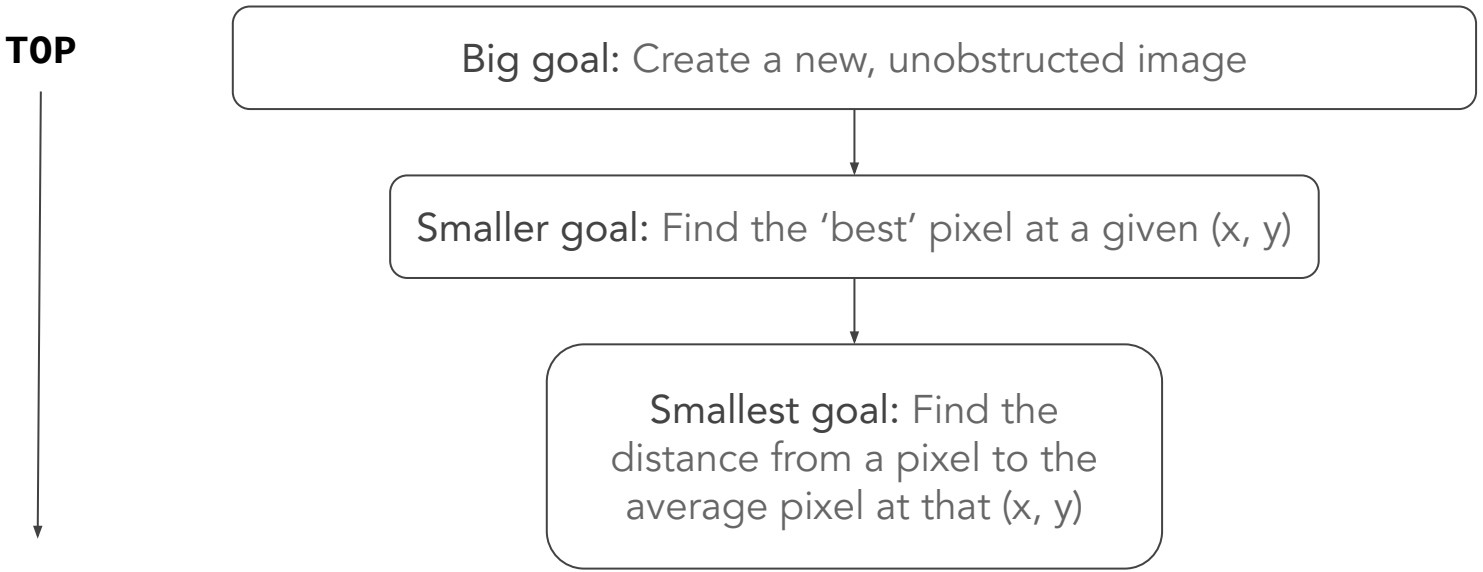
Top Down Decomposition

Think about our Ghost assignment....



Top Down Decomposition

Think about our Ghost assignment....



Incremental Testing

- Before moving on from one function to the next, you want to thoroughly test it
- This way, we can easily identify and eliminate any bugs caused by this function before using it in another function
- Python has a ~cool~ way to test individual functions called: **doctests**

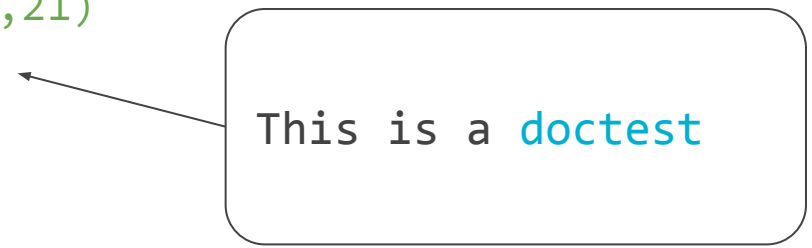
doctests

doctests

```
---
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```

doctests

```
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```



doctests

```
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```

doctests help you test an individual function by running it on certain arguments that you provide and checking the return value.

Composition of a doctest

```
---
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```

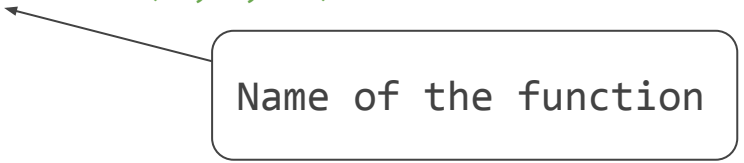
Composition of a doctest

```
---
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```

Three right arrows
followed by a space

Composition of a doctest

```
---
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```



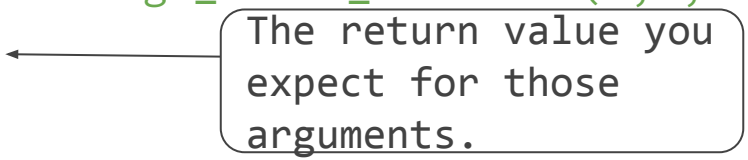
Composition of a doctest

```
---
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```

Real arguments you want to use to test your function

Composition of a doctest

```
---
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```



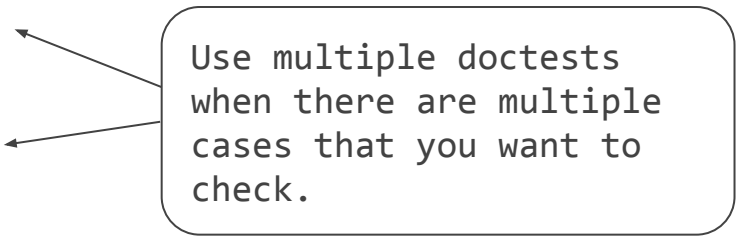
Composition of a doctest

```
---
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```

When you run this doctest, it will check if your program returns 5 when passed in 8, 7, and 21.

You can have multiple doctests for a single function

```
---
def average_minus_smallest(a,b,c):
    """ This function returns the difference btw the average value
    of a,b,c and the smallest value of a,b,c.
    >>> average_minus_smallest(8,7,21)
    5
    >>> average_minus_smallest(0,0,0)
    0
    """
    avg = (a+b+3)/3
    smallest = helper_func_for_min(a,b,c)
    return avg-smallest
```



Tying it all together...

Tying it all together...

- Use top down decomposition to break your program into smaller problems

Tying it all together...

- Use top down decomposition to break your program into smaller problems
- Write a function for each problem

Tying it all together...

- Use top down decomposition to break your program into smaller problems
- Write a function for each problem
- Incrementally test as you write each function
  - AKA use doctests to ensure each function is bug-free before moving on to the next

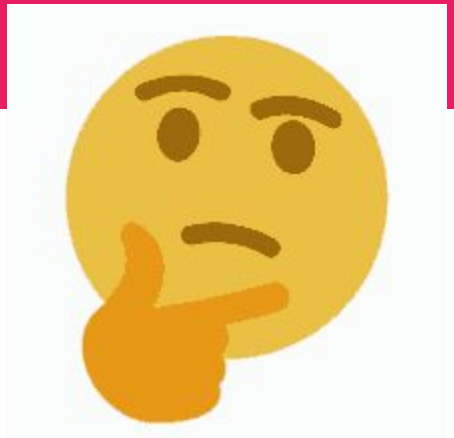
Tying it all together...

- Use top down decomposition to break your program into smaller problems
- Write a function for each problem
- Incrementally test as you write each function
  - AKA use doctests to ensure each function is bug-free before moving on to the next
- Build your entire program

Tying it all together...

- Use top down decomposition to break your program into smaller problems
- Write a function for each problem
- Incrementally test as you write each function
  - AKA use doctests to ensure each function is bug-free before moving on to the next
- Build your entire program
- Become python master 

What questions do you have?



Practice Problem: GCD

Greatest Common Divisor

Greatest Common Divisor

- Write a program that helps a user find the greatest common divisor of 3 numbers

Greatest Common Divisor

- Write a program that helps a user find the greatest common divisor of 3 numbers
- GCD is the largest positive integer that divides each of the integers given

Greatest Common Divisor

- Write a program that helps a user find the greatest common divisor of 3 numbers
- GCD is the largest positive integer that divides each of the integers given
- Your program should use helper functions to break this challenging task into smaller subproblems

Greatest Common Divisor - Our Key Insights

Greatest Common Divisor - Our Key Insights

- Breaking the problem down into smaller problems

Greatest Common Divisor - Our Key Insights

- Breaking the problem down into smaller problems
  - Asking the user for 3 numbers
  - Compute the greatest common divisor
  - Repeat these two tasks until the user enters SENTINEL value

Greatest Common Divisor - Our Key Insights

- Breaking the problem down into smaller problems
  - Asking the user for 3 numbers
  - Compute the greatest common divisor
  - Repeat these two tasks until the user enters SENTINEL value
- Break the program into functions

Greatest Common Divisor - Our Key Insights

- Breaking the problem down into smaller problems
  - Asking the user for 3 numbers
  - Compute the greatest common divisor
  - Repeat these two tasks until the user enters SENTINEL value
- Break the program into functions
  - get_user_input()
    - Asks users for 3 numbers and returns them
  - compute_gcd(a, b, c)
    - Input = 3 integers
    - Returns = the GCD of the 3 integers
  - main()
    - Repeat those two steps while the user’s input != the SENTINEL value

```python
SENTINEL = -1

def main():
    play_game = int(input("Enter any integer to start. Entering -1 will quit. "))
    while play_game != -1:
        a,b,c = get_three_numbers()
        gcd = compute_gcd(a, b, c)
        print('The GCD of '+str(a) + " " + str(b) + " " + str(c) + " is " + str(gcd))
        play_game = int(input("Enter any number (besides -1) to start"))


def get_three_numbers():
    a = int(input("Enter a positive number "))
    b = int(input("Enter a positive number "))
    c = int(input("Enter a positive number "))
    return a, b, c

def compute_gcd(a,b,c):
    if a <= b and a <= c:
        lower = a
    elif b <= a and b <= c:
        lower = b
    else:
        lower = c

    gcd = 1
    for i in range(1, lower + 1):
        if a % i == 0 and b % i == 0 and c % i == 0:
            gcd = i

    return gcd
```