Variables and Decomp

A very exciting day

Stanford | ENGINEERING
Computer Science

Housekeeping

- No class on Tuesday (4th of July), **there will be a recorded lecture to watch before Thursday**

- Wednesday review starts next week, meet Clinton!

- We will have all lecture content for assignment 1 done today!

- Ecy will be lecturing on Thursday

Today

- **Decomp recap + more decomp!**
  - **Revisit fill_world_blue**
  - **Implement challenge problem hurdles**

- Introducing: Variables!
  - What are they
  - How do we use them
  - Challenge problem that uses variables + loops

Calling functions recap

```
def func1():
    # code block A

def func2():
    # code block B
    func1()
    # code block C
```

Running func2:

1. Run code block B
2. Run code block A
3. Run code block C

Syntax note

```
def helper_function(bit):
    # must "take in" bit

def main_bit_problem(filename):
    # required first line
    bit = Bit(filename)
    helper_function(bit) # must "pass in" bit
```

We will talk about this more later, but for now, when decomposing Bit functions:

1. Always take in bit when defining (put "**`bit`**" in parenthesis after **`def function_name`**)
2. Always pass in bit when calling

Why make multiple functions?

- Often a task breaks down into smaller logical tasks like:

  "Go to the farthest wall"

  "Spin in a circle"

  "Paint 3 squares"
- Those tasks can be nicely decomposed into separate functions, and then you could call them from your solution, and it becomes nice and readable!

```
def soln(filename):
    bit = Bit(filename)
    go_to_far_wall(bit)
    spin(bit)
    paint_3(bit)
```

Why make multiple functions?

```
def soln(filename):
    bit = Bit(filename)
    go_to_far_wall(bit)
    spin(bit)
    paint_3(bit)
```

- It is good style to decompose (decomp) your solution
- It makes your code readable for your collaborators (and for Future You)
- It can help you solve a big problem by making you solve several small ones

Why make multiple functions?

- One day you will solve **Big Problems** using code like:

  "Given this giant set of patients' biological and demographic data, what is the average age of occurence of X disease and in patients with X biological marker?"

  "Given this map of natural resources, existing communities, land ownership and historical weather data, where is the best place to build new homes?"

  "Build a tool that scans an email for harmful language or imagery and hides them from the reader"

Why make multiple functions?

- Big Problems are often made up of small problems that other people have already solved!

  "Given this giant set of patients' biological and demographic data, what is the average age of occurence of X disease and in patients with X biological marker?"

  ✅ Can reuse someone's **`read_data_set()`**
  🧠 Need to write **`pull_specific_patients()`**
  ✅ Can reuse someone's **`compute_avg()`**

Why make multiple functions?

- If you solve a Big Problem in a way someone else can read and understand, they can use your solution for their own Bigger Problem!

"Build a tool that scans an email for harmful language or imagery and hides them from the reader "

```
def scan_email():
    separate_text_and_images()
    id_bad_text()
    id_bad_images()
    hide_bad_item()
```

Why make multiple functions?

- If you solve a Big Problem in a way someone else can read and understand, they can use your solution for their own Bigger Problem!

"Build a tool that scans ~~an email~~ **any message** for harmful language or imagery and hides them from the reader "

```
def scan_message():
    if is_email():
        scan_email()
    else:
        # new code
```

Principles of Decomp

- 🧠 A good function does **one conceptual thing**
- 🧠 You know what that thing is from the name
- 🧠 It is **short**: Less than ~10 lines, at most ~3 levels of indents
- 🧠 **Reusable** by you and others, **easy to read** and **modify**
- 🧠 Well commented: **pre- and post-conditions are clear**

Principles of Decomp

- A good function does **one conceptual thing**
- You know what that thing is from the name
- It is **short**: Less than ~10 lines, at most ~3 levels of indents
- **Reusable** by you and others, **easy to read** and **modify**
- Well commented: **pre- and post-conditions are clear**

There are two types of programs:
One is so complex, there is nothing obvious wrong with it.

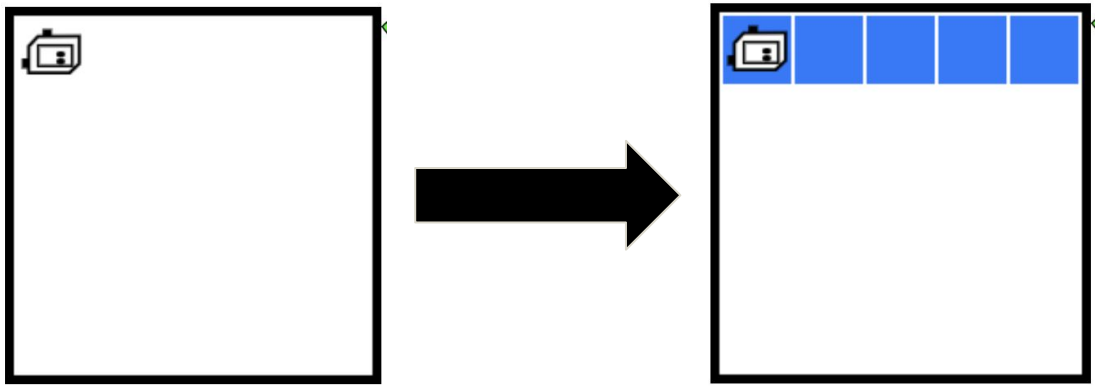
One is so clear, that this obviously nothing wrong with it.

A note on Pre- and Post- Conditions

Recall **`fill_row_blue`**, which we used in **`fill_world_blue`:**

- **Pre conditions**: Assume bit is facing down at left edge

- **Post conditions**: The row bit is on is blue and she is back where she started, facing down



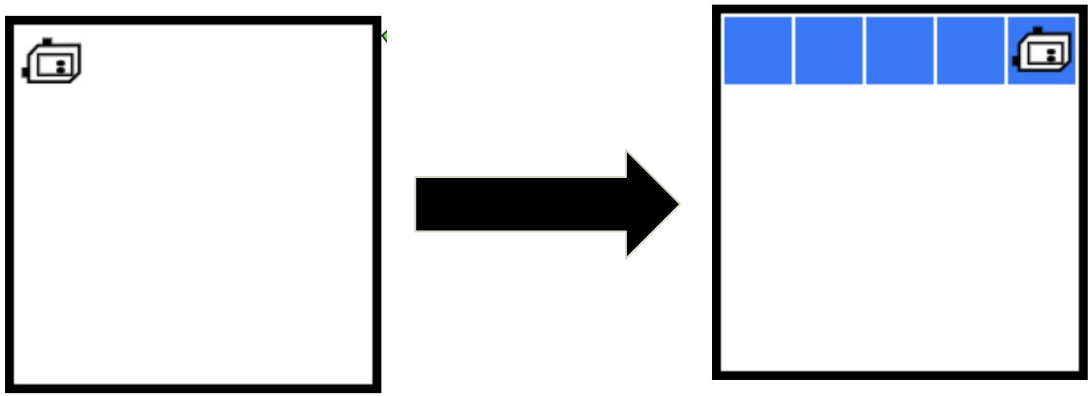
Stanford | ENGINEERING
Computer Science

A note on Pre- and Post- Conditions

Why make bit walk all the way back to where she started?
What if **`fill_row_blue`** left bit at the end of the row?

- **Pre conditions**: Assume bit is facing down at left edge

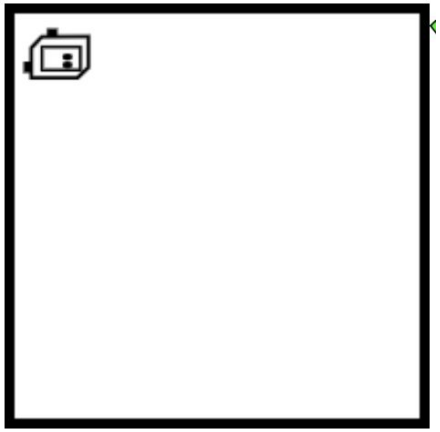
- **Post conditions**: The row bit is on is blue and she is at the left edge of the row, facing down



Stanford | ENGINEERING
Computer Science

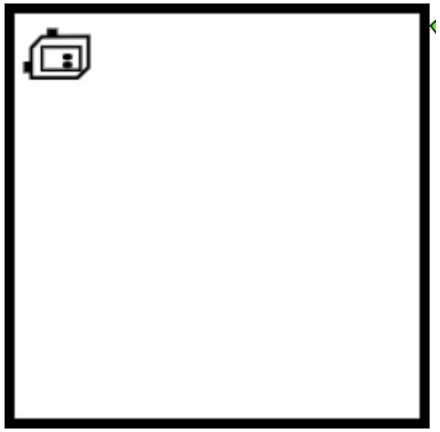
Implement `fill_world_blue` now

```
def fill_world_blue(filename):
    # Old implementation!
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.move()
    fill_row_blue(bit)
```



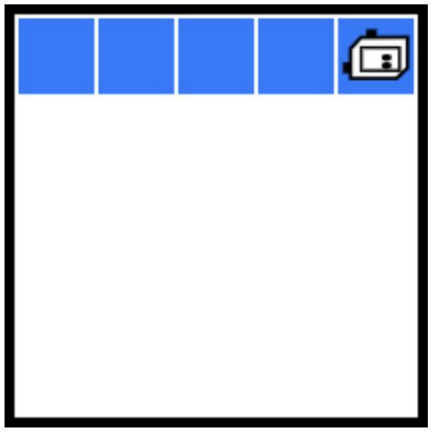
Implement `fill_world_blue` now

```
def fill_world_blue(filename):
    # Old implementation!
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.move()
    fill_row_blue(bit)
```



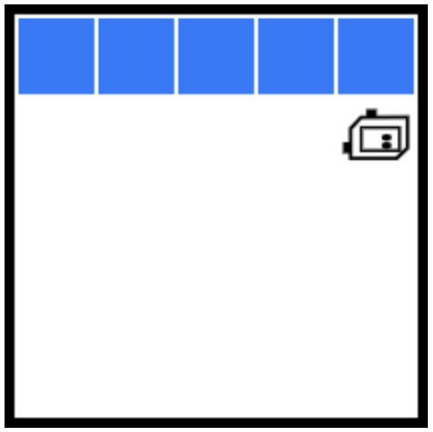
Implement `fill_world_blue` now

```
def fill_world_blue(filename):
    # Old implementation!
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.move()
    fill_row_blue(bit)
```



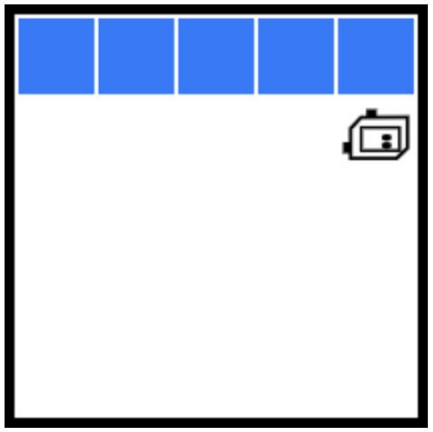
Implement `fill_world_blue` now

```
def fill_world_blue(filename):
    # Old implementation!
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.move()
    fill_row_blue(bit)
```



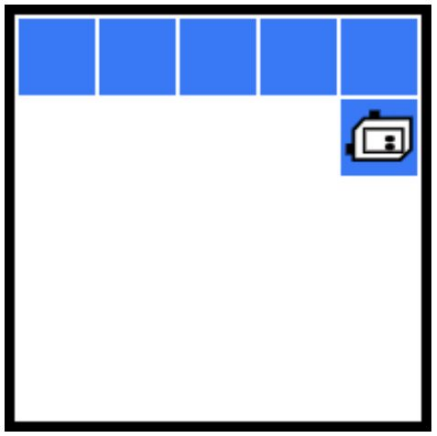
Implement `fill_world_blue` now

```
def fill_world_blue(filename):
    # Old implementation!
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.move()
    fill_row_blue(bit)
```



Implement `fill_world_blue` now

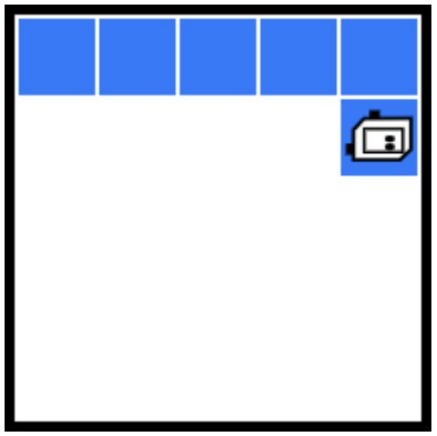
```
def fill_world_blue(filename):
    # Old implementation!
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.move()
    fill_row_blue(bit)
```



Why didn't fill_row_blue fill the whole row?

Implement `fill_world_blue` now

```
def fill_world_blue(filename):
    # Old implementation!
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.move()
    fill_row_blue(bit)
```



Recall pre-condition:

Assume bit is facing down at **left** edge

But we were at the *right* edge when we called `fill_row_blue`

Stanford | ENGINEERING
Computer Science

Implement `fill_world_blue` now

```
def fill_world_blue(filename):
    # New implementation!
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        # the fix: go back to right edge
        bit.right()
        while bit.front_clear():
            bit.move()
        bit.left()
        bit.move()
    fill_row_blue(bit)
```

Which is better?

```
def fill_world_blue(filename):
  bit = Bit(filename)
  while(bit.front_clear()):
      fill_row_blue(bit)
      bit.move()
   fill_row_blue(bit)
```

```
def fill_world_blue(filename):
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.right()
        while bit.front_clear():
            bit.move()
        bit.left()
        bit.move()
     fill_row_blue(bit)
```

Which is better?

```
def fill_world_blue(filename):
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.move()
    fill_row_blue(bit)
```

The original solution is:

- ✅ More readable
- ✅ Shorter
- ✅ Has fewer levels of indents

```
def fill_world_blue(filename):
    bit = Bit(filename)
    while(bit.front_clear()):
        fill_row_blue(bit)
        bit.right()
        while bit.front_clear():
            bit.move()
        bit.left()
        bit.move()
    fill_row_blue(bit)
```

Pre/Post Condition Tips

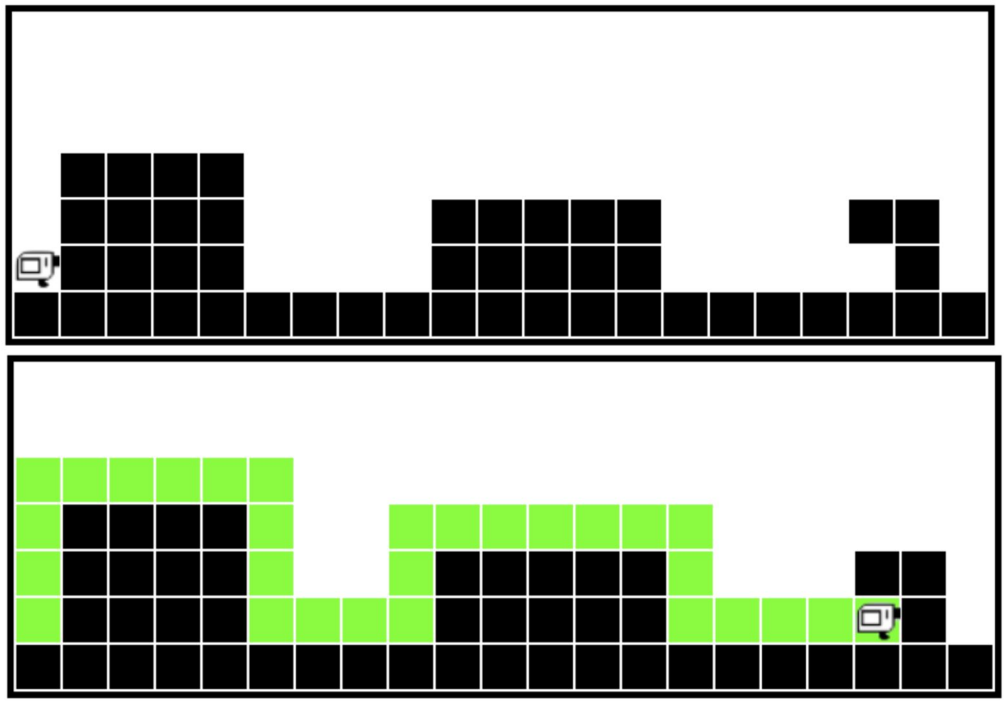
- If your helper function would be called in a loop, try to make the pre and post conditions match up!

- In general with Bit problems, it is best for Bit to end in the same position she started for helper functions, or **at least facing the same direction**

- Getting Bit back to her old position or facing the old direction might add some clunky code to your helper- it is better to have it there than in your main solution function!

Put it all together: hurdles

- Bit starts facing up at the left edge of the world. There are a series of “hurdles” of any height and width. There is a niche with the top blocked at the end of the world.
- Make Bit walk over all the hurdles, painting every square green. Bit should end in the niche facing up



Before

After

Stanford | ENGINEERING
Computer Science

Top Down strategy

1. Start with the big problem and identify a sub problem.

2. Imagine you have a function that solves the sub problem. How should you call it to solve the big problem?

3. Repeat this process to write the code for the sub problem!

Top Down strategy

1. Start with the big problem and identify a sub problem.

Hurdle sub-problem: **`solve_1_hurdle(bit)`**
Pre: Bit is facing up, right in front of a hurdle
Post: Bit is at the next hurdle, facing up

2. Imagine you have a function that solves the sub problem. How should you call it to solve the big problem?

```
def solve_hurdles(filename):
    bit = Bit(filename)
    while bit.front_clear():
        solve_1_hurdle(bit)
```

Stanford | ENGINEERING
Computer Science

Top Down strategy

1. Start with the big problem and identify a sub problem.

Hurdle sub-problem: **`solve_1_hurdle(bit)`**

Pre: Bit is facing up, right in front of a hurdle

Post: Bit is at the next hurdle, facing up

2. Imagine you have a function that solves the sub problem. How should you call it to solve the big problem?

Let's code: hurdles

This is all the content you need for assignment 1

(you don’t need variables for assignment 1)

Today

- ~~Decomp recap + more decomp!~~
  - ~~Revisit fill_world_blue~~
  - ~~Implement challenge problem hurdles~~

- **Introducing: Variables!**
  - **What are they**
  - **How do we use them**
  - **Challenge problem that uses variables + loops**

I am very excited

Variables

- We will continue to build on our definition of variables and “memory” throughout this class

- You will continue to build on your definition of variables and “memory” throughout your career as a computer scientist

- This makes me very emotional

- For now: variables are something that hold a value, and you can update that value as much as you want

Variables

- For now: variables are something that hold a value, and you can update that value as much as you want
- Here is the syntax for making a variable:

```
var_name = #something
```

```
def variable_example(filename):
    bit = Bit(filename)
    color = bit.get_color()
```

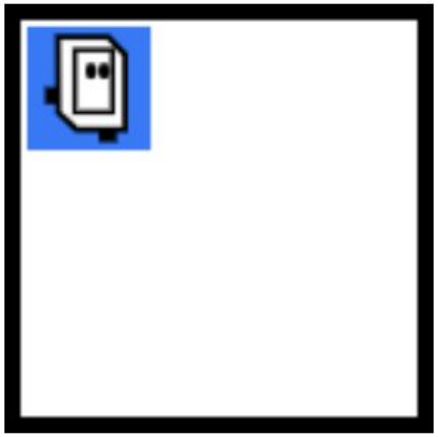
The variable `color` has value `'green'` for this map

Variables

- For now: variables are something that hold a value, and you can update that value as much as you want
- Here is the syntax for making a variable:

```
var_name = #something
```

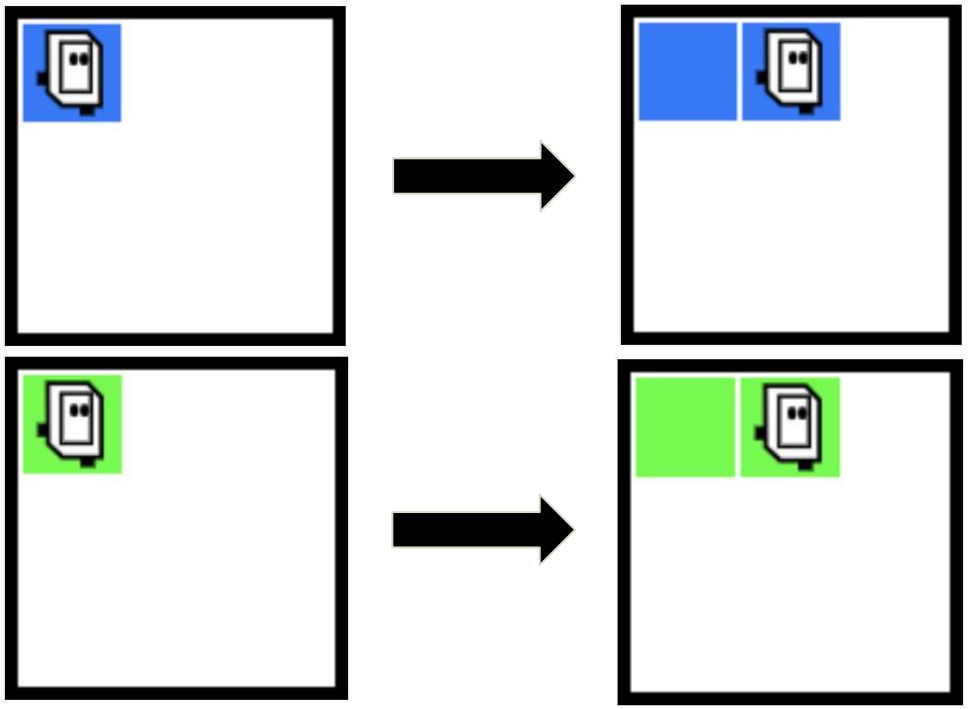
```
def variable_example(filename):
    bit = Bit(filename)
    color = bit.get_color()
```



But on this map, `color` has value `'blue'`

Using Variables

Let's write a function **`copy_cat`** in which Bit starts on a square **with some color**, and then paints the square in front of her the **same color**. Bit will always have an open square in front of her



Stanford | ENGINEERING
Computer Science

Implement copy_cat

```
def copy_cat(filename):
    bit = Bit(filename)
    color = bit.get_color()
    # we have the OG color, what now?
```

Implement copy_cat

```
def copy_cat(filename):
    bit = Bit(filename)
    color = bit.get_color()
    bit.move()
    bit.paint(color)
```

Color will have value `'blue'`, `'red'`, or `'green'`, and we can "pass it into" **`bit.paint`** just like we are used to passing `'blue'`, `'red'`, or `'green'`, into **`bit.paint`**

Stanford | ENGINEERING
Computer Science

Implement copy_cat

```
def copy_cat(filename):
    bit = Bit(filename)
    color = bit.get_color() # green
    bit.move()
    bit.paint(color)
```

`bit.get_color` returns `'green'`,
and we save that in `color`

Implement copy_cat

```
def copy_cat(filename):
    bit = Bit(filename)
    color = bit.get_color() # green
    bit.move()
    bit.paint(color)
```

`bit.get_color` returns `'green'`,
and we save that in `color`

Implement copy_cat

```
def copy_cat(filename):
    bit = Bit(filename)
    color = bit.get_color() # green
    bit.move()
    bit.paint(color)
```

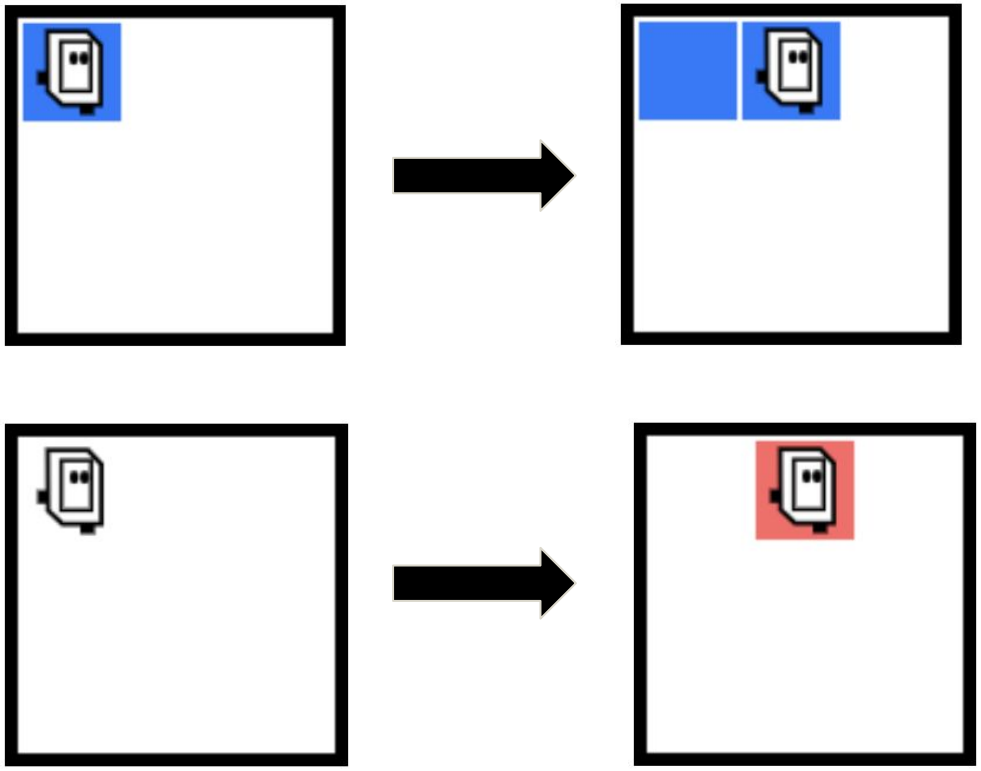
`bit.get_color` returns `'green'`, and we save that in `color`

Now, calling `bit.paint(color)` is like calling `bit.paint('green')`

Stanford | ENGINEERING
Computer Science

Modify copy_cat

Now, Bit might not start on a square with a color to copy. If that's the case, paint the next square red



Implement copy_cat_mod

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = 'red' # default color to value 'red'
    if not bit.get_color() == None:
        # if there is indeed a color here
        # change the value of color!
        color = bit.get_color()
    bit.move()
    bit.paint(color)
```

Implement copy_cat_mod

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = 'red' # default color to value 'red'
    if not bit.get_color() == None:
        # if there is indeed a color here
        # change the value of color!
        color = bit.get_color()
    bit.move()
    bit.paint(color)
```

`color` currently has value `'red'`

Implement copy_cat_mod

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = 'red' # default color to value 'red'
    if not bit.get_color() == None:
        # if there is indeed a color here
        # change the value of color!
        color = bit.get_color()
    bit.move()
    bit.paint(color)
```

`color` currently has value `'red'`

Implement copy_cat_mod

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = 'red' # default color to value 'red'
    if not bit.get_color() == None:
        # if there is indeed a color here
        # change the value of color!
        color = bit.get_color()
    bit.move()
    bit.paint(color)
```

`color` now has value `'green'`

Implement copy_cat_mod

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = 'red' # default color to value 'red'
    if not bit.get_color() == None:
        # if there is indeed a color here
        # change the value of color!
        color = bit.get_color()
    bit.move()
    bit.paint(color)
```

`color` now has value `'green'`

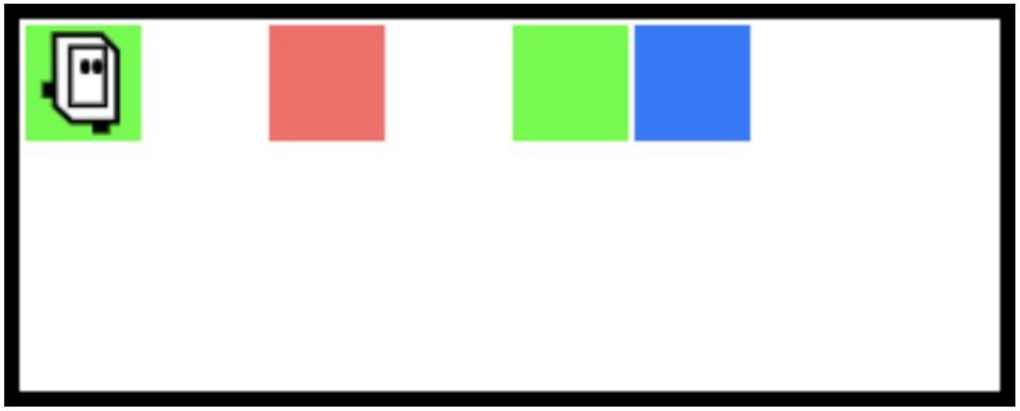
Implement copy_cat_mod

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = 'red' # default color to value 'red'
    if not bit.get_color() == None:
        # if there is indeed a color here
        # change the value of color!
        color = bit.get_color()
    bit.move()
    bit.paint(color)
```

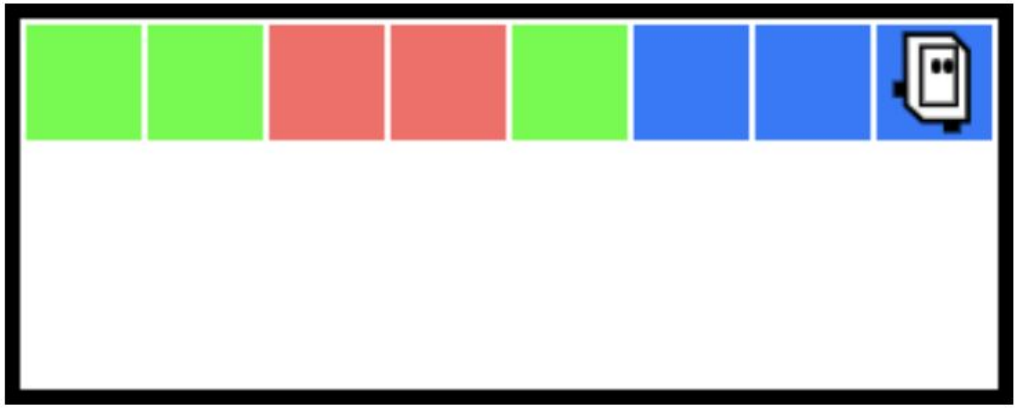
`color` now has value `'green'`

Variables and loops

Let's write a function **`copy_cat_row`** in which Bit starts on a square that **has a color**. Bit will paint the row that color until she finds a new color, then she'll switch to that one.



Before



After

Implement copy_cat_row

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    # first, let's save our starting color
```

Implement copy_cat_row

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = bit.get_color() # starting color

    # Next, let's add in code to walk to each
    # square
```

Implement copy_cat_row

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = bit.get_color() # starting color
    while bit.front_clear():
        bit.move() # move to square to be painted

        # Now, we need code to either paint a blank
        # square … what to do if the square isn't
        # blank?
```

Implement copy_cat_row

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = bit.get_color() # starting color
    while bit.front_clear():
        bit.move() # move to square to be painted
        if bit.get_color() == None:
            # if blank, paint the blank square
            bit.paint(color)
        else:
            # otherwise, change color
            color = bit.get_color()
```

Implement copy_cat_row

```
def copy_cat_mod(filename):
    bit = Bit(filename)
    color = bit.get_color() # starting color
    while bit.front_clear():
        bit.move() # move to square to be painted
        if bit.get_color() == None:
            # if blank, paint the blank square
            bit.paint(color)
        else:
            # otherwise, change color
            color = bit.get_color()
```

Tricking the experimental server

- The experimental server only has a limited number of problems on it, which it makes it hard to test custom Bit puzzles (like copy_cat_row)
- But you can still do it!

1. Pick a problem with maps that you like (e.g. big open maps or maps with a lot of obstacles)
2. If you need the world to look a certain way before solving a problem, write code to make Bit set it up, then get her in the position you want
3. Write a function to solve your custom Bit problem, and Run see if it worked! Ignore what the server says- just eyeball verify that Bit did what you wanted!

Checking copy_cat_row on go_rgb

If time: beloved

Recap

- Decomp is powerful stuff! It helps you solve big problems and it helps make your code more readable

- Top-Down decomp strat: Take the big problem and identify one smaller subtask. Solve the big problem assuming you have a function that does the subtask. Repeat on subtask!

- Variables are ways for us to remember a value, like the color of a square, throughout our code.

- We can change variables throughout our code to make it dynamic

- Remember to watch lecture video before Thursday!