Welcome to CS106A

We are going to have fun!

Stanford | ENGINEERING
Computer Science

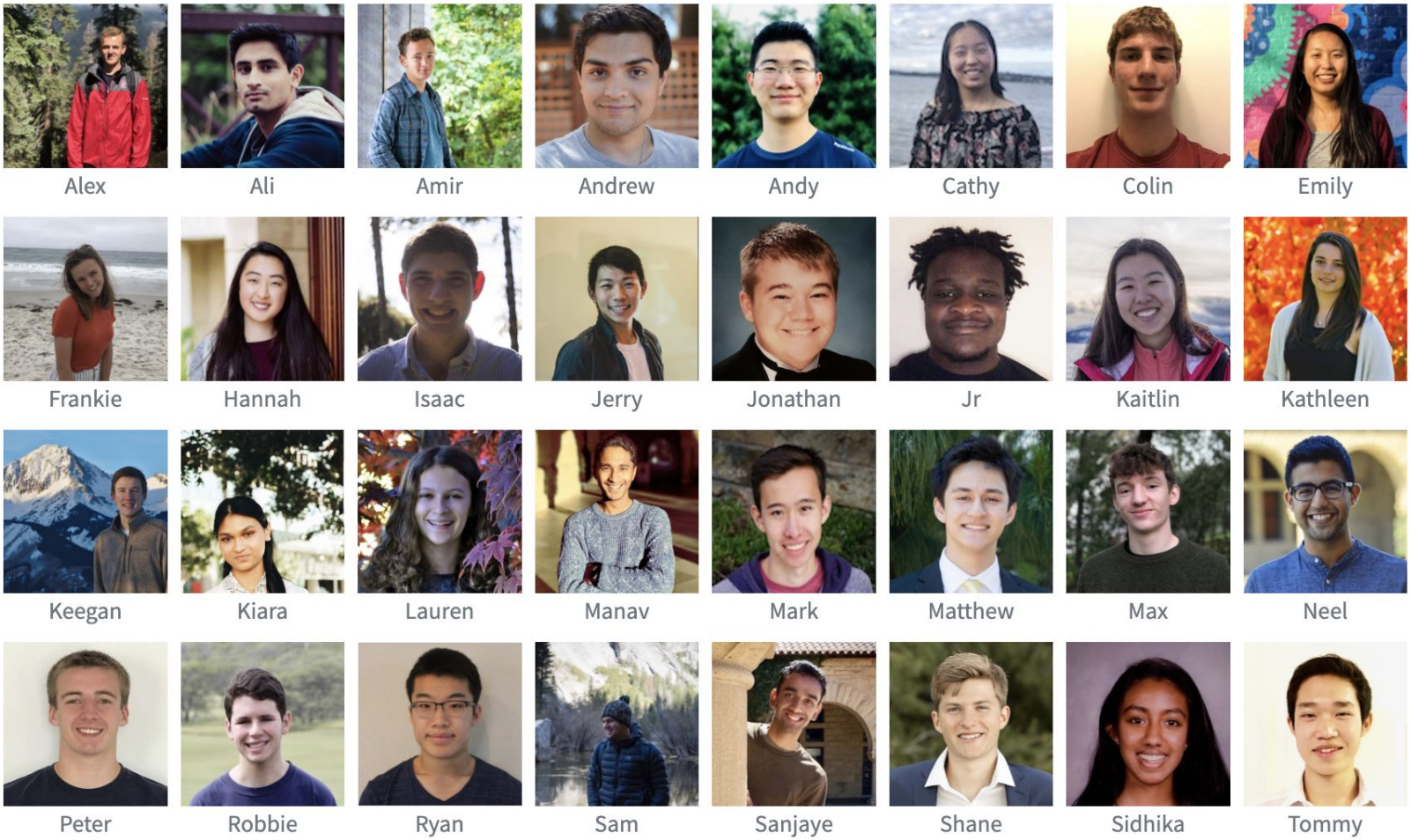
Lecturer: Frankie Cerkvenik

- Stanford undergrad in CS (systems) and MS in CS (Computer and Network Security)
- Interests: CS Education and the future of safety on the internet
- From Minnesota, I love canoeing and fishing and hockey :)

Head TA: Ecy King

- Stanford SymSys undergrad (Human-Centered AI)
- currently a CS coterm (HCI)!
- born in Scotland; raised in Fresno/Clovis, CA; family from Sierra Leone
- Interests: Educational Empowerment, Fractal Gridding, psychology, doodling, music, writing, racquet sports

Section Leaders



*these are SLs of years past…spot frankie!

Course Mechanics

What you need to do

- Lecture
  - Just show up!
  - Tues, Thurs, Fri 1:30-245 in SkillAud
- Optional Weekly Review Sessions (starting next week)
  - Wednesdays 1:30-2:45 in SkillAud
- Section
  - Weekly 50-min section with a section leader
  - Sections signups on class webpage **not Axess**
  - Signups close **today at 5** and start **this week**
- Assignments
  - 6 assignments (about 1/wk) throughout the quarter
- Exams
  - Midterm 7/26 in the afternoon/evening
  - **Final exam 8/18 at 3:30pm**

Sections and IGs

- Lectures go fast! We don't expect you to retain all of it
- Sections are an opportunity to go over problems with a section leader
- Extra section problems are given for you to go over on your own
- **IGs = Interactive Grading sessions: you will meet with your section leader after most assignments to go over your code!**

Grading Breakdown

- 6 programming assignments (65%)
  - Assignments turned in on time receive a 2% on time bonus
  - After the due date, each assignment will have a full-credit "grace period" of 24 hours unless documented otherwise on the assignment.
  - Contact Frankie and Ecy if you need to arrange extra time on homework for extenuating circumstances
- Midterm exam (10%)
- Final Exam (15%)
- Section and IG attendance and participation (10%)

Assignment Grading

++	A submission so good it “makes you weep”
+	Exceeds requirements (and has great style)
✅ +	Satisfies all requirements, with good functionality and style
✅	Meets the requirements, but with small problems
✅-	Has some somewhat serious problems
-	Is worse than that, but shows real effort
--	Better than nothing

Office Hours

- Frankie's office hours - Fridays 3 - 4:30 in Durand
  - Get conceptual help and talk about assignments, lecture topics, or anything else on your mind
- Ecy's office hours - Tuesdays 3 - 4:30 in Durand
  - Group office hours about assignments, concepts, and life
- LAIR
  - In **Durand 353 Thurs-Sun 5-9pm starting today**, schedule posted on website
  - Get conceptual and debugging help from section leaders!
- Frankie and Ecy's tea time - after lecture on Th outside
  - Come talk to us about anything outside of CS106A!

Ed Discussion Forum

- An online discussion forum for asking questions
- Great for:
  - Clarifying questions about lecture or section problems
  - Clarifying or conceptual homework questions
  - Logistical questions
  - Conceptual or what if… questions
- Medium good for:
  - Debugging questions
  - Please do not post code publicly!

Exams

- Midterm 7/26 in the afternoon/evening
  - If you have an immovable academic conflict, email Frankie **before Friday** to arrange an alternate time
  - If you emailed before the first day of class, email again!
- Final 8/18 at 3:30pm-6:30pm: **no alternate exam**
- Please send OAE letters to Frankie **by Friday**

What is CS106A?

What is Computer Science?

"Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes. Science is not about tools, it is about how we use them and what we find out when we do."

— Michael Fellows and Ian Parberry

"Instead of imagining that our main task is to instruct a computer what to to, let us concentrate rather on **explaining to human beings what we want a computer to do.**"

— Don Knuth

CS106A learning goals

1. Learn about what computers can do
   - Computers are not magic boxes! They are problem-solvers!
   - They can solve specific problems using a specific set of instructions (code)

CS106A learning goals

1. Learn about what computers can do
   - Computers are not magic boxes! They are problem-solvers!
   - They can solve specific problems using a specific set of instructions (code)
2. Learn how to use computers to solve problems
   - Explore fundamental techniques in computer programming.
   - Develop good software engineering style.
   - Gain familiarity with the Python programming language.

CS106A learning goals

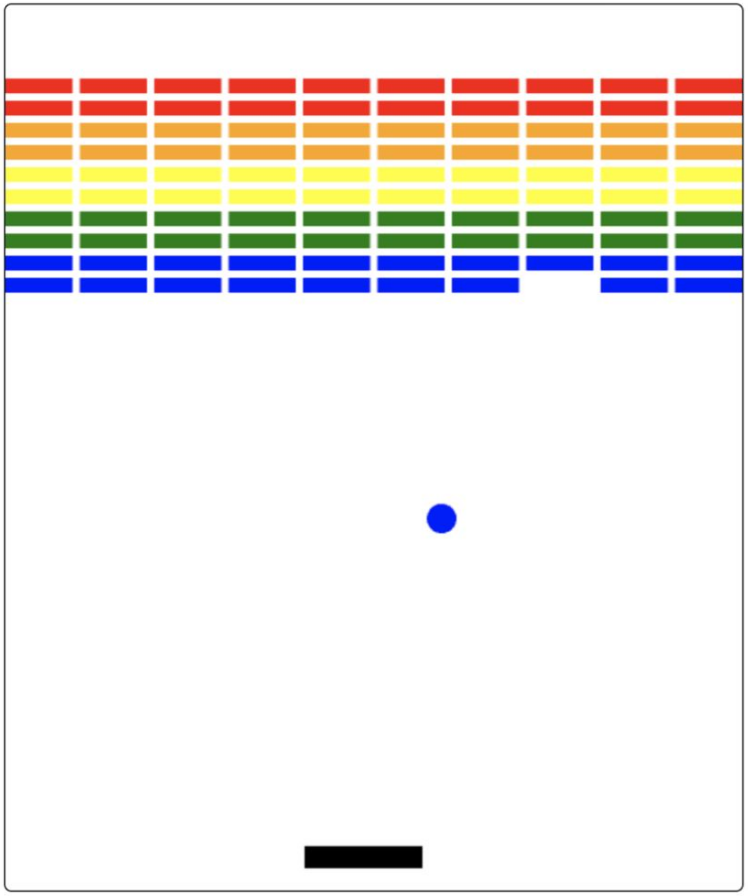
1. Learn about what computers can do
   - Computers are not magic boxes! They are problem-solvers!
   - They can solve specific problems using a specific set of instructions (code)
2. Learn how to use computers to solve problems
   - Explore fundamental techniques in computer programming.
   - Develop good software engineering style.
   - Gain familiarity with the Python programming language.
3. Frankie's agenda: get you to love it enough to keep going!

CS106A Topics

- Image processing

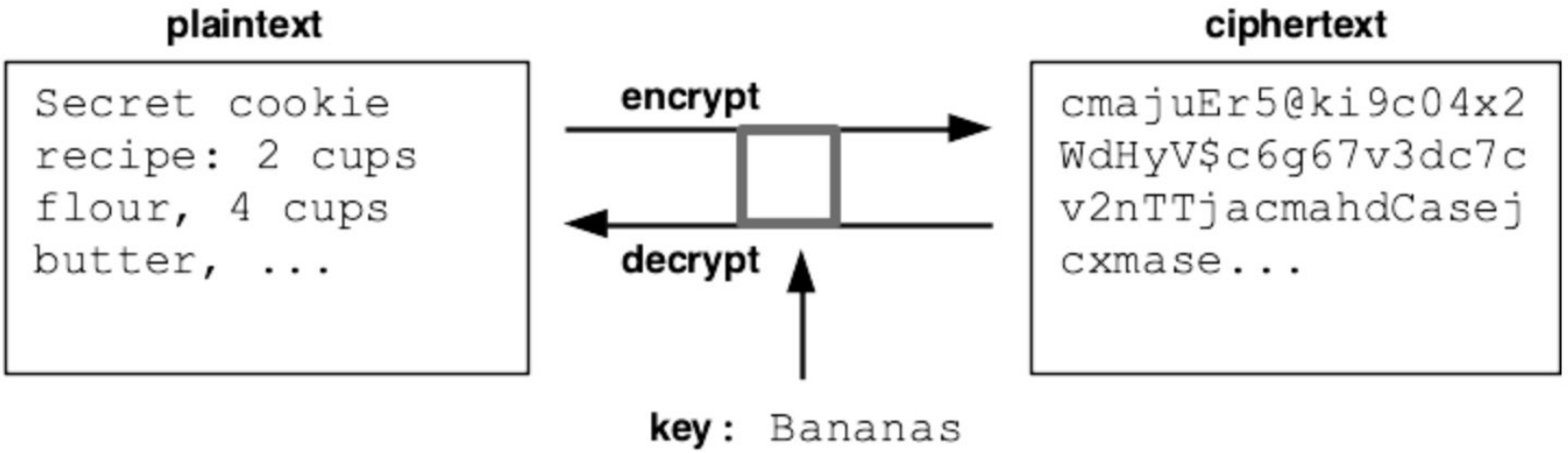
CS106A Topics

- Image processing
- Animation and games



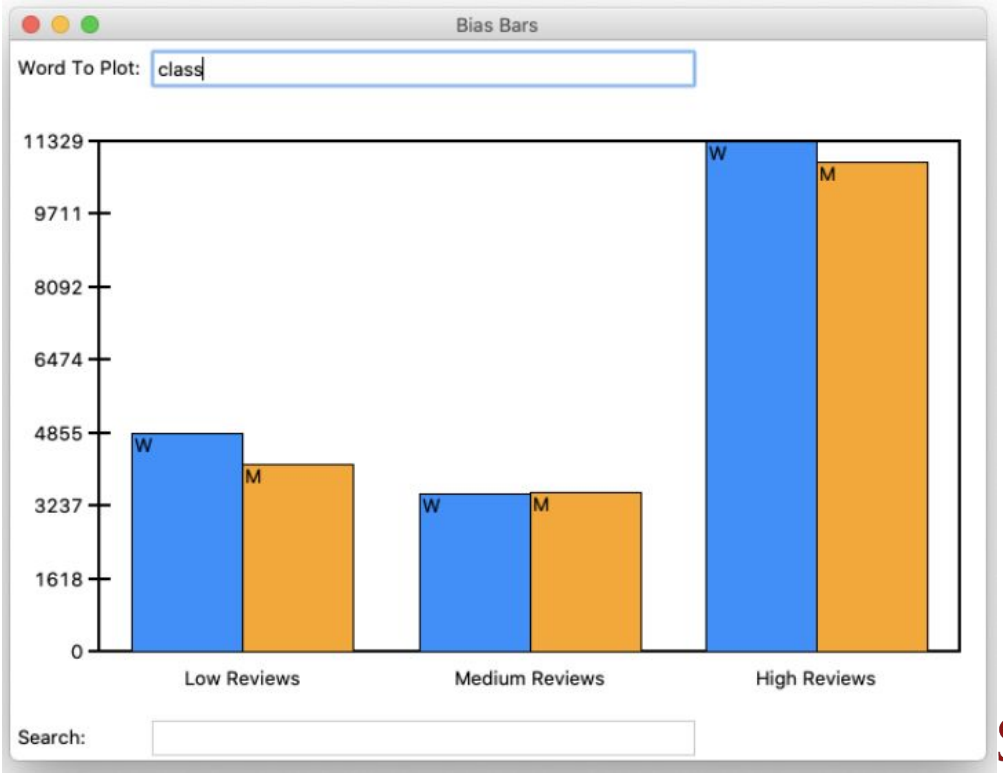
CS106A Topics

- Image processing
- Animation and games
- Text-based problems (ciphers and language processing)



CS106A Topics

- Image processing
- Animation and games
- Text-based problems (ciphers and language processing)
- Data Processing and analytics



CS106A Topics

- Image processing
- Animation and games
- Text-based problems (ciphers and language processing)
- Data Processing and analytics
- Sneak peek at how the internet works!

The sky is the limit!

Announcements

- Section signups are out on the website and will be due **tonight at 5pm**

- Assignment 0 (not graded) is released and due **this Friday**

- Assignment 1 will be released **Wednesday** and is due **next Friday**

- If you would like to switch to 106B, try to decide **this week**, as both classes have assignments due next Friday.

Let’s dive in!

Code looks weird at first

- Computers solve very **specific** problems very well
- And are otherwise very **bad** at solving problems
- Particularly, computers need their instructions in a **very specific format** (code)
- We need to start with a bunch of weird and intimidating **syntax rules** and **definitions**
- There are **many** but they are all very **simple**

Definitions

- **Program**: A collection of code for a computer to run, which may take in specific inputs like a file name or a number
- **Function**: A logical unit of code that may be used and reused in a program

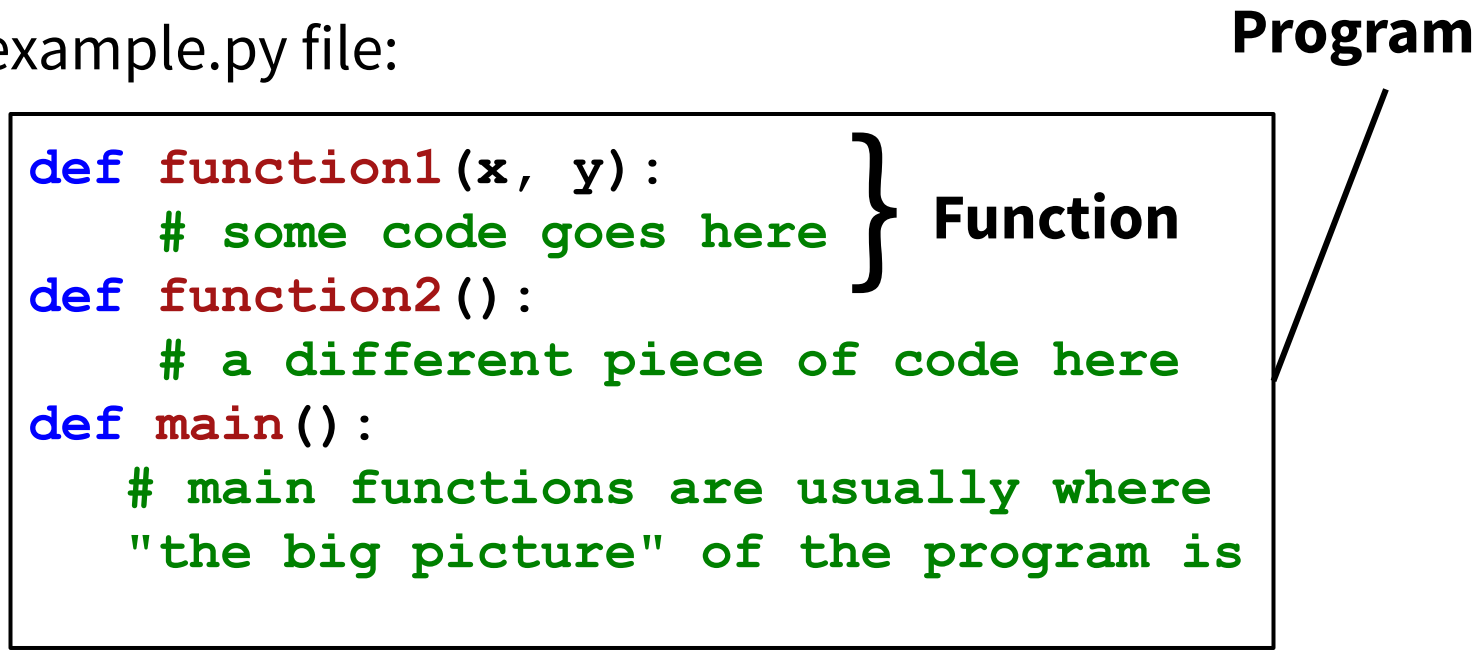
Computer **programs** are made up of different **functions**

Like **organisms** are made of different types of **cells**

Python Syntax

- A python program with python functions would be written in a file named something like “example.py”

example.py file:



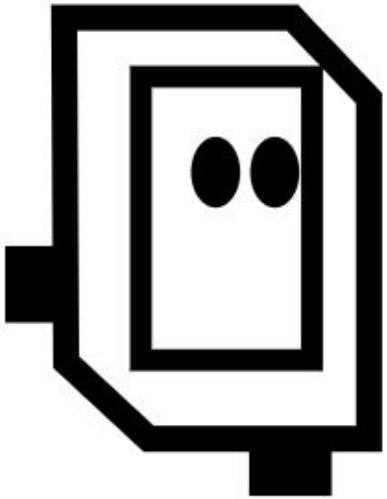
Python Syntax

example.py file:

```
# Lines starting with # are comments and ignored by
# the computer
def function1():
    # Underneath def function_name(): is the code for
    # what should happen when the function is used

    # This code won't run until the function is called

def main():
    function1() # functions are "called" like this!

    # The main function is usually where "the big
    # picture" of the program is - more on that later
```

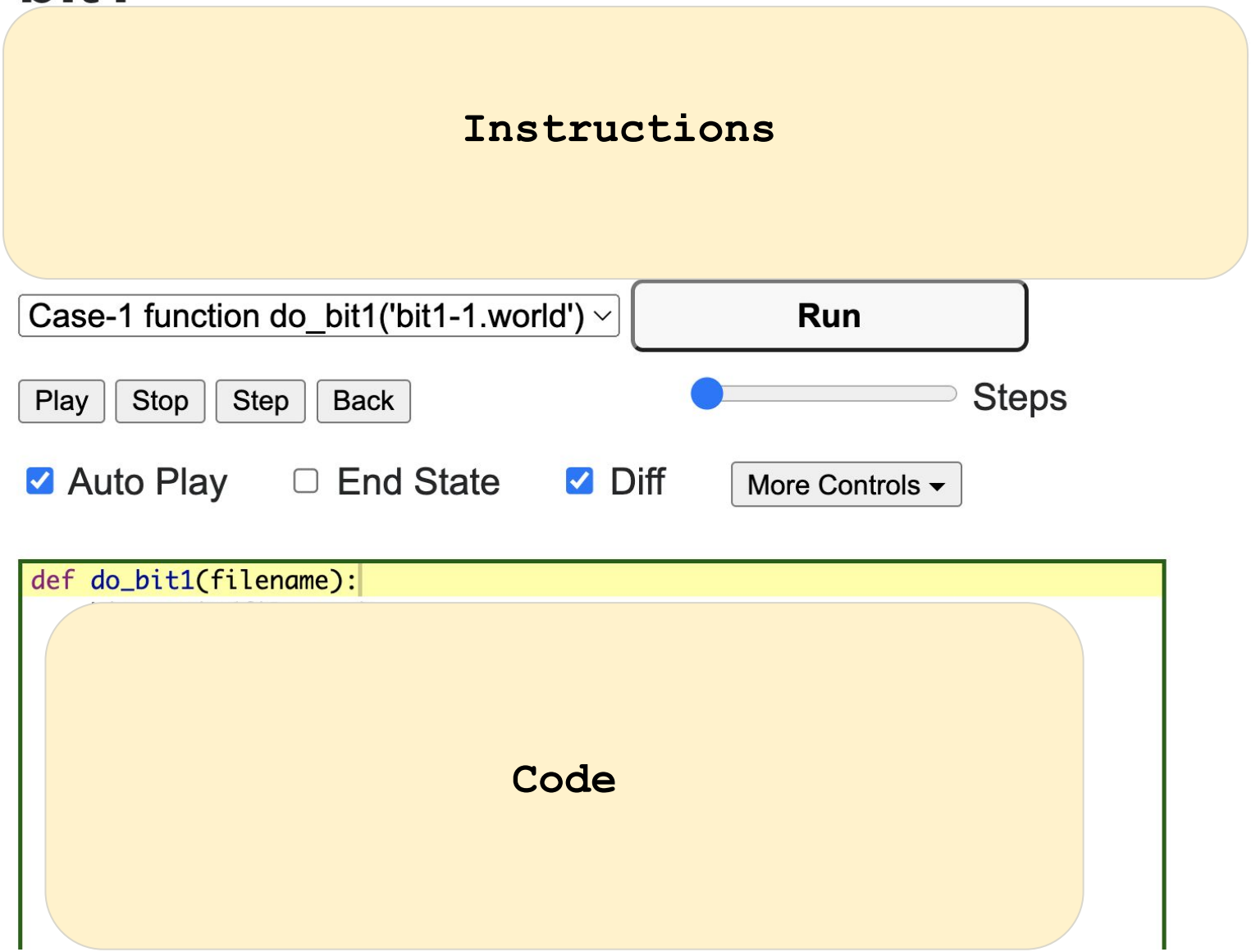
Bit and the Experimental Server

- We will learn more about files on our computer and the main function soon
- To get started, we will write and run code on our experimental server (thanks Nick Parlante!)
- Our programs will focus on a little robot named Bit



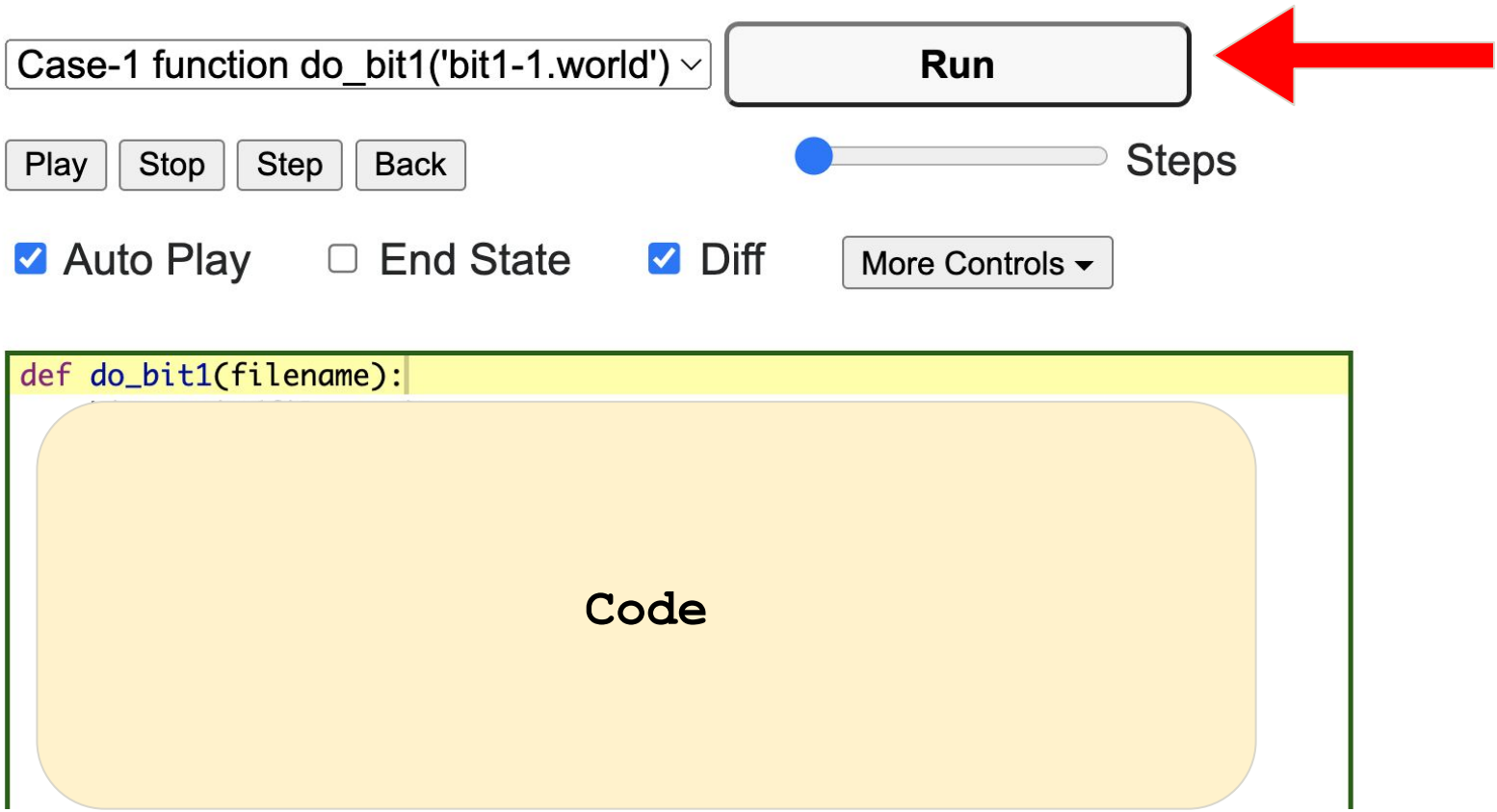
do_bit1

bit1


Instructions
Case-1 function do_bit1('bit1-1.world')
Run
Play
Stop
Step
Back
Steps
Auto Play
End State
Diff
More Controls
def do_bit1(filename):
Code

bit1

This code is complete - run it to see how it works. Bit starts facing the right side of the world. The square in front of bit and the square below that are clear. Move bit forward and paint that square blue. Then paint the square below that green.



What happened?

bit1

This code is complete - run it to see how it works. Bit starts facing the right side of the world. The square in front of bit and the square below that are clear. Move bit forward and paint that square blue. Then paint the square below that green.

Case-1 function do_bit1('bit1-1.world') | Run

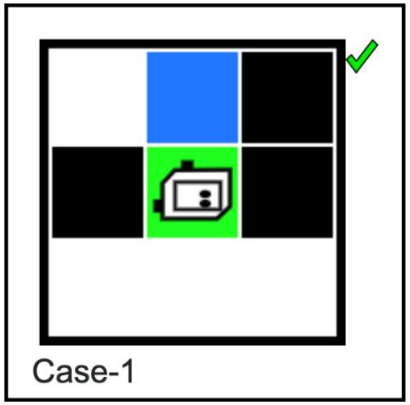
Play | Stop | Step | Back | Steps

Auto Play | End State | Diff | More Controls

```
def do_bit1(filename):
    bit = Bit(filename)
    bit.move()
    bit.paint('blue')
    bit.right()
    bit.move()
    bit.paint('green')
```

Ran this code…

Case-1: **1 Correct** ✓ 1c | Run All



…To make bit do this!

Code line by line

```
def do_bit1(filename):
    bit = Bit(filename) # Set up world
    bit.move()
    bit.paint('blue')
    bit.right()
    bit.move()
    bit.paint('green')
```

bit1

This code is complete - run it to see how it works. Bit starts facing the right side of the world. The square in front of bit and the square below that are clear. Move bit forward and paint that square blue. Then paint the square below that green.

Case-1 function do_bit1('bit1-1.world') | Run

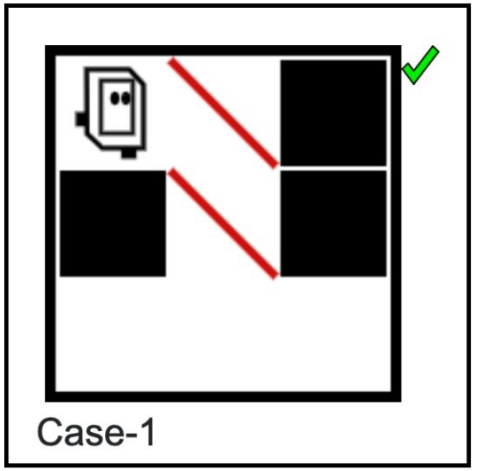
Play | Stop | Step | Back | Steps

Auto Play | End State | Diff | More Controls

```
def do_bit1(filename):
    bit = Bit(filename)
```

Uncheck Auto Play

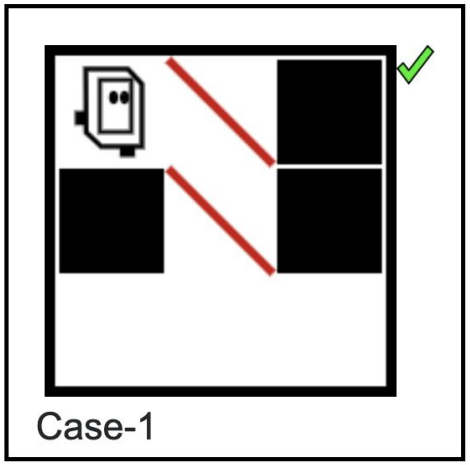
Case-1: **1 Correct** ✓ 1c



See how Bit's world starts before running more code!

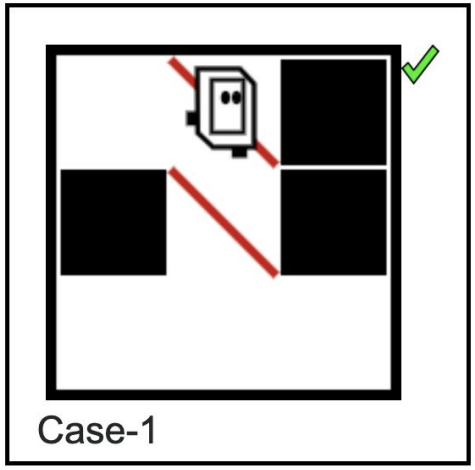
Code line by line

```
def do_bit1(filename):
    bit = Bit(filename) # Set up world
    bit.move()
    bit.paint('blue')
    bit.right()
    bit.move()
    bit.paint('green')
```



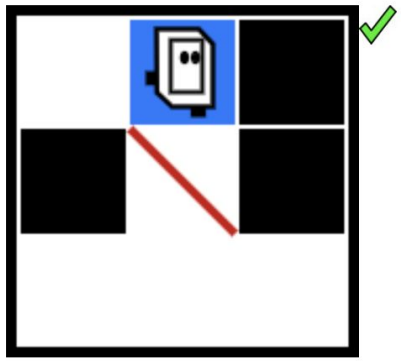
Code line by line

```
def do_bit1(filename):
    bit = Bit(filename) # Set up world
    bit.move()# Step forward one square
    bit.paint('blue')
    bit.right()
    bit.move()
    bit.paint('green')
```



Code line by line

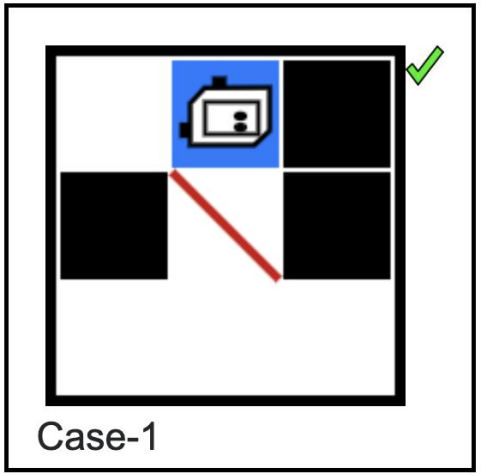
```
def do_bit1(filename):
    bit = Bit(filename) # Set up world
    bit.move()# Step forward one square
    bit.paint('blue')# Paint current square blue
    bit.right()
    bit.move()
    bit.paint('green')
```



Case-1

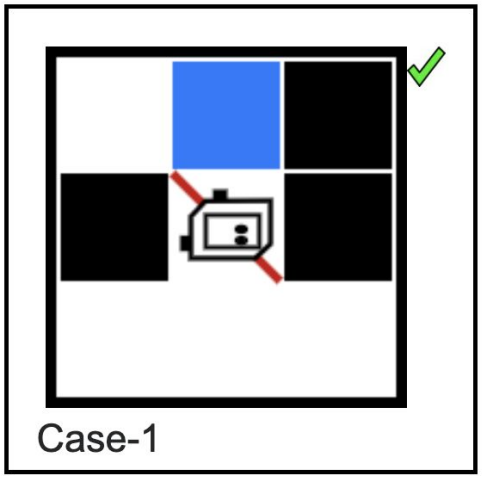
Code line by line

```
def do_bit1(filename):
    bit = Bit(filename) # Set up world
    bit.move()# Step forward one square
    bit.paint('blue')# Paint current square blue
    bit.right()# Turn 90 degrees to the right
    bit.move()
    bit.paint('green')
```



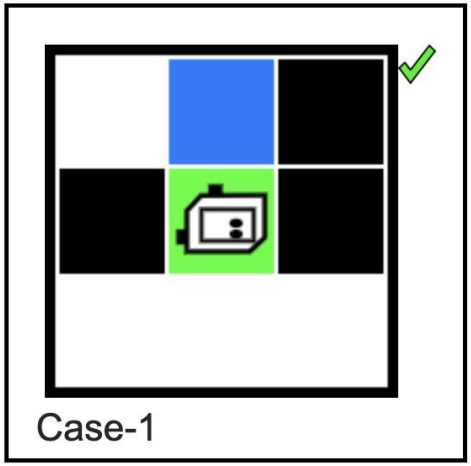
Code line by line

```
def do_bit1(filename):
    bit = Bit(filename) # Set up world
    bit.move()# Step forward one square
    bit.paint('blue')# Paint current square blue
    bit.right()# Turn 90 degrees to the right
    bit.move()# Step forward one square
    bit.paint('green')
```



Code line by line

```
def do_bit1(filename):
    bit = Bit(filename) # Set up world
    bit.move()# Step forward one square
    bit.paint('blue')# Paint current square blue
    bit.right()# Turn 90 degrees to the right
    bit.move()# Step forward one square
    bit.paint('green')# Paint current square green
```



Coding with Bit step by step

1. Read the instructions

2. Uncheck “Auto Play” and Run - see how Bit is positioned and what Bit’s world looks like

3. One line at a time, tell Bit *exactly* what to do

4. Hit Run again and see if you got it right!

5. Repeat steps 3-4 until you are happy :)

Bit knows how to do 4 things

- `move()`

  Bit will take one step in the direction Bit is facing

- `left()`

  Bit will turn 90 degrees left

- `right()`

  Bit will turn 90 degrees right

- `paint(color)`

  Bit will paint its current position `color`

  Options for color are `'red'`, `'green'`, `'blue'`

Bit knows how to do 4 things

- `move()`
  Bit will take one step in the direction Bit is facing
- `left()`
  Bit will turn 90 degrees left
- `right()`
  Bit will turn 90 degrees right
- `paint(color)`
  Bit will paint its current position `color`
  Options for color are `'red'`, `'green'`, `'blue'`

Only bit knows these instructions, so we need to say `bit.move()`, not just `move()`

do_bit2

(if time)

Coding with Bit step by step revisited

1. Read the instructions

2. Uncheck “Auto Play” and Run - see how Bit is positioned and what Bit’s world looks like

3. **One line at a time, tell Bit *exactly* what to do**

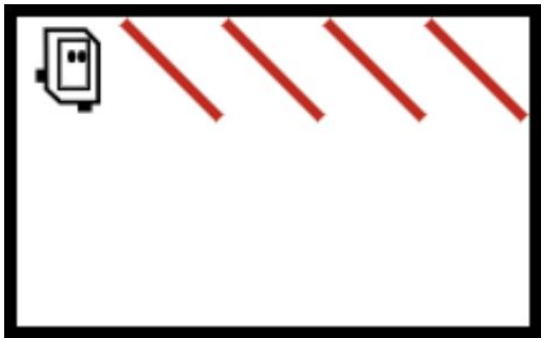
   This can be very tedious :/

4. Hit Run again and see if you got it right!

5. Repeat steps 3-4 until you are happy :)

Painting a row

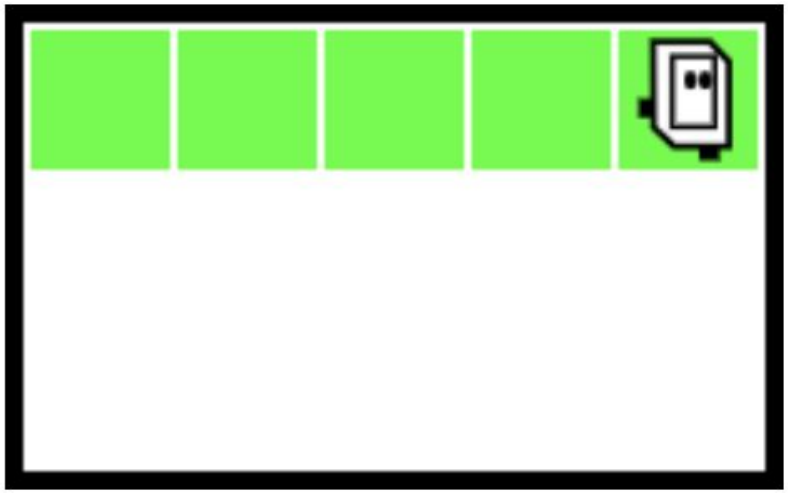
Instructions: With Bit starting in the upper left corner facing right, paint the entire first row green



One solution…

Instructions: With Bit starting in the upper left corner, paint the entire first row green

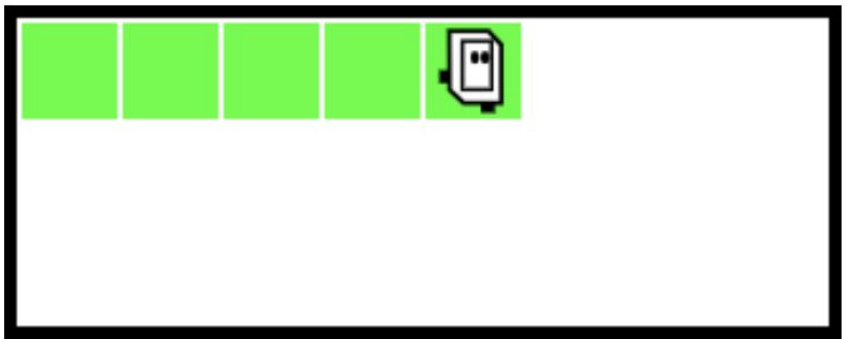
```
def do_row(filename):
    bit = Bit(filename)
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
```



Does it work on all maps?

Instructions: With Bit starting in the upper left corner, paint the entire first row green

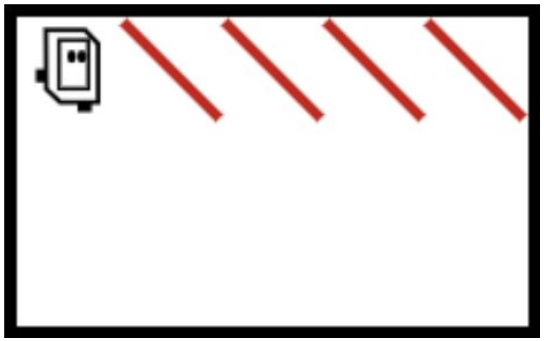
```
def do_row(filename):
    bit = Bit(filename)
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
```



How easy it is to change a just little?

Instructions: With Bit starting in the upper left corner, paint the entire first row **red**

```
def do_row(filename):
    bit = Bit(filename)
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
```



How many lines of code need to change?

Introducing the `while` loop

We can run certain lines of code over and over **while** some condition is true

```
def do_row(filename):
  bit = Bit(filename)
  while #some condition:
    # all indented lines run
    # while the condition is True
  # these lines out here
  # only run once
  # (after condition becomes False)
```

Introducing the `while` loop

We can run certain lines of code over and over **while** some condition is true

```
def do_row(filename):
    bit = Bit(filename)
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
    bit.move()
    bit.paint('green')
```

```
def do_row(filename):
  bit = Bit(filename)
  while #something:
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

Now we just need syntax to ask if Bit is blocked

```
def do_row(filename):
  bit = Bit(filename)
  while #Bit is not blocked:
    bit.paint('green')
    bit.move()
```

Questions Bit can ask

- `front_clear()`

  returns True or False if the square Bit is facing is blocked

- `left_clear()`

  returns True or False if the square to Bit's left is blocked

- `right_clear()`

  returns True or False if the square to Bit's right is blocked

- `get_color()`

  returns `'green'`, `'red'`, `'blue'` or `None` for the color of the square Bit is currently on

Introducing the `while` loop

Let's run this and see if it worked!

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

1st loop

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():# returns True!
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

1st loop iteration

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

1st loop iteration

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

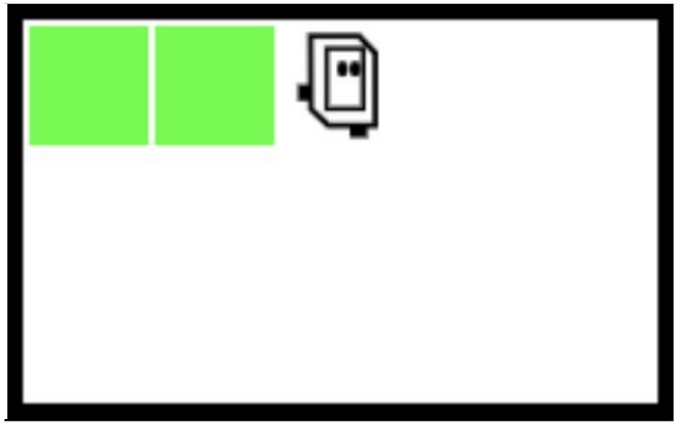
2nd loop iteration!

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

End of 2nd loop iteration!

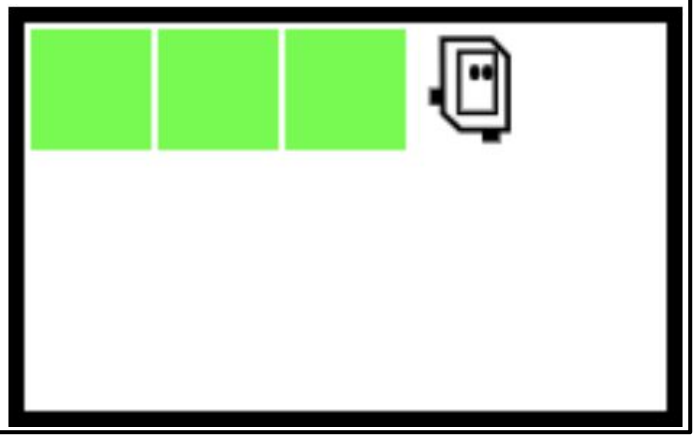
```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
```



Introducing the `while` loop

End of 3rd loop iteration!

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
```



Introducing the `while` loop

End of 4th loop iteration!

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

Beginning of 5th loop iteration?

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

Beginning of 5th loop iteration?

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear(): #returns False
    bit.paint('green')
    bit.move()
```

Introducing the `while` loop

Beginning of 5th loop?

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear(): #returns False
    bit.paint('green')
    bit.move()
  # skip loop code, done
```

Introducing the `while` loop

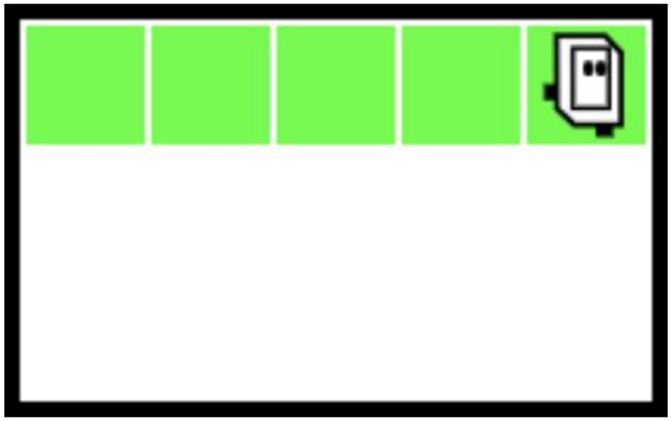
Didn't paint last square :(How to fix the bug?

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear(): #returns False
    bit.paint('green')
    bit.move()
  # skip loop code, done
```

Introducing the `while` loop

Didn't paint last square :(How to fix the bug?

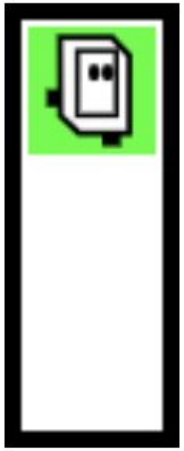
```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
  # paint last square
  bit.paint('green')
```

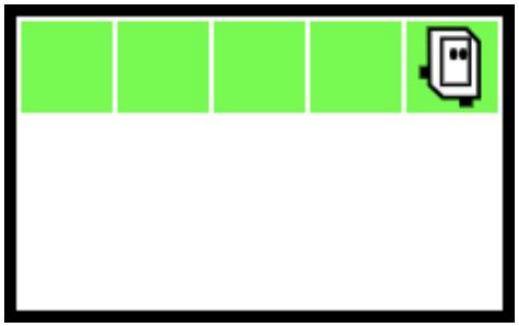


Does it work on all maps?

Instructions: With Bit starting in the upper left corner, paint the entire first row green

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
  # paint last square
  bit.paint('green')
```





How easy it is to change a just little?

Instructions: With Bit starting in the upper left corner, paint the entire first row **red**

```
def do_row(filename):
  bit = Bit(filename)
  while bit.front_clear():
    bit.paint('green')
    bit.move()
  # paint last square
  bit.paint('green')
```

How many lines of code need to change?

Recap: `While` Loop

```
while #condition:
    # code that loops
# code that doesn't loop
```

1. First the condition is checked (it should be `True` or `False`, more on that later)
2. If the condition is `True`, the "code that loops" runs, then back to step 1.
3. If the condition is `False`, the "code doesn't loop" runs, and the looping process is over

Conditions

- Conditions are statements that evaluate to either `True` or `False`

Conditions

- Conditions are statements that evaluate to either `True` or `False`

```
while #condition:
    # code that loops

while bit.front_clear():
    #if bit.front_clear returns True, loop

while True:
    # Always loops forever!
    # maybe a bad idea… don't do this :)
```

Conditions

- Conditions are statements that evaluate to either `True` or `False`
- The statement `left == right` is `True` if left is equal to right and `False` otherwise

```
while bit.get_color() == 'green':
    # loop if bit is standing on a green


while bit.front_clear() == False:
    #if bit.front_clear returns True, loop
    #Redundant! Don't need '== True'
```

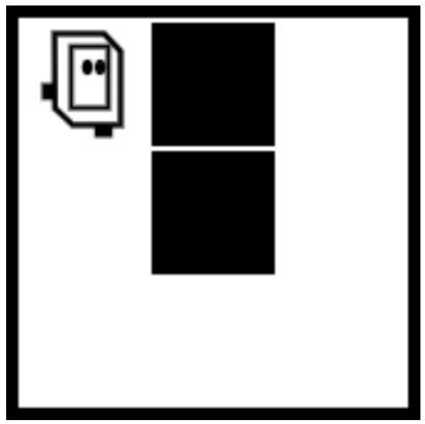
Conditions

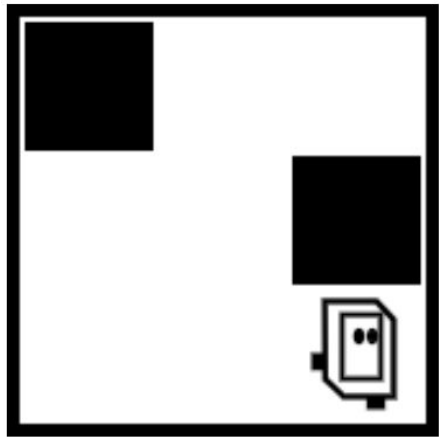
- Conditions are statements that evaluate to either `True` or `False`
- The statement `left == right` is `True` if left is equal to right and `False` otherwise
- Adding the word `not` in front of a conditions changes it from `False` to `True` or from `True` to `False`

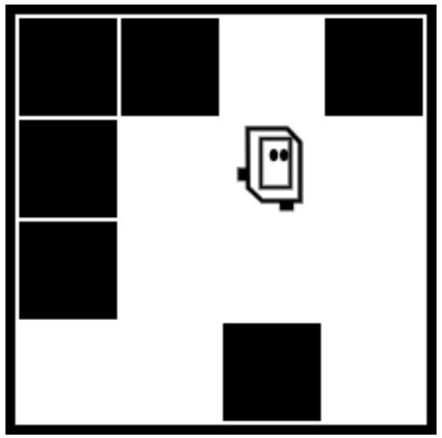
```
while not bit.get_color() == 'green':
    # loop if bit is standing on anything
    # except a green

while not bit.front_clear():
    #loop if bit.front_clear returns False
    #AKA loop while bit is blocked!
```

Unblock Bit

- Assume there is at least one direction that Bit is not blocked (and Bit may be facing it!)
- Write the function `unblock(filename)`, in which Bit finds a direction she is not blocked and takes one step in that direction

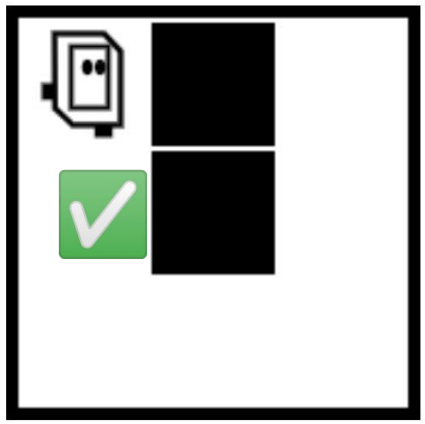


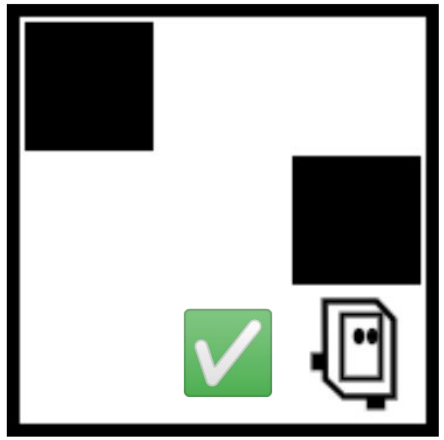


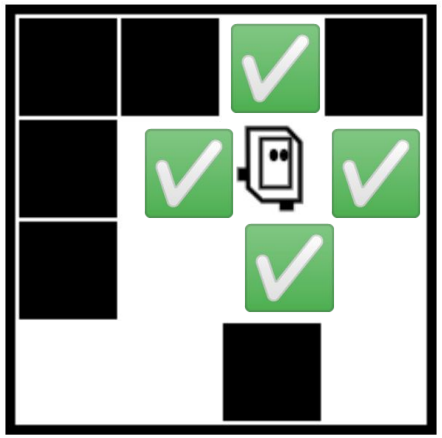


Unblock Bit

- Assume there is at least one direction that Bit is not blocked (and Bit may be facing it!)
- Write the function `unblock(filename)`, in which Bit finds a direction she is not blocked and takes one step in that direction







Unblock Bit

- Assume there is at least one direction that Bit is not blocked (and Bit may be facing it!)
- Write the function `unblock(filename)`, in which Bit finds a direction she is not blocked and takes one step in that direction

```
def unblock(filename):
    bit = Bit(filename)
```

Unblock Bit

- Assume there is at least one direction that Bit is not blocked (and Bit may be facing it!)
- Write the function `unblock(filename)`, in which Bit finds a direction she is not blocked and takes one step in that direction

```
def unblock(filename):
    bit = Bit(filename)
    while not bit.front_clear():
```

Unblock Bit

- Assume there is at least one direction that Bit is not blocked (and Bit may be facing it!)
- Write the function `unblock(filename)`, in which Bit finds a direction she is not blocked and takes one step in that direction

```
def unblock(filename):
    bit = Bit(filename)
    while not bit.front_clear():
        bit.right()
```

Unblock Bit

- Assume there is at least one direction that Bit is not blocked (and Bit may be facing it!)
- Write the function `unblock(filename)`, in which Bit finds a direction she is not blocked and takes one step in that direction

```
def unblock(filename):
    bit = Bit(filename)
    while not bit.front_clear():
        bit.right()
    # bit.front_clear must have been True!
    bit.move()
```

Syntax Errors

- AKA typos!
- Humans can usually understand code that has typos, but computers are not as smart as humans :)
- Python catches syntax errors either:
  - **before runtime**: before any code runs - also known as a compile error. Ex: biterr1
  - **during runtime**: while the code is running - also known as a runtime error. Ex: biterr4

Fixing Syntax Errors Step by Step

1. Run your code!
2. Oh no! My code didn't run all the way through :(
3. Read the error message, **pay attention to the line number**
4. Go to that line number, fix the issue
5. Try step 1 again

Syntax Errors

- AKA typos!
- Humans can usually understand code that has typos, but computers are not as smart as humans :)
- Python catches syntax errors either:
  - **before runtime**: before any code runs - also known as a compile error. Ex: biterr1
  - **during runtime**: while the code is running - also known as a runtime error. Ex: biterr4

Syntax errors are not the same as bugs! Bugs happen when your code *can* run all the way through, but it doesn't do what you wanted it to

Recap

- Welcome to 106A!
- Remember to fill out that section signup form ASAP
- Bit is a robot that can `move`, turn `left` and `right`, and `paint`
- We can make certain lines of code run over and over with `while` loops
- `while` loops require a condition that tells them if it's okay to keep looping
- Conditions are `True`/`False` statements
- Syntax errors happen! Luckily, Python tells us how to fix them

See you next time!