CS106A - Lecture 1 - Code and Bit

Today: Bit, code, functions, and the while-loop

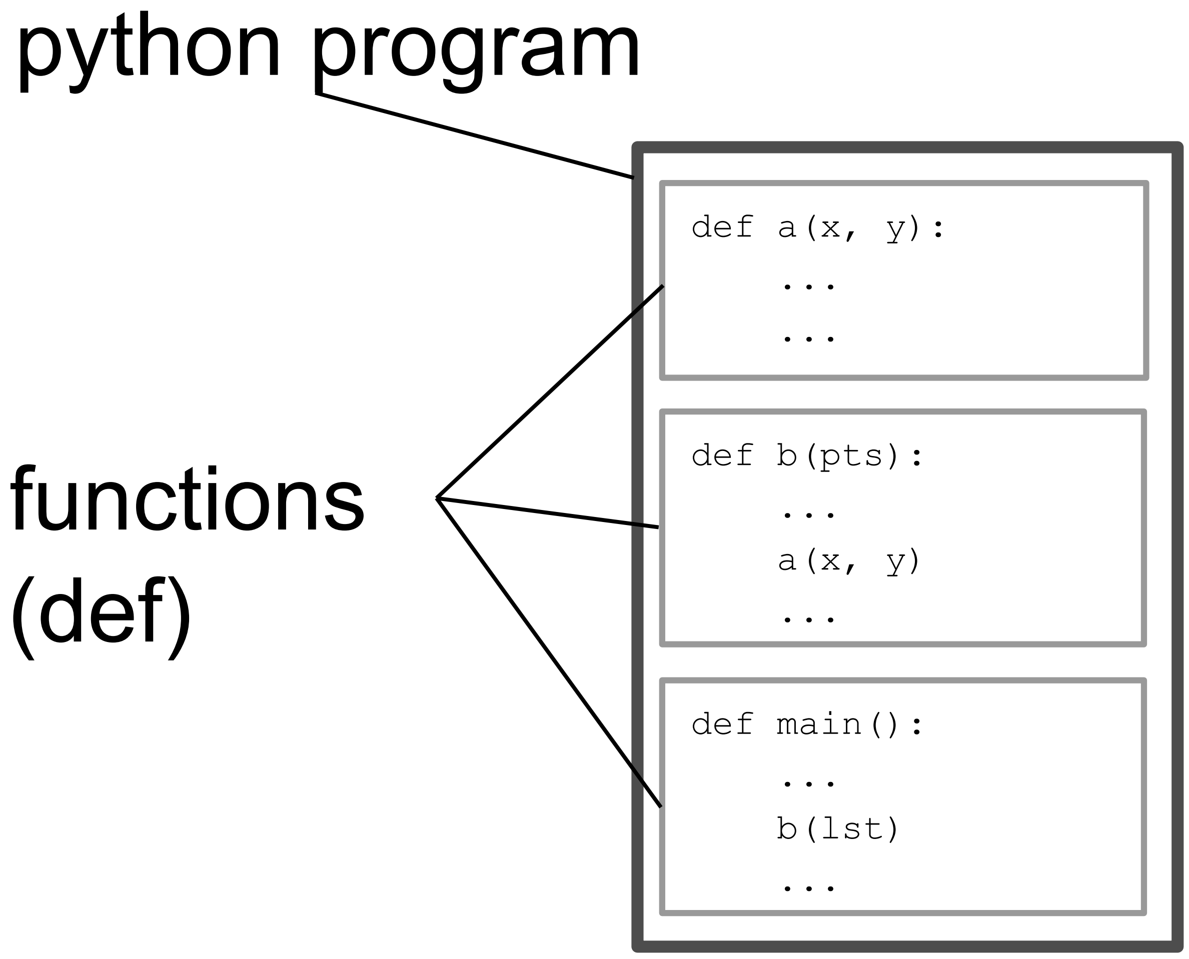
Heads Up - 10 Little Things

- We are moving fast right here at the start
- I need to show 10 little things for anything to work
- They are numerous
- They are weird looking
- But they are not that difficult
- So just hang in there for a couple lectures

Program Made of Functions

A computer **program** (or "app") is a collection of code for the computer to run. The code in a program is divided into smaller, logical units of code called **functions**, much as all the words that make up an essay are divided into logical paragraphs.

Python programs are written as text in files named like "example.py". The program text will be divided into many functions, each marked by the word **`def`**:

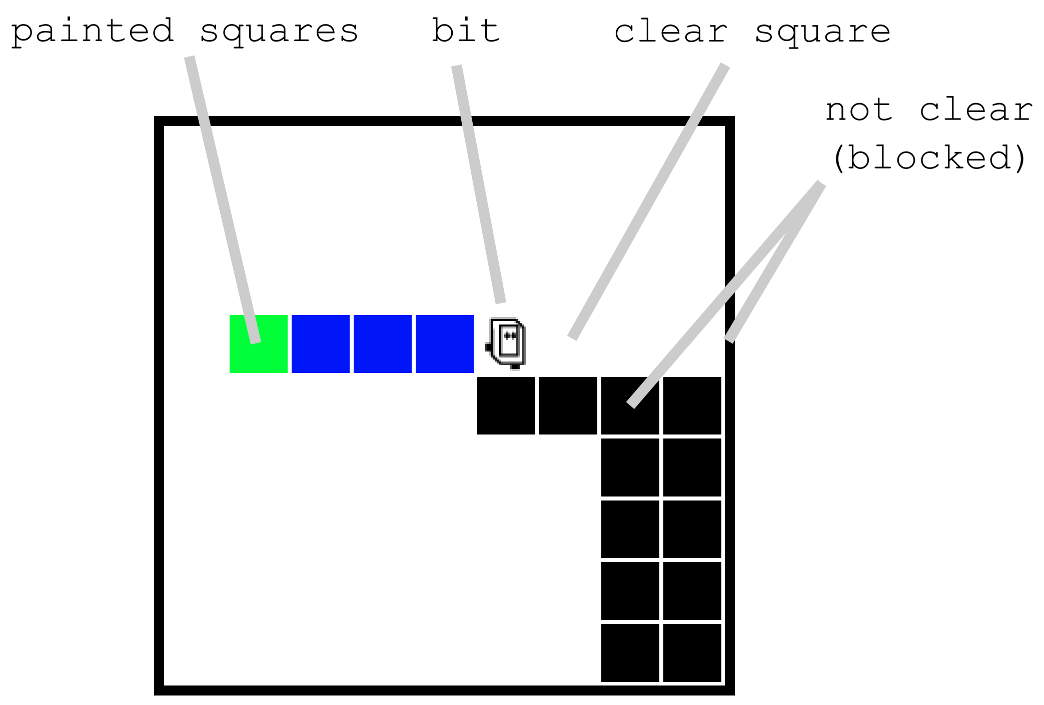


Bit Robot - Getting Started

We have this "Bit" robot, lives in a rectangular world divided into squares, like a chessboard. Bit is my research project - open it up to the world soon. We'll use bit at the start of CS106A to introduce and play with key parts of Python. The code driving Bit around is 100% Python code. The nice thing about Bit code is that the actions have immediate, visual results, so you can see what the code is doing.

Here is bit in two sentences: bit moves from square to square, can paint the square it is on red, green, or blue. Black squares and the sides of the world are blocked -

bit cannot move there.



For reference, here is a more detailed guide to Bit's features: Bit Reference

Bit can do 5 actions in the world: `move, left, right, paint, erase`

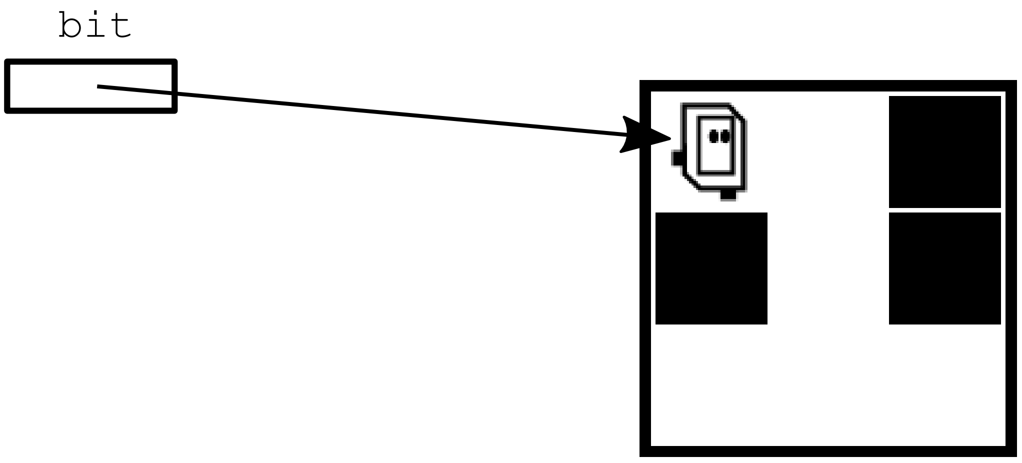
For now we'll just jump in with an example.

Example do_bit1() Function

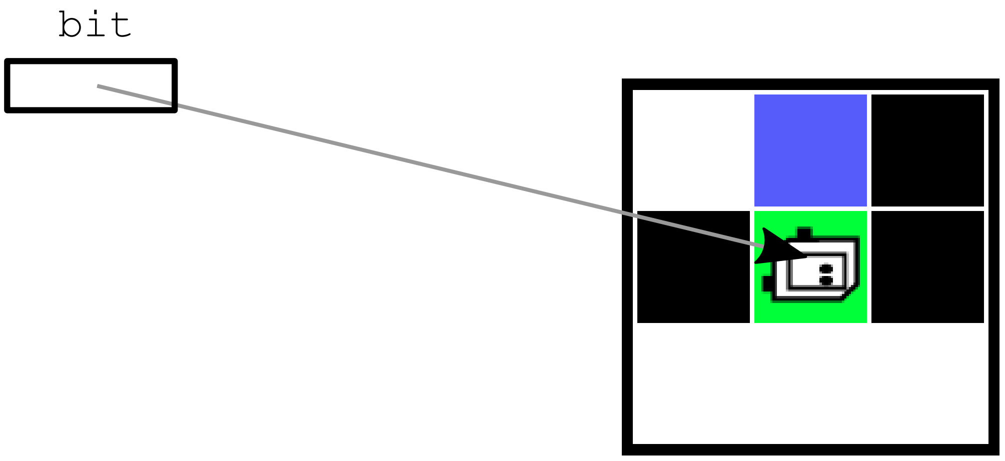
Below is a code in a Python function named "do_bit1" that creates bit in a little world and commands bit to do a few things. There is **syntax** here - not pretty. Don't worry about all the syntax in there just yet. First let's see what it does.

```
def do_bit1(filename):
    bit = Bit(filename)   # Creates "before"
picture
    bit.move()
    bit.paint('blue')
    bit.right()
    bit.move()
    bit.paint('green')
```

1. The function starts bit off facing the right side of the world with a couple empty squares in front (before-run picture):



2. The code in the function moves bit over a couple squares, painting some of them. So running the function does those actions, and afterwards the world looks like this (after-run picture):



Run It First

- Before looking at the code details
- Run the function to see what it does

- What does "run" mean?
- Run a Function = run its lines top to bottom
  Also known as "calling" the function
- Each line does something
- In this case, each line in the function prompts a little action by bit

Experimental Server Link

Here is the bit1 problem on the experimental server (you can run it too, or just watch):

> do_bit1() example

We'll use Nick's experimental server for code practice. See the "Bit1" section at the very top; the "bit1" problem is there. Often the code in a lecture example like this will be incomplete, and we'll work it out in lecture. There is a Show Solution button on each problem, so you can always play with it again later and check your answer.

Run It

Click the Run button and see what it does. See how bit takes actions in the world first, and then we'll fill in what's going on with the lines of code. You can use the "steps" slider to run the code forwards and backwards, seeing the details of the run.

Now let's look at all of that syntax to see how this works.

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

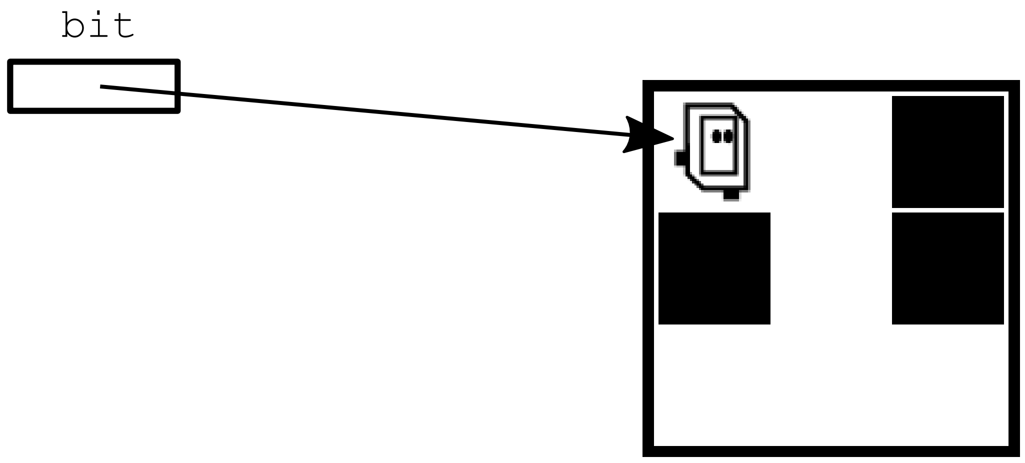
Click the Run button a few times to run this function, then we'll discuss what each line of code does.

1. Run a Function = Run Its Lines

The **Run** button in the web page runs this `do_bit1` function, running its lines of code from top to bottom. Running a function is also known as "calling" that function. In this case, each line in the function commands bit to do this or that, so we see bit move around as the function runs from top to bottom.

2. `bit = Bit(filename)` - Set Up World

This line creates the world with bit in it, and sets the variable `bit` to point to the robot. We needs this line once at the top to set things up, then the later lines use the robot. We'll explain this with less hand-waving later. This drawing shows the setup - the bit robot is created along with its world, and the variable `bit` is set to point to the robot.



3. `bit.move()` = Call the "move" Function

- There is a function named "move" that Bit understands
  This function moves bit forward one square
- The syntax `bit.move()` calls the "move" function

- Calling a function runs its lines
- This is the "object oriented" or "noun.verb" style of function call
- Note the parenthesis at the right side of the call `bit.move()`
  The call does not work without the parenthesis - syntax!

4. `bit.paint('blue')` = Call the "paint" function

- There is a function named "paint" that Bit understands
  It colors the square bit is on
- The paint function takes a "parameter" - extra information to use
- The parameter value we want to use is written within the parenthesis
- e.g. `bit.paint('blue')`
  This means: run the paint() function, passing in `'blue'` as the parameter value
- e.g. `bit.paint('green')`
  Paints the square green instead
- The paint() function works with the three parameter values: `'red'` `'green'` `'blue'`

5. Other functions: `bit.left()` and `bit.right()`

- There are also functions named "left" and "right"
- These turn bit left and right 90 degrees in the world
- The syntax to call these functions is the same, e.g. `bit.right()`

6. `def` = Define a Function

Now we'll go back and talk about the first line. This defines a function named `do_bit1` that holds the lines of code we are running.

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

- 1. The definition of a function starts with the word: `def`
- 2. The `def` is followed by a name for the function, e.g.: `do_bit1`
  The name is chosen by the programmer
  We'll have better examples later, we promise
  The name is often verb-like, referring to what the function does
  One or more lowercase words, separated by underbars
- 3. After the name is a parenthesis pair, not explained today, then a colon
- 4. All the indented lines below the `def` are the lines which make up this function
  In this case the lines move bit around
- Summary: a function = a name + a bunch of lines of code

A Little Talk About Syntax Errors

"Syntax" is not a word that regular people use very often, but it is central in computer code. The syntax of code takes some getting used to, but is not actually difficult.

- Syntax - code is structured for the computer
  Remember the computer is kind of dumb!
- Very common error - type in code, with slight syntax problem
- e.g. syntax error: `bit.move[)`

- Professional programmers make that sort of "error" all the time
- Just type it in, run it, fix the errors
- Not a reflection of some flaw in the programmer
- Do not give in to the following thoughts (imposter syndrome):
  "Another syntax error .. maybe I'm not cut out for this!"
  "I bet nobody else is getting these"
- Just the nature of typing ideas into the mechanical computer language
- Fixing these little errors is a small, normal step
- The computer is a powerful tool in your hands
  But you need to talk to them their way on syntax

"Bit Errors" Exercise - Sport and Fix!

- Exercises to desensitize: a bunch of typical errors, see them and fix them
- Skill: **Read the error message**
  Though cryptic, often an excellent clue
- These all have errors, trying run and fix (lecture demo, or on your own)
- Would a human understand what these mean, even with the errors?
- These are on the experimental server, uses your Stanford login
- **Compile Error** - syntax problem, code does not run
  e.g. mismatched parenthesis or quotes, un-even indentation
- **Runtime Error** - code runs, then hits an error on some line
  The program halts on the error line
  Bit gets a funny "error" face for a runtime error
- Lecture, start with 1, 3, 6
- biterr1 (syntax error)
- biterr2
- biterr3

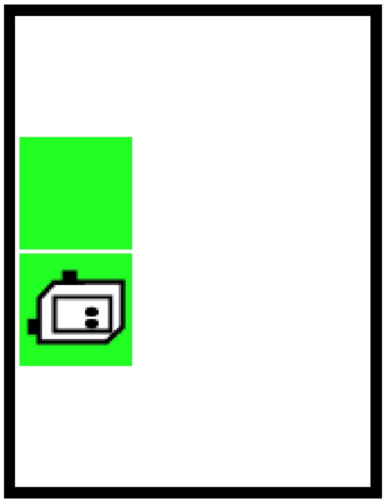
- biterr4 (runtime error)
- biterr5
- biterr6 (bad move)
- biterr7
- biterr8

(optional) You Try One: bit2

> do_bit2() example

Here is an exercise like bit1 with the code left for you to do. We probably won't have time for this today, but you can do it on your own.

Bit begins at the upper left square facing right. Paint the square below the start square green, and the square below that green as well, ending like this:



The word **pass** is a placeholder for your code, remove it. Conceptually this does not look hard, but expect to make some mistakes as you get the little syntax and naming details right as required by the (lame, needy) computer

---

Teaching Aside: Words vs. Just Run It

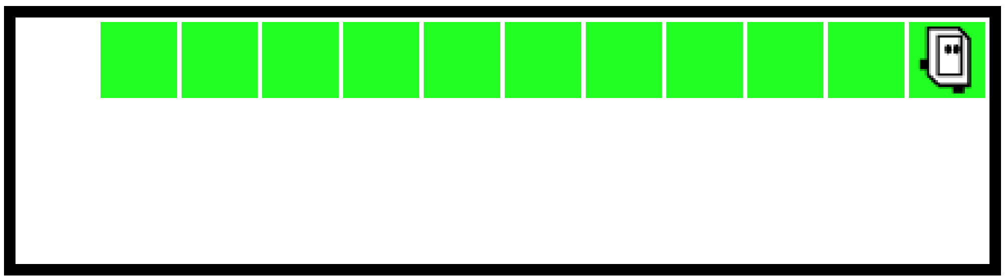
- Could have words, explaining how a thing works
- But that only goes so far
- Could put up an example of the thing up and just run it
- SEE how it works
- We'll use both of these strategies

Aside: Checkmark / Diff Features

- The system knows what the world is supposed to look like when the code works correctly
  If the output is correct at the end of the run, it gets a green checkmark
- "diff" Feature - diagonal red marks on incorrect squares
  The system knows what the world is supposed to look like, marking squares that are the wrong color with a red diagonal

Suppose I Give You This go-green Problem

Bit starts at the upper left square as usual. Move bit forward until blocked by a wall, painting every moved-to square green. The resulting world looks like:



- This looks like a lot of work + tedious

- To make all that green, run these two lines many times
  ```
  bit.move()
  bit.paint('green')
  ```
- Need to stop when reaching far wall
- Q: How to do this?
- A: a While Loop

Run go-green code

First run the code a few times. This code is complete. Watch it run a few times to see what the loop does, and we'll look at the details below.

> go-green

While Loop

The loop is a big increase in power, running lines many times. There are a few different types of loop. Here we will look at the **while** loop.

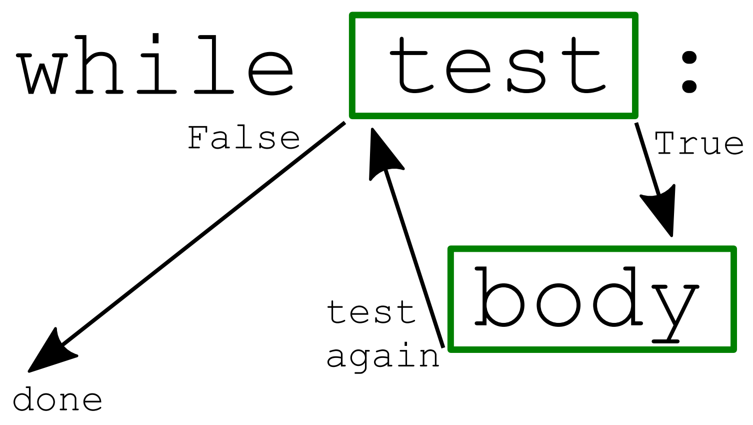
While Loop Syntax

The while loop syntax is made of four parts: the word **while**, a **test** expression, a colon (`:`), and on the next line indented **body** lines to run.

```
while test-expression:
    body lines
```

While Loop Sequence

This diagram shows the sequence between the while loop test and the body:



How the While Loop Works

- First the loop evaluates the test-expression at the top - does it return `True` or `False`?
  `True` and `False` are the "boolean" values in Python, we'll see more later
- If `True`, the loop runs all the lines of the body, from top to bottom
  After running the body, the loop always comes back to the top to check test again
- Eventually when the test returns `False`, the loop is done, and the run continues after the loop body

There is not much to say about the body; it is just a series of lines to run. The test expression is more interesting, controlling the run of the loop.

Run go-green

For today, it's maybe sufficient to run go-green a few times, get a feel for how it runs the body lines many times. We'll study it in more detail in lecture-2.

Optional: Go-Left, Go-Right Exercises

We'll continue with while loops next lecture. If you'd like to play around with some code which is similar to go green try writing the code for go-left first. The word "pass" is a placeholder in Python that does nothing. For exercises, replace the word "pass" with your code.

> go-left

The go-right exercise requires two loops - write one loop to go to the right corner. Write a second loop to go down. The first loop and the second loop should be at the same indentation, not one inside the other. Note that all these problems have a Show Solution button, so you can play with them without writing the code.

> go-right