Today: insights about loops, if statement, ==, make-a-drawing, left_clear(), bit puzzles

Aside: Bit Controls

- Cmd-Return (Mac) or Ctrl-Return with cursor in code - just Run
  Very handy when pounding away on your code
- Diff marks - red marks on world vs. desired output
- Speed
- Auto Play - uncheck, Run button shows bit at start state, can play from there
- End State - check, instead of animating, just show end state

---

Words and The Nature of Computing

"The computer does not have any insight, instead following the simple, mechanical instructions written by the programmer."

These are just words, and rather than try to explain them in different ways, I will hope that after you've written 10 Bit programs, you will understand in your bones what those words are trying to say.

Recall: Go-Green

> go-green

We had this example at the end lecture-1. Now we'll look at the details

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.

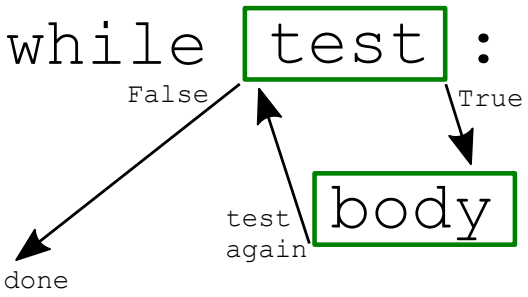
While Loop Syntax

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

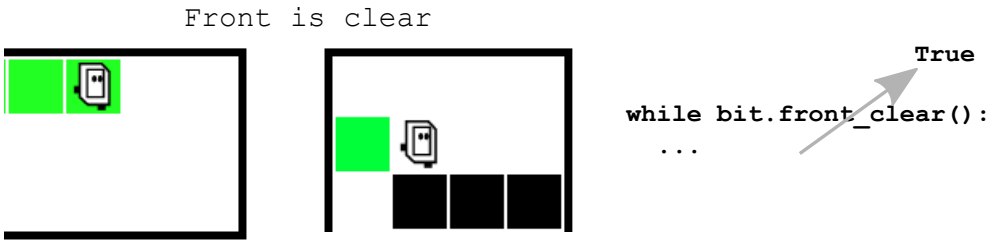
There is not much to say about the body; it is just a series of lines to run. The test expression is more interesting, controlling the run of the loop.

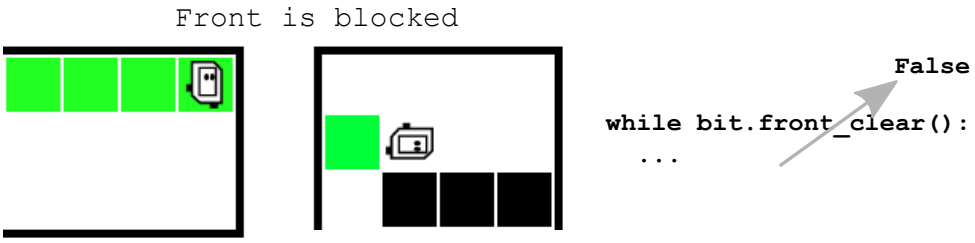
CS Concept - "Expression"

An expression is a piece of code that runs as usual and also returns a value to that spot in the code. The `front_clear()` function, acts as an expression here. When called, the `front_clear()` function checks the square in front of bit, returning `True` if the square is clear for a move, or `False` otherwise. The values `True` and `False` are the special "boolean" values in Python, which we'll see in more detail later.

`bit.front_clear()` Visualization

A nice way to visualize the action of an expression is drawing a diagonal arrow through the expression code, showing the returned value coming out of the expression when it runs. Here is a drawing showing the boolean value coming out of the `bit.front_clear()` function call with bit not blocked and then blocked:





In Bit's world, the way forward is clear if the square in front of Bit is not blocked by the edge of the world or a black square (we'll see black squares shortly).

`bit.front_clear()` - In The While Loop

Here again is the go-green loop:

```
while bit.front_clear():
    bit.move()
    bit.paint('green')
```

The while loop first checks the text expression `bit.front_clear()`, checking the square ahead and returning `True` if it is clear. If the test returns `True`, the loop goes in and runs the body from top to bottom. In this case, that moves bit forward one square. After the body, the loop always goes back to the top to check the test again. In this way, the loop runs forward through all the empty squares, until bit is on a square where the front is not clear, and the loop exits.

Observation: Test = Go

When the test is `True`, bit keeps going. We think of this as Test=Go. Later when we're writing code to advance bit through some other situation, we'll use Test = Go to think about the test.

Run go-green - Questions Below

> go-green

Run the go-green function, and watch it to see how the while loop works. In particular, use the "step" button to advance one line at a time as bit is on the next-to-last square and then the last square, seeing the test return `True` for all squares but the last.

Q1 - What Comes After Line 5?

Position bit a few squares from the end with the steps slider. Step to line 5 and stop. What line comes after line 5?

Q2 - What Comes After Line 3?

Step to line 3. What line comes next? It depends on where bit is actually. Run to the end to see both cases.

Optional: Go-Left, Go-Right Exercises

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

Move-To Convention, move() then paint()

- The go-green loop looks like this:
```
bit.move()
bit.paint('green')
```
- This works fine, painting all the squares except missing the first square
- The loop operates on each "moved to" square

move() then paint() in loop:

Equally Valid Convention: paint() then move()

- Could change the loop body to paint() then move():
```
bit.paint('green')
bit.move()
```
- This works fine too!

- It paints all the squares except missing the last square
- Demo: try it, turn off "diff" option
- Both conventions work, but we tend to design problems for the move-to convention

paint() then move() in loop:

Loop move + paint Conclusions

- 1. Loop with move + paint always misses the first or last square
- 2. Don't drive yourself crazy trying to use one loop to hit all the squares
- 3. To hit all the squares, you need to add extra code before or after the loop
  We'll do this on the next problem

All Blue Example

Next example - use this to play with more features of loops. This code is complete.

> All-Blue

Often we have an English language description of what we want, and then we're thinking about the code to get that effect.

Description: Bit starts in the upper left square facing the right side of the world. Move bit forward until blocked. Paint every square bit occupies blue, including the first. When finished, turn bit to face downwards. (Demo - this code is complete. Shows the basic while-loop with front_clear() to move until blocked, and also taking an action before and after the loop.)

all-blue() Code

```
def all_blue(filename):
    bit = Bit(filename)
    bit.paint('blue')  # 1 paint before the loop
    while bit.front_clear():
        bit.move()
        bit.paint('blue')
    bit.right()  # Turn after the loop
```

1. Lines Before/After Loop Different

- Lines before and after the loop to handle the first/last squares
  `bit.paint('blue')` - before loop
  `bit.right()` - after loop
- These lines are outside the loop, run once
- Point: indentation of a line, inside vs. outside the loop, makes a big difference
- **Indentation** controls what's in the loop
- Demo: what if want to paint the last square red?
  Change the last line to `bit.paint('red')`
  Try indenting it wrong

2. One Code - Many Cases - Generality

- "Generality" - a real Math Department word
- There is one copy of your code algorithm
- There are many different cases it should work on
  Infinitely many cases actually
  Think of all the different sizes and shapes of Bit world
  The `while` construct is very powerful for generality
- General = it should work for all the different input worlds
- Click the run menu to select different "cases", run against them
- Develop your code using just one case for testing
- Then use the Run All option as a final test
- Run All gives the **Big Green Checkmark**
  For grading, need do Run All
  Final test for each HW problem
  Don't debug with Run All - output is worse, debug with just one case
- I like how generality is kind of ethereal, but with 1-code many-cases .. it's very concrete
- Note: Run-all does not *prove* the code is correct in an airtight way
  If the code works for run-all it is probably correct, but it's not a 100% proof

3. Zero Loops is OK - Edge Cases

Look at Case-3 and then Case-4. Turn off the Auto-Play option, so when you click Run, the world stays at the start, waiting for you to click Step. See how the while test behaves on Case-3 and then Case-4. For Case-4, the while-test is `False` the very first time! The result is, the loop body lines run zero times. This is fine actually. There were zero squares to move to, so looping zero times is perfect. This is an example of an "edge" case, the smallest possible input. It's nice that the while loop handles this edge case without any extra code.

4. Bad Move Bug - Exception

Let's do something wrong. Change the loop body to have three moves instead of one, which is buggy code. The problem is, this code may move bit into a wall (a

not-clear square).

```
...
while bit.front_clear():
    bit.move()
    bit.move()
    bit.move()
    bit.paint('blue')
bit.right()
```

With the correct code, the test checks the one square ahead, and the loop body moves one time, that square (e.g. the go-green loop). In this way, each move is preceded by a test and is 100% not going to fail. With three moves - for the second and third moves, we don't know that those squares are clear for a move.

Run the buggy code - bit reaches the right side, but keeps trying to `move()`. Moving into a wall or black square (i.e. not a clear square) is an error, resulting in an "exception" error message like this and the program halts at that line:

```
Exception: Bad move, front is not clear, in:all_blue line 6 (Case 1)
```

If you see an exception like that, your code is doing a `move()` when the front is not clear. You need to look at your code logic. Normally each `move()` has a `front_clear()` check before it, ensuring that the world has space for a move.

Python Guide - Debugging Chapter

See the Debugging chapter in the Python Guide for more information Python Debugging

Debug Rule 1 - Code Crashes -> Read the Error Message

First rule of debugging - if there's an error - read the error message and see what line number it is talking about. The error messages can be cryptic, and in some cases they are not much help. But very often, the error message and the line number if mentions are excellent clues about the problem in the code.

5. Infinite Loop Bugs

Infinite Loops

- Oddball bug case: infinite loop
- The body lines run forever
- The lines never manage to make the while-test `False`
- May get error message: timed out - possible infinite loop
  The Bit system notices code running for a long time, terminates it
- Click the Bit "stop" button or click on the code to stop the line hilighting
- May get "timeout" if the tab just needs to be reloaded, so try that too

The system tries to terminate the infinite loop code. Sometimes that does not work, and you have this infinite loop code running in your browser. In that case, you may need to close the experimental server tabs, or possibly restart your browser to kill off the wayward run.

Infinite Loop 1 - Waggle Dance

For a loop to work, each run of the body needs to get closer to being done. With these bugs, bit is not making progress and so it never ends.

Here is a buggy All Blue - instead of `move()` have bit do `right()` and the `left()` - a sort of "waggle dance" like bees do in the hive. Notice that bit waggles but never moves forward. Bit can do this for eternity, and never leave that first square! Everyone has days like that.

```
while bit.front_clear():
    bit.right()    # waggle dance!
    bit.left()
    bit.paint('blue')
```

Run the code - you will see a message about a timeout, suggesting that this code does not seem to reach an end.

Infinite Loop 2 - `move vs. move()`

Python syntax requires the parenthesis `()` to make a function call. Unfortunately, if you omit them, easy enough to do, it is syntactically valid, but it does not call the function. So the code below **looks** right, but actually it's an infinite loop. Bit never moves. Try it yourself and see, then fix it.

```
while bit.front_clear():
    bit.move
    bit.paint('blue')
```

Optional: More Practice

For extra practice, here are some more problems along the lines of All-Blue:

> red2

> go-back

> Bit Loop Section - problems using 1 or more loops

If Statement - Logic

The while-loop is power. If-statement is **control**, controlling if lines are run or not

If-Statement Demo - Change Blue

We'll run this simple bit of code first, so you can see what it does. Then we'll look at the bits of code that go into it.

Problem statement: Move bit forward until blocked. For each moved-to square, if the square is blue, change it to green. (Demo: this code is complete.).

> Change-Blue Demo

For reference here is the code:

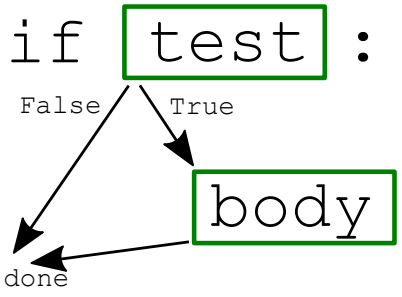
```
def change_blue(filename):
    bit = Bit(filename)
    while bit.front_clear():
        bit.move()
        if bit.get_color() == 'blue':
            bit.paint('green')
```

If Statement Syntax

The if statement syntax has four parts — the word "if", boolean test-expression, colon, indented body lines

```
if test-expression:
    body lines
```

Or structurally we can think of it like this:



If Statement Operation

- The if-statement evaluates the test expression, looking for `True` or `False`
- If the test is `True`, it runs the body top to bottom
- Otherwise it skips the body, and the run continues after the body
- This is essentially an On/Off switch - run the body lines or run nothing
- There are more complex if forms for later, for today this is the simple, common form

Look At Change-Blue - Take Apart

Here are the key lines of change-blue. We'll look at the individual parts to understand how it works.

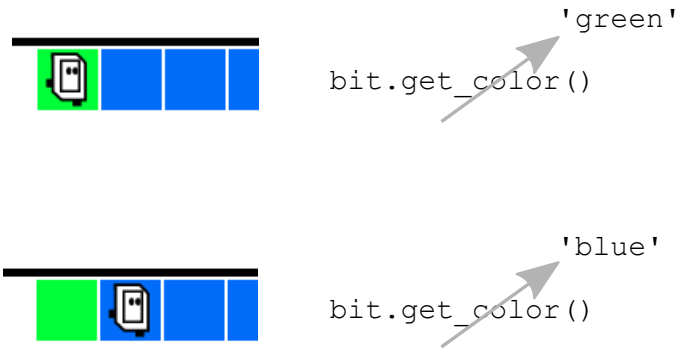
```
....
if bit.get_color() == 'blue':
    bit.paint('green')
....
```

1. bit.get_color() Expression

- Code that runs and returns a value to use is an "expression"
- e.g. `bit.front_clear()` expression - returns `True` or `False`
- e.g. `bit.get_color()` expression - today's example
- `bit.get_color()` expression returns one of these four:
  `'red', 'green', 'blue', None`
  `None` is returned if the square is not painted
- `None` is the special Python value meaning "nothing"
  It's handy in computer code to a formal "nothing" value for situations like this

Expression Visualization

Suppose bit is on a squared painted `'green'`, as shown below. Here is diagram - visualizing that `bit.get_color()` is called and it's like there's an arrow coming out of it with the returned `'green'` to be used by the calling code.



What is an "operator"

- Suppose we have an expression like `1 + 2`
- We call the `+` an "operator" - it represents the addition operation

2. == Compares Two Values - Boolean

- The operator == compares two values, checking if they are equal (written as two equal signs next to each other)
- Returns `True` if two values are equal, `False` otherwise
- Very often used in an if-test or a while-test
- `if x == 6:`
  This if-test is `True` if the variable x is 6

Look at Change-Blue Code Again

Putting this all together, we can read the change-blue code word by word to see what it does.

```
while bit.front_clear():
    bit.move()
    if bit.get_color() == 'blue':
        bit.paint('green')
```

Q: What is the translation of the code in the loop into English?

A: For each moved-to square. If the square is blue, paint it green.

Alternate != Not Equal Form

- Not-equal comparison operator written with exclamation mark: `!=`
- `if x != 6:`
  This if-test is `True` if x is not 6
- If you want to express "not-equal" in your code, use `!=`
- Examples below

More Examples == !=

```
if bit.get_color() == 'red':
    # run this if color is 'red'


if bit.get_color() == None:
    # run this if square is not painted


if bit.get_color() != 'red':
    # run this if square is anything
    # other than 'red'
    # e.g. 'green' or 'blue' or None
```

The Fix Tree Problem

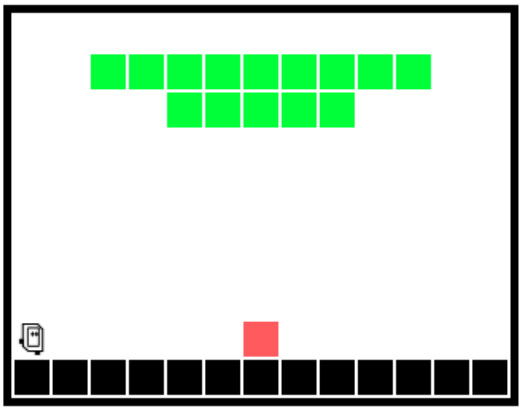
This is a challenging problem for where we are at this point. Just watch (and you can do it in parallel) as I work through the problem. I want to show the thought process and steps to build and debug some complicated code.

Note: it is very common to move bit with an until-blocked loop. This problem is a little unusual, moving bit until hitting a particular color.

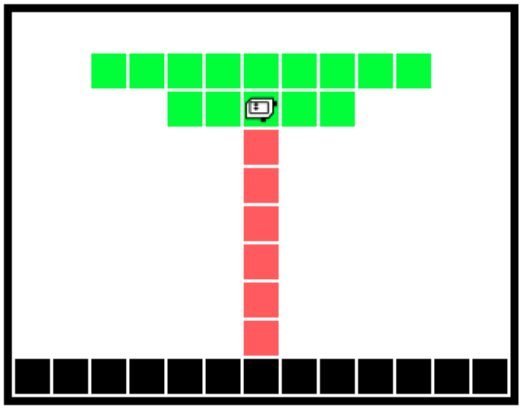
> Fix Tree

Problem: The squirrels have stolen the trunk from the tree! Bit starts facing the right side of the world. Move bit forward to find the red stump of the tree. Then Move bit up, painting the blank squares red, stopping on the first green square.

Before:



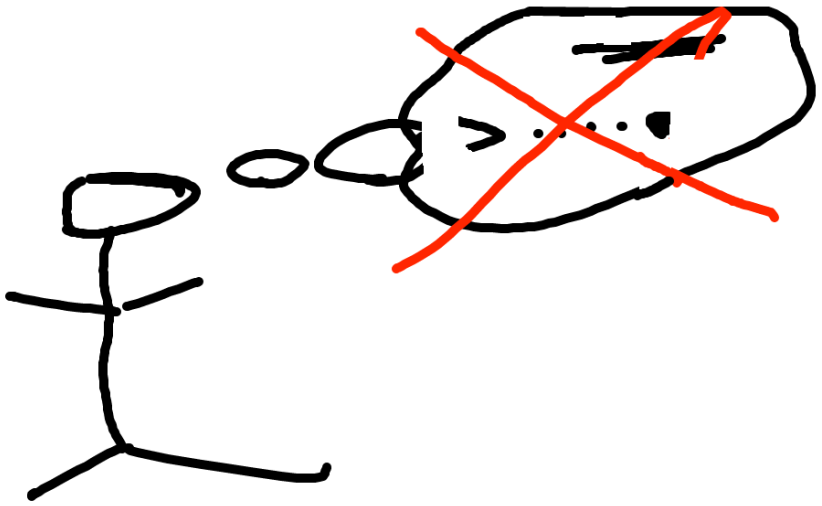
After:



Important Ideas - Thinking Drawing and Coding

We have art for these. No expense has been spared!

1. Don't do it in your head

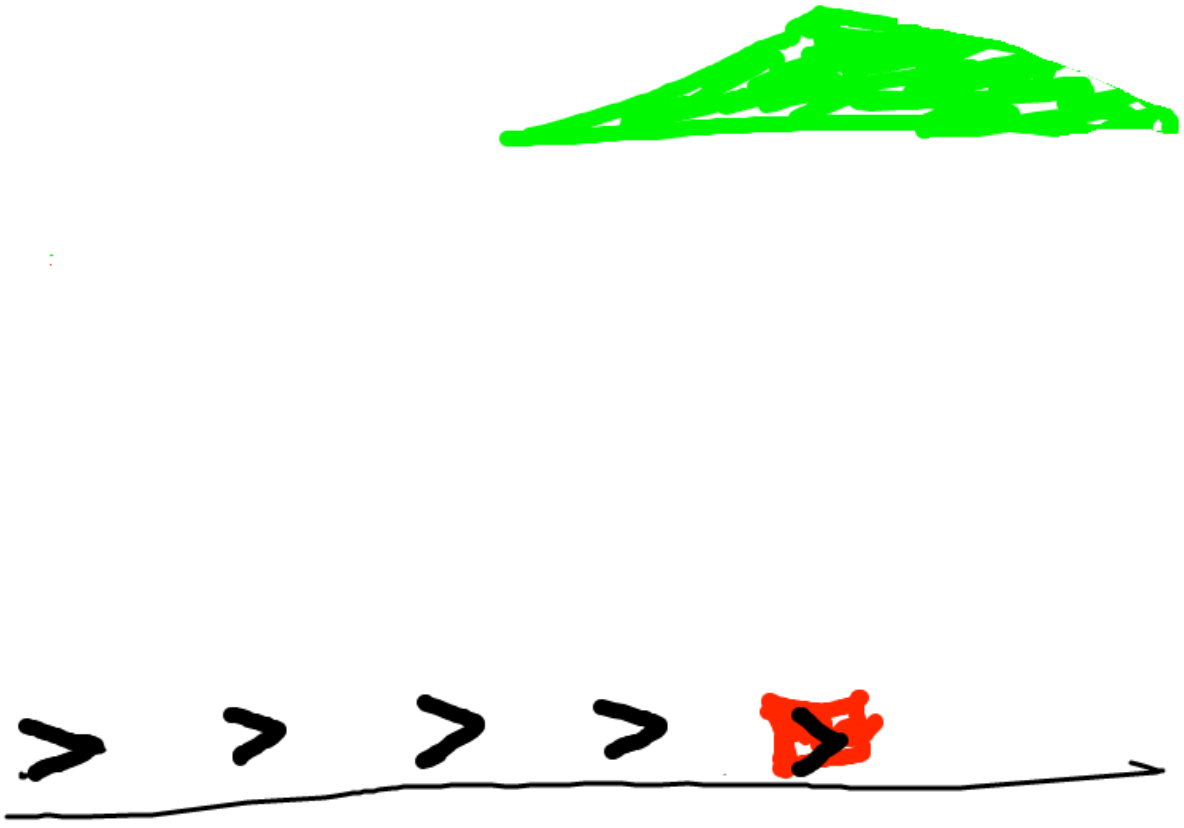


Don't do it just in your head - too much detail. Need to be able to work gradually and carefully.

2. Make a Drawing / Sketch

Draw a typical "before" state - like what do we have to start. The drawings do not need to be fancy, but it's easier to work out the details on a sketch than in your head. Key lesson for today.

3. What is a next-step goal - what do we want?



Look at the before state - what code can advance this?

4. Question: what code?

You have the current state. What **code** could advance things to the next goal?

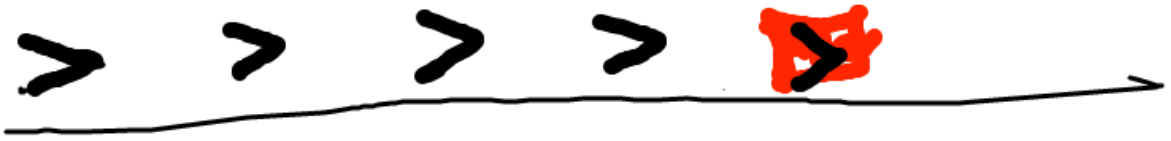
Q: Look at those bit positions. When do you want bit to move, and when not to move?

A: Want bit to move when the square is not red. What is the code for that?

Aside: if you were talking to a human, you would just say "make it look like this" and point to the goal. But not with a computer! You need to spell out the concrete steps to make the goal for the computer.

In this, case we're thinking while-loop. Draw in the various spots bit will be in. What is a square where we **do not** want bit to move? When bit is on the red square.

Otherwise bit should move. What does that look like in code?



Code looks like...

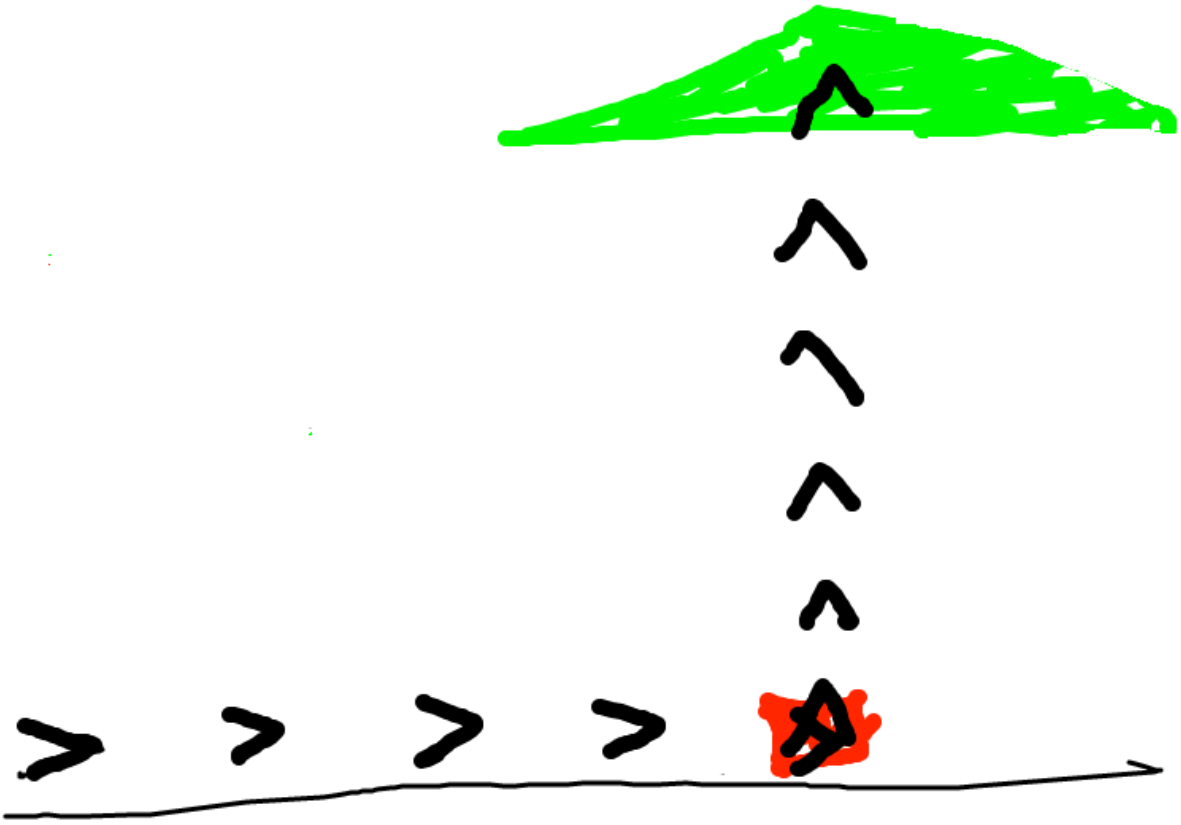
```
while bit.get_color() != 'red':
    bit.move()
```

5. Try The Code, See What It Does

Ok, run it and see. Sometimes it will not do what we thought.

6. Ok what is the next goal.

Ok what is the next goal? What would be code to advance to that?



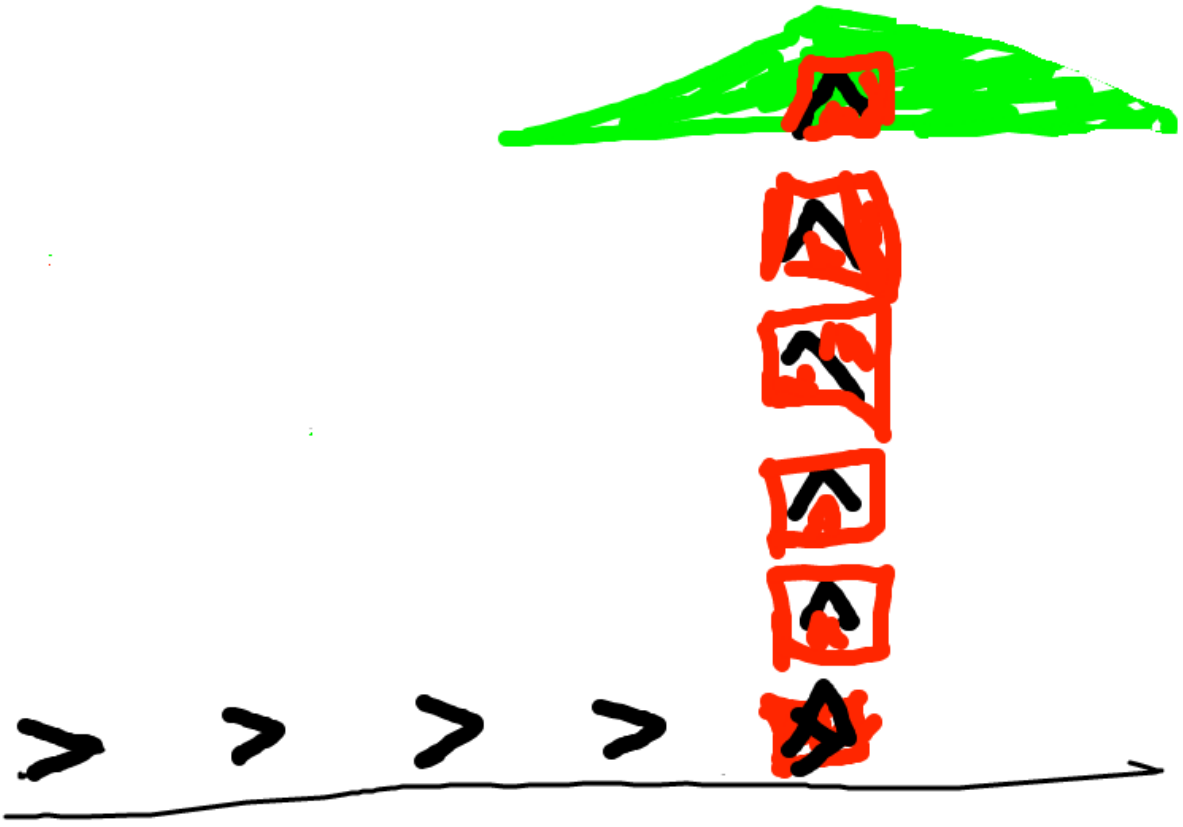
I think a reasonable guess here is similar to the first loop, but painting red, like this

```
while bit.get_color() != 'green':
    bit.move()
    bit.paint('red')
```

7. OOPS Not What Expected!

OOPS, that code looks reasonable, but does not do what we wanted. The question is: what sequence of actions did the code take to get this output? That's a better question than "why doesn't this work".

Go back to our drawing. Think about the paint/red and the while loop, how they interact.



8. Solution

The problem is that we paint the square red, and then go back to the loop test that is looking for green, which we just obliterated with red. One solution: add an if-statement, only paint **blank** squares red, otherwise leave the square alone.

This is one solution, which works perfectly..

```
def fix_tree(filename):
    bit = Bit(filename)
    while bit.get_color() != 'red':
        bit.move()
    bit.left()
    while bit.get_color() != 'green':
        bit.move()
        if bit.get_color() == None:
            bit.paint('red')
```

Other Solution

Another solution is to change the loop body to do the paint and then the move - that works perfectly too. I slightly prefer the == None solution as it seems more obvious. Both are fine. Very often there are a few different ways to solve something. If we have a strong preference for one form, we will say so, but in this case both are fine.

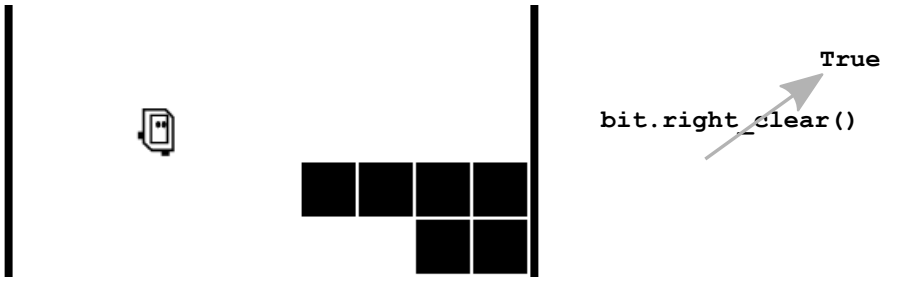
9. Thinking - Drawing - Coding

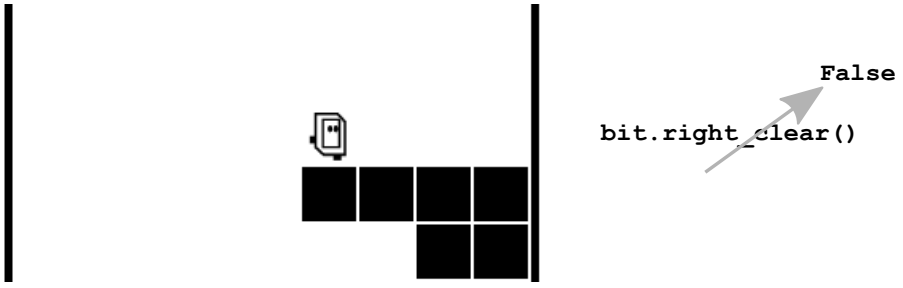
Use a drawing to think through the details of what the code is doing. It's tempting to just stare at the code and hit the Run button a lot! That doesn't work! **Why** is the code doing that?

Or put another way, in office hours, a very common question from the TA would be: can you make a little drawing of the input here, and then look at your line 6, and think about what it's going to do. All the TA does is prompt you to make a drawing and then use that to think about your code.

More Bit Tests: `bit.right_clear()`

- `bit.right_clear()` - another Bit function
- Returns `True` if the square to bit's right is clear, `False` otherwise
- Same as `bit.front_clear()` but 90 degrees to the side
- Also have `bit.left_clear()`





Using `not`

- Putting `not` to the left of a True/False value inverts it
- Use `not` with an if-test or while-test to check for the `False` condition
  Normally they look for the `True` condition, but we use `not` to invert this

```
if bit.right_clear():
    # run here if right is clear
```

```
if not bit.right_clear():
    # run here if right is blocked,
    # aka not-clear
```

Server: Bit Puzzles

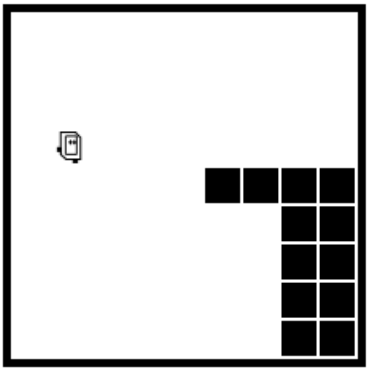
On the experimental server, the Bit Puzzles section has many problems where bit is in some situation and needs to move around using Python code. We'll look at the Coyote problems, and there are many more problems for practice.

Reverse Coyote, remember Test = Go

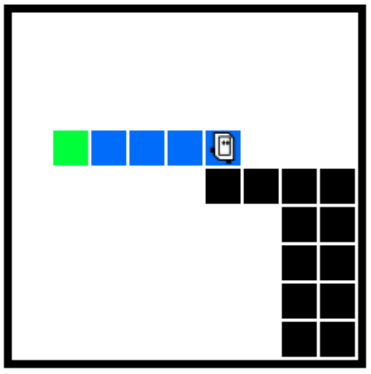
> reverse-coyote (while + right_clear)

Based on the RoadRunner cartoons, which have a real lightness to them. In the cartoons, the coyote is always running off the side of the cliff and hanging in space for a moment. For this problem, the coyote tries to run back onto the cliff before falling.

Before:



After:



Q: What is the while test for this? Remember that Test = Go. What is the test that is True when we want to move, and False when do not want to move?

A: `bit.right_clear()`

If We Have Time - You Try

These are all in the "puzzle" section. Reminder: all lecture examples have a Show Solution button if you get stuck.

> standard-coyote

> climb

> encave (a little harder, like a homework prob)