Control Flow

Loops, Conditions, Ifs, and a touch of decomp!
Housekeeping

- First sections happen(ed) today and tomorrow

- Assignment 1 (Bit) is out, will need tomorrow’s lecture

- YEAH hours this Friday 4-5pm in Gates B12

- If you are thinking about switching to 106B, do so before tomorrow!

- Weekly review lecture by Clinton on Wednesdays at 1:30

- If you have a chromebook/no laptop...come chat
Today

- Recap while loops and conditions
- Introduce if/else statements
- Introduce Decomp
Recap: While Loop

```python
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 looping process is over, and the “code doesn’t loop” runs
The classic move-forward loop

```python
while bit.front_clear():
    bit.move()
# bit will always be blocked here!
```
The classic move-forward loop

```python
while bit.front_clear():
    bit.move()
# bit will always be blocked here!
```
The classic move-forward loop

```python
while bit.front_clear():
    bit.move()
# bit will always be blocked here!
```
The classic move-forward loop

```python
while bit.front_clear():
    bit.move()
# bit will always be blocked here!
```

2nd loop
The classic move-forward loop

```python
while bit.front_clear():
    bit.move()
# bit will always be blocked here!
```
The classic move-forward loop

```python
while bit.front_clear():  
    bit.move()
# bit will always be blocked here!
```
The classic move-forward loop

```python
while bit.front_clear():
    bit.move()
# bit will always be blocked here!
```
What would happen?

```python
while bit.front_clear():
    bit.move()
bit.move()
```

What happens if bit tries to move when it is blocked?
Runtime error!

```python
while bit.front_clear():
    bit.move()
bit.move()
```

Case-2: Runtime Error

Exception: Bad move, front is not clear, in: do_left line 5 (Case 2)
The classic move-forward loop

```python
while bit.front_clear():
    bit.move()
# bit will always be blocked here!
```

Key idea: must always check if Bit’s front is clear before calling `bit.move()`
The classic infinite loop

```python
while bit.front_clear():
    bit.paint('green')
# bit will always be blocked here!
```

1st loop
The classic infinite loop

```python
while bit.front_clear():
    bit.paint('green')
# bit will always be blocked here!
```
The classic infinite loop

```python
while bit.front_clear():
    bit.paint('green')
# bit will always be blocked here!
```

2nd loop
The classic infinite loop

while bit.front_clear():
    bit.paint('green')
# bit will always be blocked here!

2nd loop
The classic infinite loop

```python
while bit.front_clear():
    bit.paint('green')
# bit will always be blocked here!
```

3rd loop
The classic infinite loop

```python
while bit.front_clear():
    bit.paint('green')
# bit will always be blocked here!
```

3rd loop
The classic infinite loop

```python
while bit.front_clear():
    bit.paint('green')
# bit will always be blocked here!
```

Case-2: Timed Out

Run timed out. Try again, or possible infinite loop.
The classic infinite loop

```python
while bit.front_clear():
    bit.paint('green')
# bit will always be blocked here!
```

Key idea: The condition in the while loop should eventually be made False by the body of the while loop.
Recap: Conditions

- Conditions are statements that evaluate to either True or False

```python
while bit.front_clear():
    # front_clear returns True or False
```
Recap: Conditions

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

```python
while bit.front_clear():
    # front_clear returns True or False
```

- The statement `left == right` is True if `left` is equal to `right` and False otherwise.

```python
while bit.get_color() == 'green':
    # if bit.get_color returns 'green',
    # this condition is True
```
Recap: Conditions

- Conditions are statements that evaluate to either True or False

\[
\text{while bit.front_clear() :}
\]
\[
\quad \# \text{front_clear returns True or False}
\]

- The statement \text{left == right} is True if left is equal to right and False otherwise
- Adding the word \text{not} in front of a conditions changes it from False to True or from True to False

\[
\text{while not bit.get_color() == 'green' :}
\]
\[
\quad \# \text{if bit.get_color returns ‘green’,}
\quad \# \text{this condition is False}
\]
**go niche**

- Bit will start at some level of the world, on the left and facing right

- Every level below her will be blocked

- Except one “hole”

- The hole will have a one-block niche on the right side

- Get Bit to that niche
Aside: Experimental Server Tricks

- Cmd-Return (Mac) or Ctrl-Return with cursor in code will Run (very handy when pounding away on your code)

- 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
More practice: Bit Loop
Today

- Recap while loops and conditions
- Introduce if/else statements
- Introduce Decomp
If statements

```
if #condition:
    # block 1 runs if condition is True
# block 2 that runs regardless
```

1. First the condition is checked (it should be True or False, more on that later)

2. If the condition is True, the code in block 1 runs, otherwise skip to step 3

3. The code in block 2 runs
Move bit (at most) once

```python
if bit.front_clear():
    bit.move()
    bit.paint('green')
```
Move bit (at most) once

```python
if bit.front_clear():
    bit.move()
bit.paint('green')
```
Move bit (at most) once

```python
if bit.front_clear():
    bit.move()
bit.paint('green')
```

Bit paints the **second square green**
Move bit (at most) once

```python
if bit.front_clear():
    bit.move()
bit.paint('green')
```
Move bit (at most) once

```python
if bit.front_clear():
    bit.move()
bit.paint('green')
```

Bit paints the **first square green**
Move bit (at most) once

```python
if bit.front_clear():
    bit.move()
bit.paint('green')
```

Key idea: Bit may or may not move, but she will always paint green
If/else statements

```
if #condition:
    # block 1 runs if condition is True
else:
    # block 2 runs if condition is False
# block 3 runs regardless
```

Key idea: *exactly one* of block 1 and block 2 will run: never both, never neither. Block 3 always runs
If/else statements

```python
if bit.front_clear():
    bit.paint('green')
else:
    bit.paint('red')
bit.right()
```
If/else statements

```python
if bit.front_clear():
    bit.paint('green')
else:
    bit.paint('red')
bit.right()
```
If/else statements

```python
if bit.front_clear():
    bit.paint('green')
else:
    bit.paint('red')
bit.right()
```
If/else statements

```python
if bit.front_clear():
    bit.paint('green')
else:
    bit.paint('red')
bit.right()
```
If/else statements

```python
if bit.front_clear():
    bit.paint('green')
else:
    bit.paint('red')
bit.right()
```
If/else statements

```python
if bit.front_clear():
    bit.paint('green')
else:
    bit.paint('red')
bit.right()
```
Put it all together: loops+ifs

double_move
Today

- Recap while loops and conditions
- Introduce if/else statements
- Introduce Decomp
Another look at functions

- In every bit exercise so far, we have implemented only 1 function to solve the entire problem - we see `def` only once

```python
def bit_func(filename):
    # all the code to solve the problem!
```
Another look at functions

- In every bit exercise so far, we have implemented only 1 function to solve the entire problem - we see `def` only once.

- We often call bit-specific functions while solving:

```python
def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()
```

- “Calling” a function means to run its code - your solution function is “called” by the experimental server when you hit run.
Calling functions

- Only Bit knows about `move` and `front_clear`, so we have to access them through Bit when calling with `bit.move()`

- But the function `go_south` is available for anyone to call!

```python
def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()
```
We call `go_south` in another function like so:

```python
def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()

def paint_south(filename):
    bit = Bit(filename)
    go_south(bit)
    bit.paint('green')
```
Run paint_south

```python
def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()

def paint_south(filename):
    bit = Bit(filename)
    go_south(bit)
    bit.paint('green')
```
def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()

def paint_south(filename):
    bit = Bit(filename)
    go_south(bit)  # function call!
    bit.paint('green')
def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()

def paint_south(filename):
    bit = Bit(filename)
    go_south(bit)  # pause here
    bit.paint('green')
def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()

def paint_south(filename):
    bit = Bit(filename)
    go_south(bit) # pause here
    bit.paint('green')
Run paint_south

def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()  # done!

def paint_south(filename):
    bit = Bit(filename)
    go_south(bit)  # done here too!
    bit.paint('green')
Run paint_south

def go_south(bit):
    bit.right()
    if bit.front_clear():
        bit.move()  # done!

def paint_south(filename):
    bit = Bit(filename)
    go_south(bit)
    bit.paint('green')
Calling functions recap

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

def func2():
    # code block B
    func1()
    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!

```python
def soln(filename):
    bit = Bit(filename)
    go_to_far_wall(bit)
    spin(bit)
    paint_3(bit)
```
Why make multiple functions?

```python
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
Put it all together: **fill_world_blue**

- Bit starts at the top-left corner of the world facing **down**
- The world has no obstacles (black squares)
- Fill every square in the world blue
- Use the provided function **fill_row_blue()**
Investigate fill_row_blue

- When using “helper” functions to solve a bigger problems, it is good to define the pre and post conditions for that helper

- **Pre conditions:** does fill_row_blue assume that bit is facing a certain direction? Does it assume she is unblocked?

- **Post conditions:** What does the world look like after calling fill_row_blue? Where is bit? Where is she facing?
Investigate fill_row_blue

- When using “helper” functions to solve a bigger problems, it is good to define the pre and post conditions for that helper

- **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
Lets code: **fill_world_blue**
If time: **blue dip**
Bonus
if/elif and if/elif/else

We will revisit this later in the quarter!
If/elif statements

```python
if #condition1:
    # block A runs if condition1 is True
elif #condition2:
    # block B runs if condition 1 is False
    # and condition2 is true
elif #condition3:
    # block C runs if conditions 1 and 2
    # are False and condition3 is true
    # Can have many more elifs here
```

Key idea: *at most one* of block A, B and C will run, but it’s possible for none to run
If/elif statements

```python
if bit.get_color() == 'red':
    bit.paint('green')
elif bit.get_color() == 'green':
    bit.paint('red')
elif bit.get_color() == None:
    bit.paint('blue')
bit.right()
```
If/elif statements

```python
if bit.get_color() == 'red':
    bit.paint('green')
elif bit.get_color() == 'green':
    bit.paint('red')
elif bit.get_color() == None:
    bit.paint('blue')
bit.right()
```
If/elif statements

```python
if bit.get_color() == 'red':
    bit.paint('green')
elif bit.get_color() == 'green':
    bit.paint('red')
elif bit.get_color() == None:
    bit.paint('blue')
bit.right()
```

Note: we don’t check any of the other conditions once we run a block (even though the second condition would now be true)
If/elif statements

```python
if bit.get_color() == 'red':
    bit.paint('green')
elif bit.get_color() == 'green':
    bit.paint('red')
elif bit.get_color() == None:
    bit.paint('blue')
bit.right()
```

English summary of this code snippet?
If/elif statements

```python
if bit.get_color() == 'red':
    bit.paint('green')
elif bit.get_color() == 'green':
    bit.paint('red')
elif bit.get_color() == None:
    bit.paint('blue')
bit.right()
```

English summary of this code snippet?
If a square is red or green, switch it to be the other one, and if its blank, make it blue. (do nothing to blue squares)
If/elif/else statements

```
if #condition1:
    # block A runs if condition1 is True
elif #condition2:
    # block B runs if condition 1 is False
    # and condition2 is true
    # Can have many more elifs here
else:
    # block C runs if conditions 1 and 2
    # are False
```

Key idea: exactly one of block A, B and C will run, never none, never more than one
If/elif/else statements

```python
if bit.front_clear():
    bit.move()
elif bit.right_clear():
    bit.right()
    bit.move()
elif bit.left_clear():
    bit.left()
    bit.move()
elif:
    bit.paint('red')
```
If/elif/else statements

```python
if bit.front_clear():
    bit.move()
elif bit.right_clear():
    bit.right()
    bit.move()
elif bit.left_clear():
    bit.left()
    bit.move()
else:
    bit.paint('red')
```

English summary: Bit will move at most once, in the first clear direction she finds, or she will paint red.
### If/elif/else statements

```python
if bit.front_clear():
    bit.move()
elif bit.right_clear():
    bit.right()
    bit.move()
elif bit.left_clear():
    bit.left()
    bit.move()
```

How are 3 ifs different from if, elif, elif?

```python
if bit.front_clear():
    bit.move()
if bit.right_clear():
    bit.right()
    bit.move()
if bit.left_clear():
    bit.left()
    bit.move()
```
If/elif/else statements

```python
if bit.front_clear():
    bit.move()
elif bit.right_clear():
    bit.right()
    bit.move()
elif bit.left_clear():
    bit.left()
    bit.move()
```

How are 3 ifs different from if, elif, elif?
Bit could move at most once with the code on the left, but could move many times on the right.

```python
if bit.front_clear():
    bit.move()
if bit.right_clear():
    bit.right()
    bit.move()
if bit.left_clear():
    bit.left()
    bit.move()
```
Recap

- While loops are powerful and we can use any condition as our test to keep going!

- We can also use if statements with any conditions to run something only **once** if the condition is true

- We can decompose big problems into smaller functions, and call them from our main solution function!