Schedule

- Friday 11/14
  - Chapter 8
- For Monday 11/17
  - Read Chapter 8

`while`

- `while` is like `if`, except...
  - With `if`, you check the condition; if it's `True` you run the indented code.
  - With `while`, you check the condition; if it's `True` you run the indented code; then you check it again...*and keep going until the condition becomes* `False`.
- `while` is more general than `for`.
  - With `while` you have more flexibility.
  - With `while`, it's much easier to (accidentally) create an infinite loop—*Always check that there's something inside the loop that will eventually cause the condition to change from* `True` *to* `False`.

`while` vs. `for`

- How would you write the following loop using `while` instead of `for`?

```
for i in range(len(s)):
    print(ord(s[i]), end=" ")
```

- Hints...
  - What needs to happen *once at the beginning*?
  - What condition needs to be checked *at the beginning of each loop iteration*?
  - What needs to happen *at the end of each loop iteration*?

Common Indefinite Loop Patterns

- *Interactive* Loop
  - Run loop code, *ask for input* ("Do you want to do this again?"), continue or exit loop based on the response you get.
  - To make this work, you need *a* `while` *loop with a condition to check the user's response.*
- *Sentinel* Loop
  - If the loop code inputs data with every iteration, you don't need to ask the user whether they want to continue.
  - Instead, specify a *sentinel* value that, if input as data, signals that there's no more data (and the program should therefore exit the loop).
  - The sentinel value *must not be a valid data value.*

What if you had `readline` and had to write `readlines`?

- *You'd need an indefinite loop*, because you don't know how many lines there are in the file.
  - You can use `readline` inside the loop to read one line in each loop iteration.
  - If you get `""` when you try to read a line, you've reached the end of the file. (You get `\n` for a blank line.)
  - So *this is a sentinel loop, and* `""` *is the sentinel value.*
- In Python you don't need to write this yourself, but in other languages you might.

Nested Loops

- You can put any statement(s) you want inside a loop, *including the statements of another loop.*
  - The most common use of a nested loop is *to loop through a set of combinations* (red, green and blue values, for example).
  - It's also common to have code with *a definite loop inside an interactive* ("Do you want to quit yet?") *loop.*
- Nested loops make code significantly *more difficult to understand.*
  - Unless the pattern is obvious (like looping through a set of combinations) *consider putting the inner loop in a separate function.*

Boolean Operators

- A *Boolean expression* is an expression that has a value of either `True` or `False`.
  - Two expressions (with numeric or string values) joined by a relational operator: `(x ** 2) < n`
  - A single variable (assigned `True` or `False`): `prime = False`
  - *Two Boolean expressions joined by a Boolean operator...*
- Boolean Operators
  - `x and y` (`True` if both `x` and `y` are `True`)
  - `x or y` (`True` if either is—*or both* are—`True`)
  - `not x` (`True` if `x` is `False`)

Post-Test Loop

- Suppose you want to create an indefinite loop, but *there isn't a good way to make the condition `True` before the first iteration...*
  - If it would require duplicated code before and then inside the loop, for example.
- You can use a *post-test loop*:

```
while True:
    # Do something interesting; give allDone a value.

    if allDone:  # Same as "if allDone == True:"
        break
```