Loops and Conditionals

Loops and Conditionals

Control Structures

The real power of programming languages comes into play with the use of *control structures*. Control structures control the flow of a program, telling it what to do next, and in what order. All programming languages use the same basic set of control structures, and just differ in how they're written out. In this module, we'll be learning about the three main kinds of control structures and how they're implemented in python:

- Conditionals, or `if`, `else` statements
- `for` loops
- `while` loops

Conditionals

Conditional statements tell python to do something `if` some condition is met. For example, we can write a little block of code that will tell us if a number is odd or even. If you divide an even number by 2, it will have no remainder, and if you divide an odd number by 2, it will have a remainder of 1. To get the remainder of division, we can use the `%` operator.

```
100 % 2
# 0
117 % 2
# 1
```

We can write a short block of code that will print `<x> is even` if the number assigned to `x` is even like so. First, let's assign x = 50 on its own line:

```
x = 50
```

Now, we can write our `if` statement.

```
if x % 2 == 0:
    print(x, "is even")

# 50 is even
```

Let's break down what's going on here. First, we have a logical statement `x % 2 == 0`. This statement is checking to see if the outcome of `x % 2` is exactly equal to `0`. If we run it as its own line of code, we'll get `True`

```
x % 2
# 0

x % 2 == 0
# True
```

What the `if <...> :` part of the code is saying is if the statement that comes between `if` and the colon `:` comes back as `True`, it will do something. In the example above, because the value `50` was assigned to `x`, then `x % 50 == 0` comes back as `True`, therefore, python will run whatever code comes on the next indented line.

Code: `if x % 2 == 0:`
`print(x, "is even")`

Description: If this part is True, then do this.

The syntax and formatting of control blocks

An important thing to keep straight when programming is which pieces of code should only be run when the `if` statement is true, and which parts to run no matter what. Every programming language is going to do this a slightly different way. For example, here's how a different programming language, R, does it.

```
## This is R code! Just here for illustration!
if(50 %% 2 == 0){
  print("it is even")
}
print("Program done!")
```

In this programming language, the code that runs if the `if` statement is true is contained within curly brackets.

Python already uses curly brackets `{}` to **create dictionaries (https://uk.instructure.com/courses/2051722/pages/lists-and-dictionaries)** , so it doesn't use it here. Instead, it uses *indentation* to define which pieces of code are being controlled by (or, within the "scope" of) the `if` statement.

```
# This is python code again!
if x % 2 == 0:
    print(x, "is even.")
    print("This line only prints if it's even too.")
print("This line prints no matter what.")
```

If you run this block of code, and `x` still has the value `50`, all three lines will print.

```
# 50 is even.
# This line only prints if it's even too.
# This line prints no matter what.
```

However, if you change the value assigned to `x` to an odd number, like `51`, and re-run the that block of code, only the last line will be printed.

```
# This line prints no matter what.
```

This is, in general, how python defines the scope of a control statement.

Lines of code with the same degree of indentation are controlled by the same control structures.

The greater the indentation, the narrower the scope of the code.

Else statements

Let's come back to our original `if` statement.

```
if x % 2 == 0:
    print(x, "is even")
```

When `x` is an even number, this `if` statement will run the `print()` command. But when `x` is an odd number, it doesn't do anything. If we want to print out a statement telling us whether the number is odd, we need to add an additional control structure: `else`.

An `else` statement can only come at the end of an `if` block. If the conditionals above have all been `False`, python will run whatever code comes after `else`. Here's how it looks:

```
if x % 2 == 0:
    print(x, "is even")
else:
    print(x, "is odd")
```

With this block of code, when `x` has been assigned `50`, python will print `50 is even`, and when `x` has been assigned `51`, python will print `51 is odd`.

Code:
```
if x % 2 == 0:
    print(x, "is even")
else:
    print(x, "is odd")
```

Description:
If this part is True, then
do this.
If all else fails, then
do this.

Elif

There's one last component to our `if` statements that will fully round them out. Right now, here's the steps that happens in our `if` block:

1. It does remainder division of 2 on `x`.
2. It checks if that result is equal to `0`
3. If that returns `True`, it prints `<x> is even`
4. If every condition above returned `False`, it prints `<x> is odd`

This is going to work for all integers, but let's think about what would happen if we assigned a number with a decimal value, like `50.1` to `x`.

1. 50.1 divided by 2 has a remainder of 0.1.
2. 0.1 is not equal to 0.
3. `0.1 == 0` returns `False`, so we don't print anything yet.
4. All the conditions above returned `False`, so it prints `50.1 is odd`.

This is not quite right. It seems like what we really want to do is have some kind of check first, to see if the value in `x` has a decimal value before we do the rest of our even and odd checks.

There are lots of little math tricks we could do to check to see if `x` has a decimal place, but I think what I'll want to do is check to see of remainder division returns a remainder that is greater than 0 and less than 1.

```
x = 50.1
x % 1 > 0 and x % 1 < 1
```

So, the start of our `if` block will start with this decimal check.

```
if x % 1 > 0 and x % 1 < 1:
    print(x, "is a decimal.")
```

Now we're ready to run our even and odd checks that we worked out before, but we need to do so in a way that tells python that they belong to the same block of checks. We only want python to move on to the even and odd checks if the number is not a decimal. We do this with `elif`, which is a mashup of `else` and `if`. It only runs if the block above was `False`, and it does its own logical check before running its own code.

Our final block of code to classify numbers into even, odd, or decimals, is:

```
if x % 1 > 0 and x % 1 < 1:
    print(x, "is a decimal.")
elif x % 2 == 0:
    print(x, "is even")
else:
    print(x, "is odd")
```

Go ahead and check it out with any given number.

Code:
```
if x % 1 > 0 and x % 1 < 1:
    print(x, "is a decimal")
elif x % 2 == 0:
    print(x, "is even")
else:
    print(x, "is odd")
```

Description:

If this part is True, then
do this.
Otherwise, if this part is True, then
do this.
If all else fails, then
do this.

More example: Is item in list

Let's say we wanted to identify which of the Brontë sisters wrote a particular novel. We'd first need to create a list of novels for each sister.

```
charlotte = ["The Professor", "Jane Eyre", "Shirley", "Villette"]
emily = ["Wuthering Heights"]
anne = ["Agnes Grey", "The Tenant of Wildfell Hall"]
```

Now, we can test novel titles against each list.

```
novel = "Jane Eyre"
```

```
if novel in charlotte:
    print("Charlotte Brontë wrote", novel)
elif novel in emily:
    print("Emily Brontë wrote", novel)
elif novel in anne:
    print("Anne Brontë wote", novel)
else:
    print(novel, "was not written by one of the Brontë sisters")
```

More Examples: Counting

If we complexify our Brontë checking code, we can also keep track of how many times we've checked whether a particular sister wrote a book. First, we need to *initialize* a dictionary that will keep track.

```
bronte_checks = {"charlotte": 0, "emily": 0, "anne" : 0}
```

Now, we include a line of code in each part of the `if` block to increase the value in the dictionary by 1.

```
if novel in charlotte:
    bronte_checks["charlotte"] = bronte_checks["charlotte"] + 1
    print("Charlotte Brontë wrote", novel)
elif novel in emily:
    bronte_checks["emily"] = bronte_checks["emily"] + 1
    print("Emily Brontë wrote", novel)
elif novel in anne:
    bronte_checks["anne"] = bronte_checks["anne"] + 1
    print("Anne Brontë wote", novel)
else:
    print(novel, "was not written by one of the Brontë sisters")
```

For Loops

Where `if` statements are used to control which pieces of code are run, depending on certain conditions, `for` loops are used to run the same block of code over and over again on items in an *iterable*. For simplicity, let's just stick to lists.

Let's take our list of novels that Charlotte Brontë wrote from above as an example.

```
charlotte
# ['The Professor', 'Jane Eyre', 'Shirley', 'Villette']
```

If we wanted to print out the name of every novel, one way we could approach it is just to do it by hand with indexing.

```
print(charlotte[0])
print(charlotte[1])
print(charlotte[2])
print(charlotte[3])
```

However, not only is this an inefficient use of our time, it's also not going to work if we don't know how many items are in the list, which we won't most of the time.

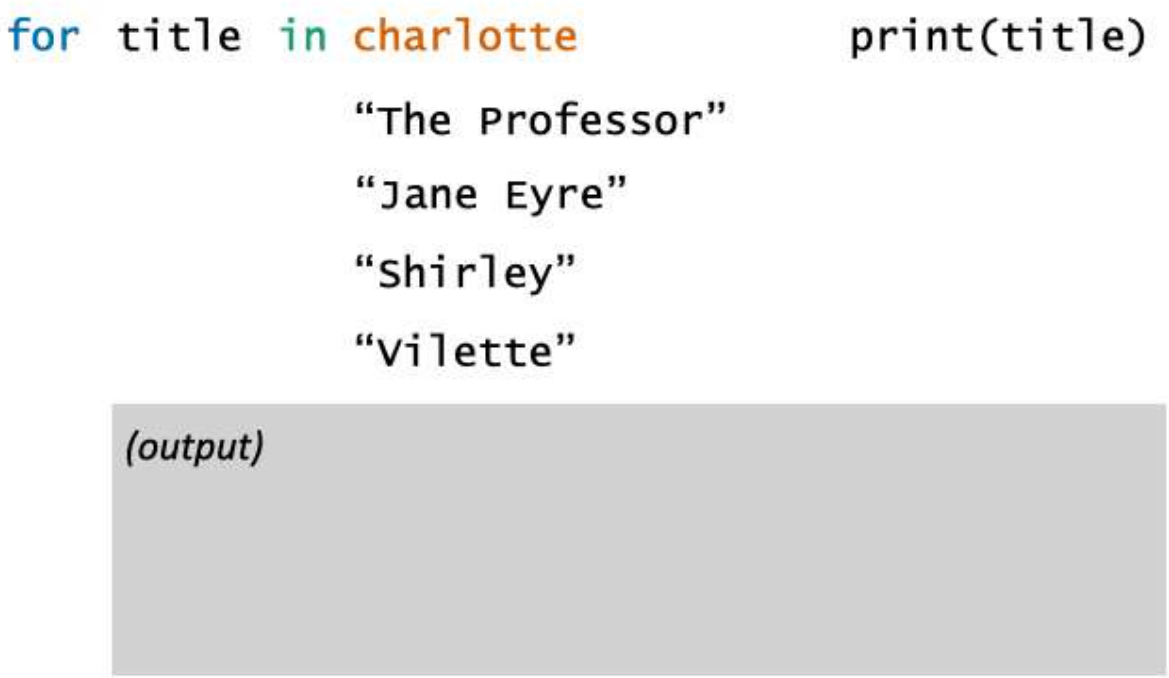
Instead, we'll use a `for` loop to run the `print()` function on every value in the list automatically.

```
for title in charlotte:
    print(title)

# The Professor
# Jane Eyre
# Shirley
# Villette
```

This `for` loop is saying that we want to do *something* for every value inside the list `charlotte`. And as we loop through every value in the list `charlotte`, it's going to assign that value to a variable called `title`. Instead of saying `for title in charlotte:` we could have said `for book in charlotte:`, and it would have done the same thing, as long as we used the variable `book` instead of `title`.

Then, every time the `for` loop assigns a new value from `charlotte` to the variable `title`, it runs the indented code.



We don't *have* to do anything with the variable `title` inside of the loop. Python will still run whatever code is inside of the loop every time it updates the value assigned to `title`. For example, if we forgot that the `len()` function gave us the length of a list, we could count the number of books Charlotte Brontë wrote like this:

```
book_counter = 0

for title in charlotte:
    book_counter = book_counter + 1
```

```
print(book_counter)
# 4
```

Setting up an empty *counter* or a *collector* before starting a `for` loop is a very common thing to do. For example, if I wanted to know how many characters Charlotte Brontë used in the title of each book, and in total over all, I could do this:

```
total_characters = 0
book_characters = []

for title in charlotte:
    # exclude spaces
    no_space = title.replace(" ", "")

    # Update total counter
    total_characters = total_characters + len(no_space)

    # update by-book list
    book_characters.append(len(no_space))
```

More examples: Preparing Lines

In **the previous module on reading in text (https://uk.instructure.com/courses/2051722/pages/using-text-and-strings)**, we went over how text often needs to be cleaned up before we can get ready to analyze it. Whitespace characters need to be stripped off of the left and right edges, and we need to convert each line into either lowercase or uppercase.

```
# define the book path
book_location = "data/mansfield_park.txt"

# open the text file
book_file = open(book_location, mode = 'r')

# read in all lines
book_lines = book_file.readlines()

# get one line
one_line = book_lines[200]

# uncleaned line
one_line
'ours, Sir Thomas, I may say, or at least of _yours_, would not grow up\n'

# cleaned line
one_line.lstrip().rstrip().lower()
# 'ours, sir thomas, i may say, or at least of _yours_, would not grow up'
```

But, there are 16,045 total lines in *Mansfield Park*. It would be inefficient to do this to each line individually. Instead, we'll do it with a `for` loop with the following steps.

1. Create an empty collector list.
2. For each line in `book_lines` clean it up.

3. Add the cleaned up line to the collector list.

```
# collector
clean_lines = []

for line in book_lines:
    # cleanup
    cleaned = line.lstrip().rstrip().lower()

    # collection
    clean_lines.append(cleaned)
```

More examples: Full Lines

If you look at `clean_lines`, you'll see that there are a bunch of lines that are zero-length. They look like just `''`. These were blank lines in the original text used for things like separating paragraphs. They're not all that important for our analysis, so we'll get rid of them by combining a `for` loop and an `if` statement. We'll do it with the following steps:

1. Set up a counter, set to 0, to keep track of the number of blank lines
2. Set up an empty collector list.
3. Loop through every value in `clean_lines`, assigning it to `line`.
4. If the length of the value in `line` is greater than 0, append it to the collector list.
5. Otherwise, add 1 to `n_blank`.

```
# a counter, just to see
n_blank = 0

# a collector
full_lines = []

for line in clean_lines:
    if len(line) > 0:
        full_lines.append(line)
    else:
        n_blank = n_blank + 1
```

Doing this, we can see that there were 2,110 blank lines in the text, and now full_lines only contains lines with text!

More examples: Lists of words!

We also saw how to split up a line of text into a list of words with the `.split()` method. If we split a line into words by using spaces, it looks like this:

```
full_lines[200]
# 'on this point. whatever i can do, as you well know,
# i am always ready'
```

```
full_lines[200].split(" ")
# ['on',
#  'this',
#  'point.',
#  'whatever',
#  'i',
#  'can',
#  'do,',
#  'as',
#  'you',
#  'well',
#  'know,',
#  'i',
#  'am',
#  'always',
#  'ready']
```

If we wanted to turn every line into a list of words, we could do this with a for loop with the following steps.

1. Set up an empty collector list.
2. Loop through every value in `full_lines`, assigning it to `line`.
3. Split `line` into a list of words based on the spaces.
4. Append this list of words to the collector list.

```
# set up collector
word_lists = []

for line in full_lines:
    # split on spaces
    words = line.split(" ")

    # append to collector
    word_lists.append(words)
```

Now, `word_lists` is a list of lists.

More Examples: Getting the average number of words per line

If we wanted to know how many words there are per line, we'd need to get the total number of words, then divide it by the number of lines. I'll do this with a `for` loop with the following steps:

1. Set up an empty collector list for the length of each list of words.
2. Loop through all of the values in word_lists, assigning each value to the variable `words`.
3. Append the length of `words` to the collector list.
4. Get the sum of all values in the collector list with `sum()`.
5. Divide that by the length of `word_lists`.

```
# set up collector
line_len = []

for words in word_lists:
    # get length of list
    n = len(words)
```

```
    # append to collector
    line_len.append(n)

total_words = sum(line_len)
total_lines = len(word_lists)

average_len = total_words / total_lines
```

Turns out, the average number of words per line is 11.68 words.

More Examples: How many "the"

Sometimes, we may need to have a `for` loop inside of a `for` loop. Right now, `word_lists` is a list we can loop over, and every value in it is also a list we can loop over. One use case for embedding two `for` loops here would be if we wanted to count up how many instances of "the" there are in the book. Since the "words" in each list still have punctuation attached to them, we'll have to use some regular expressions.

1. Create a counter for the number of "the", set to 0.
2. Loop every value in `word_lists`, assigning it to `words`.
3. Loop through every value in `words`, assigning it to `w`.
4. If `w` is a match for the regular expression `\bthe\b`, add 1 to the counter.

```
import re

# Set up counter
the_counter = 0
total_word_counter = 0

for words in word_lists:
    for w in words:
        total_word_counter = total_word_counter + 1

        if re.match(r'\bthe\b', w):
            the_counter = the_counter + 1
```

It turns out that there are 6,340 instances of "the" in *Mansfield Park.* In order to get the proportion of all words which were "the", we just need to do `the_counter / total_counter`.

While Loops

Another kind of loop is the `while` loop. These are a lot like `if` statements. In an `if` statement, when the condition is `True`, python runs the indented code once.

```
x  = 0
```

```
if x < 10:
    print(x, "is less than 10.")
```

In `while` loops, there is also a logical condition, but instead of running the indented code just once, python loops back and re-checks the conditional. If it's still `True`, it runs the indented code again, and loops back to re-check the conditional. It will keep doing this forever until the conditional returns `False`.

```
while x < 10:
    print(x, "is less than 10.")
    x = x - 1

# 0 is less than 10
# 1 is less than 10
# 2 is less than 10
# 3 is less than 10
# 4 is less than 10
# 5 is less than 10
# 6 is less than 10
# 7 is less than 10
# 8 is less than 10
# 9 is less than 10
```

It is important to write `while` loops carefully! It is very easy to write a `while` loop that will accidentally run forever. If you think you've done this, hitting CTRL + c on your keyboard will interrupt the process.

We usually won't be using `while` in this class. They're mostly used when you want a script to wait for user input. We can do kind of fun looking things with them though.

```
name = "The University of Kentucky"

while name:
    print("|" + name + "|")
    name = name[0:-1]
```