Day 4 Notes

#coding-intro

This note discusses a fundamental element of most Turing complete programming languages: control of program execution. We'll be exploring the features of the Python language that enable this control, but the same ideas and construct exist in many other languages, including Java, C, and so on.

> Turing completeness is a fancy way of saying that a programming language is able to describe any program that computes anything that *can* be computed. In practical terms, this means any application you see on your computer today (this definition is overly simplistic, but the details are not important for this note).

Motivation for control

So far, programs we've seen flow from start to finish in a completely predictable manner. Let's say we write a "calculator" program that looks like this:

```
z = 10 + 10
print(z)
```

No matter how many times we run this program, we'll get the same output (20). This tends to be not too useful since once we've calculated this value, there's not much value to this program.

Real world calculators **take in user input and react accordingly.** For example, you may want to add together two numbers one day, and then divide two numbers the next. Having one program that can handle all of this is great, but is impossible with the skills we've learned so far. This is one of many reasons why we would like to control the execution of the program.

In English, we would like the program to behave like this:

```
ask user for two values: x and y
read in x and y
```

```
ask user to reply with add/sub/mul/div
if user responded with add: print x + y
else, if user responded with sub: print x - y
else, if user responded with mul: print x * y
otherwise, print x / y (we're assuming they put "div")
```

This is a much more useful program. Although we will not cover how to read user input directly from running a Python script, you can still take in your own input by calling your function with different arguments (operands).

Boolean expressions

While everyone is familiar with numbers, there is another useful type of data known as boolean values. Booleans are either **true** or **false**. This is very useful for control, because you often need to make a decision based on if an expression evaluates to something that is considered true or false. For example, here are some common mathematical comparisons:

```
>>> 10 > 5
True
>>> 11 < 10
False
>>> x = 4  # This is an assignment statement
>>> x == 4 # This compares the values of two expressions
True
```

Boolean operators

Much like how mathematical operators combine numbers to produce an numerical output, boolean operators combine true/false to produce a true/false output.

- `and` takes in two expressions, and returns `True` if they are both truthful. Otherwise, it returns `False`.
- `or` takes in two expressions, and returns `False` if they are both "falsy". Otherwise, it returns `True`.

> There is a important distinction between `False` and something "falsy" (and a distinction between something `True` and truthful). To be precise, I didn't want to limit it to just `False`,

but anything `False` is considered "falsy".

- `not` takes in a **single** expression, and returns the negation of the result. So something truthful becomes `False`, and something falsy becomes `True`.

`if` statement

The `if` statement is very similar to what the calculator example looked like above. The basic structure looks like this:

```
if <e1>:
  <et1>
elif <e2>:
  <et2>
elif <e3>:
  <et3>
...
else:
  <ef>
```

`<e1>`, `<e2>`, `<e3>`. are expressions. `<et1>`, `<et2>`, and `<ef>` represent one or more expressions. Take note of the indentation levels.

You can have any number of `elif` clauses (which is an `elif`, an expression on the same line, and an indented code block following it), but they must appear after an `if` clause and before the `else` clause (if it exists). There is a maximum of 1 `else` clause per `if` clause. Here is a more concrete example:

```
x = 10
if x < 15:
  print("medium")
elif x < 30:
  print("large")
else:
  print("extra large")

if x > 0:
```

```
  print("small")
```

In this case, the first `x < 15` would evaluate to `True`. So we execute the code under the first `if` and print "medium". According to the rules of the `if` statement, we don't check any of the other `elif` or `else` associated with this `if` once we hit something that's `True`. We skip ahead to the next `if` statement.

At the next `if`, `x > 0` evaluates to `True` and we print "small".

The total output would be "medium" and then "small".

What would the output be if `x = 17`? We would get "large" from the `elif` clause and "small" from the last `if`.

`while` loop

The while loop has the following structure:

```
while <e1>:
  <et>
```

Again, `e1` is an expression and `<et>` is one or more expressions.

We first check to see if `e1` evaluates to something truthful. If so, we execute `<et>` (which could be one or more expressions. We then repeat our previous steps.

If `e1` ever becomes something "falsy", then we skip `<et>` and go on to the next line. Here's an example using `while`:

```
x = 0
while x < 4:
  print(x)
  x = x + 1
print(x)
```

What will this display? First, `x` will be 0, so we print 0. Then we increment `x` till we hit 4. Finally, we print `x` one more time. Therefore, the total output is:

```
0
1
2
3
4
```

Resources

Same resources as Note 1, but copied here again:

- Learn Python has a free tutorial for Python.
- Codecademy has a good Python tutorial, but don't pay for their pro course. You'll need to make a free account to access it. They also have lessons for other languages.
- CS 61A is UC Berkeley's intro to computer science course. Much of this course is based on the information from the early lectures! The lecture videos and assignments are available online. Check out Spring 2018 here, which is taught by a professor. In the summer, it is taught by students.