Day 1 Notes

#coding-intro

Introduction

Although course slides are published online, they don't contain very detailed information. The purpose of these notes is to provide a summary of what I went over in class.

Why Programming?

- **What does a computer do?** They're great at doing simple, repetitive tasks.
- **What is programming?** Computers can't read your mind (yet), so we write programs to tell them what to do. Computer code are the instructions you give to a computer, and they follow rules according to a programming. Programming is commonly defined as the act of writing computer code.
- **Why should you care?** If you ever want to become a computer engineer or programmer, then learning how to program is a must. Even if you never plan on writing a single line of code again, however, it's still important to know a little bit of what's going on behind the scenes.

Setup

Please install the following (ask for help if you run into any issues):

- A terminal emulator (macOS comes with one)
- Python programming language
- Atom, a text editor

Python

Python is a popular, free, and easy to learn programming language.

Python also comes with a helpful **interpreter**, which you can use to run code. You can also put Python code into a file and then run it later.

To start the interpreter, first open up your terminal, and then enter the following:

```
python3
```

If you're on Windows, you may need to use:

```
python
```

You'll see three angle brackets `>>>` which indicates that you can now type in Python code. Try the following (don't type `>>>`):

```
>>> print("Hello world!")
```

Then press enter. You'll should see some output. You can also put in arithmetic expressions to do some quick mathematics:

```
>>> 2 + 2 - 1
3
```

When you're done, use the following to exit the interpreter:

```
>>> exit()
```

Writing Programs

Every Python program is made up of expressions and statements.

Expressions

An expression describes a computation and evaluates it to a value. The most familiar expressions are probably the arithmetic expressions you've seen in your math class. Open up the interpreter again and try some out.

```
>>> 2 * 2
```

```
4
>>> 2 ** 3
8
>>> 8 / 3
2.6666666666666665
```

Every expression has an **operator**, and an optional number of **operands**. Operands are inputs to the operator, and the operator describes the rules for what should happen to the inputs. Just think of what you know from math expressions: the addition operator takes in *numbers*, and tells us to sum them together.

Therefore, in the example above, the Python interpreter is just **evaluating call expressions** – that is, calling a function.

This "infix" notation (where the operator comes between the operands) is familiar from mathematics, but is included primarily for convenience. The typical form of expressions is closer to this:

```
>>> mul(2, 2)
4
>>> pow(2, 3)
8
```

Note that the following lines above will not work in your interpreter unless you do the following:

```
>>> from operator import *
```

This form is called **function call notation**. There's an operator and a set of parentheses with operands within. A **call expression** is when you call a function. You can combine multiple call expressions together:

```
>>> add(mul(2, 3), 4)
```

```
10
```

In this case, our calls are nested. What comes first? The order is that you evaluate the operator first, then the operands, then finally *apply* the operator to the values of the operands. Applying is when we actually do the adding or multiplying!

For the example above, we evaluate in this order:

1. `add(mul(2, 3), 4)`. Look to the very left of this line – the operator is `add`. We are done evaluating `add`, so we need to evaluate the operands: `mul(2, 3)` and `4`.
2. `mul(2, 3)`. We do the same procedure from before: the operator is `mul`, and the operands are `2` and `3`. Finally, we can apply and get back 10.
3. Going back to `add(mul(2, 3), 4)`: the operand `mul(2, 3)` evaluates to 6, and `4` evaluates to 4. We can apply the operator now and get back 10!

Of course, math functions are built in. But what if you want to make your own functions? We'll get to that later.

Statements

The most common statement in Python is the assignment statement:

```
>>> x = 10
>>> x
10
>>> y = x + 20
>>> y
30
```

The assignment statement takes a name on the left of the `=` and an expression on the right. The expression is evaluated, and the value of that expression is assigned to the name.

There are also built-in names, which you can fetch using an **import** statement.

```
>>> pi
Error
```

```
>>> from math import pi
>>> pi
3.141592653589793
```

Comments

The computer won't read **comments**, they are just there for humans to read. To write a comment in Python, type `#` and everything afterwards will be part of that comment and ignored. You can do this in the interpreter too!

```
>>> 2 + 30 / 3 # Order of operations matters
12
```

Functions

Finally, we can write our own functions. Here's a simple one that just returns 10.

```
def ten():
    return 10
```

The **name** of the function is `ten`. It takes in zero operands. Note that the second line is indented. The special `return` statement describes the result of calling this function. Some functions don't have a return, and that's okay.

This function is a bit wonky, since you'd probably never see something like it in math. We can make something a bit more realistic:

```
def add_numbers(x, y):
    result = x + y
    return result
```

The **name** of the function is `add_numbers`. It takes in two **operands**, `x` and `y`. Now, there are two lines indented – this tells Python that those two lines are executed whenever this function is called.

You can probably guess what this function does, but just to make sure:

```
>>> add_numbers(10, 20)
30
>>> add_numbers(ten(), add_numbers(ten(), 20)) # You can nest your functions
too!
40
```

Navigating the command line

Much like how you can use Finder or Windows Explorer to navigate your files, you can use the terminal to do so as well. Here are useful commands:

- `ls dir_name` lists contents of directory
- `cd dir_name` changes directory
- `mkdir dir_name` makes a new directory
- `mv src dest` move src file to dest

Demo

Here's a cool demo with Shakespeare, taken from cs61a.org. Type each line separately into the Python Interpreter.

```
# Objects
# Note: Download from http://composingprograms.com/shakespeare.txt
shakes = open('shakespeare.txt')
text = shakes.read().split()
len(text)
text[:25]
text.count('the')
text.count('thou')
text.count('you')
text.count('forsooth')
text.count(',')
```

```
# Sets
words = set(text)
len(words)
max(words)
max(words, key=len)

# Reversals
'draw'[::-1]
{w for w in words if w == w[::-1] and len(w)>4}
{w for w in words if w[::-1] in words and len(w) == 4}
{w for w in words if w[::-1] in words and len(w) > 6}
```

Resources

- 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.