Lecture 1

Class Objective; Thinking Algorithmically; Hello World!; Variables; Values, Literals, Data Types; Variables and Expressions (and Printing)

1. Class Objective

Objective: to familiarize ourselves with ideas involved in creating computer programs.

We'll use the Python 3 language, and we'll place some focus on mathematical applications.

What is a program?

A set of instructions to be executed by a computer. May involve input and output.

Let's look at a few examples! [Here's where I show you Roots, and Digits, and Tetris.]

What is programming good for?

Video games, apps on your phone, every program on your desktop, the operating systems on your phone/desktop, software for embedded systems (e.g. thermostats, microwaves, space probes), data science/machine learning, quantitative finance ...

What about Python 3 in particular?

It is one of the most widely used languages on Earth -- and most of the other most widely used languages (e.g. C, C++, Java) have major similarities in syntax. (In fact, once you learn a first language, learning a second language is usually not nearly as difficult.) Python is also used frequently in fields like data science and quantitative finance, because of a vast array of well-developed libraries. At the same time, Python is generally recognized as being very welcoming to the newcomer.

By the end of this course, you will know a lot about automating simple repetitive tasks.

You will know almost nothing about serious programming :(

but you will be in a good place to start learning more and/or teaching yourself :)

How do programs and programming work?

Typically, computers take their instructions in **machine code** (0's and 1's). We're not really going to learn how that works.

Humans generally use a **high-level language** (like Python) to create **source code**; then, an **interpreter** (or in other languages, a **compiler**) translates this to a program in machine code, which the computer can run. We will learn to speak the high-level language, to learn how to get the computer to execute the instructions we want it to.

Source code: what you write to create a program. Often, this is done in an **Integrated Design Environment** (or IDE), which is a platform that contains text editing features, an interpreter, a debugger, etc.

High-level language: a language containing natural language elements, automated features, little direct interaction with RAM memory addresses.

Examples: Python, C, C++, C#, Java, R, Matlab, Ruby, PHP, etc.

This is to be contrasted with **low-level languages**, like Assembly, direct machine code.

Learning to speak a high-level language well enough to write source code isn't actually too bad. The hard part is putting your ideas into terms the language can accomodate.

2. Thinking Algorithmically

Example: Suppose you are given two positive integers. What is the process you go through to determine which is larger, or if they are equal?

To explain what I'm asking for:

You didn't come out of the womb knowing that $378 > 245$, or $98 < 1001$, or $4425 < 4437$, or $421 = 421$. Nor did anybody teach you these four specific facts; they taught you a process. What *exactly* is that process? Don't be clever -- explain the way you learned back when you were young!

(We'll assume that we all know how 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are ordered, but that we need to be taught about ordering bigger integers -- i.e. we're past Kindergarten, but haven't made it to 2nd grade yet. We'll also assume that our integers are less than 10 digits long for simplicity.)

Given two positive integers, which is larger, or are they equal?

Steps:

1. Count number of digits in each integer.
2. If they have different number of digits:

   The one with more digits is greater. (Finished!)
3. Otherwise:

   1. First, select the leftmost digits of both numbers for examination.

   2. If the digits we are examining are different:

      The integer with the higher digit is greater. (Finished!)

   3. If the digits we are examining are the rightmost digits, but they are the same:

      The integers are the same. (Finished!)

   4. Otherwise:

      Go back to step 2, except now examining the next digit to the right in each integer.

This contains:

- precise specification of type of input expected ("two positive integers")
- output (*first is larger*, *second is larger*, or *equal*)
- selection (the two "if-otherwise" forks)
- logical expressions (things that can be true or false, like "they have different numbers of digits")
- repetition ("go back to step 2")

This is an example of an **algorithm**: a step-by-step procedure for solving a problem. An algorithm ought to be precise, and it ought to finish executing after a finite number of steps.

The algorithm is just one part of the **development process**.

Development process:

1. Understand problem
2. Design algorithm
3. Code solution
4. Test
5. Are there bugs? (There always will be.) Then go back to step 2 and/or step 3.

Goals for code:

- Correct
- Readable/maintainable (comments!)
- Adaptable
- Time/space efficient

3. Hello world!

It's time to write our first program. When this class was taught in C++, this was a lot more annoying. Now, you just have to type out `print("Hello world")`. Type that out in the box below, and then press Shift-Enter. Jupyter, the program which runs the document you're reading right now, is linked to a Python 3 interpreter, so that pressing Shift-Enter will run the program.

```
In [ ]: # EXAMPLE 3a: Hello world!
        # Make a program that, when run, prints "Hello World"

        print("Hello world!")
```

So, when this program is run, a phrase prints out to the screen. Not exactly Tetris, but you gotta start somewhere.

We've learned our first command: the print function. There's more to say about this, but for the moment, notice the parentheses and the quotation marks.

A word about the programs we will use to create and run Python 3 code.

In class, we will use **Jupyter** most of the time to run our programs within the notes. Jupyter is a program designed for presentations interspersed with code -- in other words, EXACTLY the type of thing you would want to make, say, Lecture notes in. There are "Markdown cells" which are meant to contain text, and there are "Code cells", which start with "In [number]:", and which you can run in Python by pressing Shift-Enter.

If you're not a teacher making notes for class, or you're not making a presentation, you may not want to use Jupyter. So, when you create your own programs outside of the notes, I suggest that you write and run them within **Spyder**. Spyder makes it easier to write and run longer bits of code, especially if your program occupies several files. So, let's go to Spyder, to write and run this program from there.

[This is the part where you open up Spyder.]

4. A Non-Trivial Program, with Variables

Most of what we will do with our programs is manipulation of data: reading it in, calculating with it, manipulating it, spitting it out. Even Tetris -- the graphics are numerical data output in a particular way, the pieces are represented by numbers, the keystrokes are represented numerically. So, let's start with the basics of data and its storage using *variables*.

Look at the code below, which contains some variables. We'll explain how it works in a few minutes, but for now, try to predict what it will do when run.

```
a = 10
b = 20
hey_yall_heres_the_sum = a + b
message = "What's going to print on this line?"
print ("The sum of a and b is:")
print (hey_yall_heres_the_sum)
a = 40
print (message, a + b)
```

Once you've got an idea (hint: the lines are executed in order -- pay attention to that!), cut and paste that into the cell below, and press Shift-Enter.

```
In [ ]: # EXAMPLE 4a: Addition
        # Introducing: variables (and data types)


        a = 10
        b = 20
        hey_yall_heres_the_sum = a + b
        message = "What's going to print on this line?"
        print ("The sum of a and b is:")
        print (hey_yall_heres_the_sum)
        a = 40
        print (message, a + b)
```

This program has 10 lines (not counting blank lines, which always get ignored). The first 2, which start with #, are called **comments**. The Python interpreter ignores them -- but you should always document what your code is doing! And I require them on programming assignments.

The next lines are executed in order. The first four of these lines create **variables** named `a` and `b` and `hey_yall_heres_the_sum`, and `message`, and **assign** them values. The first two assignments are straightforward: `a` holds the value 10 and `b` holds the value 20. The third assignment assigns looks at the values given to `a` and `b`, adds them, and gives this value to the variable `hey_yall_heres_the_sum`.

The fourth assignment is another straightforward one, but notice that it assigns a different type of value -- a "word", or as we'll come to call it, a **string** (as in "string of characters"). We'll pay a lot of attention to **data types** later on: is our value an integer? a real number (or **float**)? a string? a true-false value (or **bool**)?

Then come the print statements. Notice that in the first one, we print a **string literal** (that means something in quotation marks), which prints to the screen exactly as typed. The other print statements print *variables* (not in quotation marks), which causes the variables' respective *values* to be printed out. In the last print statement, two variables are in the print statement, separated by a common; these will get printed on the same line, separated by a single space.

Finally, notice that `a = 40` line before the last print statement. Initially `a` had the value 10: and indeed, any reference to `a` above that line produced the value 10. But then we encounter `a = 40`, and reference to `a` thereafter produce the value 40. So when we print `a+b`, we get the value `60`.

Does this seem really easy? It's as easy as it sounds, and at the same time, it's totally not as easy as it sounds, so be careful as we progress.

Oh, and you might be saying "gee, that seems like a useless program." If so, that's probably because it is a useless program. Many of the programs I show you, especially for the first few lectures, are mere demonstrations of language features. The demonstration here is: "variables store (and remember) data."

5. Values, Literals, Data Types

Having shown you a little bit about variables, I'd like to rewind, and carefully discuss the very basic elements of the language, so that we can learn how to predict what the Python interpreter will do when it encounters any bit of code.

All the data that a program will use consists of ***values*** of various types: numerical values, text values, logical values -- and down the line we will see fancier types of values.

A ***literal*** is a value that is written directly into your code. In Python, there are four basic types of literals (there's actually a bunch more, but you will rarely need them is this class). The first two are numeric: any run of digits that contains no decimal point is called an `int` (short for *integer*), and any run of digits that does contain a decimal point is called a `float` (short for *floating-point number*, but you should probably think of them just as "decimal numbers").

In [ ]:

```
# EXAMPLE 5a: Ints and floats

print(32)          # In this line, 32 is an int literal
x = -4             # In this line, -4 is an int literal
print(x)           # No literals in this line
6298341            # This line is just an int literal. It's kind of a dumb line,
                   # because you don't DO anything with the value.

print(17.34)       # In this line, 17.34 is a float literal
print(50,000,000)  # BEWARE -- no commas are allowed in int literals!
print(50000000)    # Do it like this instead
```

A third type of literal is the `str` (short for *string*, as in "string of characters"). A string literal is any run of typed characters, surrounded by quotation marks.

A fourth type of literal is the `bool` (short for *boolean*). There are exactly two `bool` literals: `True` and `False`, typed exactly that way, without quotations. They'll be useful to use in a few lectures, because they are useful for constructing logical expressions.

In [ ]:
```
# EXAMPLE 5b: Ints and floats

print("ABC")                              # "ABC" is a str literal
"Stupid line ABC-123!"                    # A longer str literal
print("Red leather", "Yellow leather")  # Two str literals inside a print

y = True                                  # True is a bool literal
print(y, False)                           # False is a bool literal
```

Every value in Python has a data type. A ***data type*** is essentially just a category of values; we've just met four of them. Data types help Python figure out what operations can be done with different values. For example, look at the following.

In [ ]:
```
# EXAMPLE 5c: Type Error

x = 3 + "Hello"
# You'll never make it to the next line, because you can't add
# an int and a str.

print(x)
```

If you ever need to confirm the type of a value, the `type()` function can help you.

In [ ]:
```
# EXAMPLE 5d: Guess the types

print(type(-4), type(-4.00), type(-4.) )

print(type(True), type("False"))
print(type("12"))
print(type(hello world))
```

The last line doesn't contain a valid value, since `hello world` isn't in quotes! You can fix this program by just commenting out the last line (that means putting a # in front of the line).

6. Variables and Expressions (and Printing)

A ***variable*** is a *name* that is associated with a *value*. (Warning: this is actually a simplification. But it wouldn't be helpful to describe the truth right now.)

The name part of a variable is called the variable's ***identifier***. Rules for Python identifiers:

- Only letters, digits, and underscores (_), no spaces or other punctuation
- May not begin with a digit
- May not be a reserved word (`if`, `import`, `def`, etc.)

So, these are legal: `hey123`, `AHHH`, `_x_y_z`

And these are not: `1st_thing`, `what?`, `no spaces`

Also:

- Identifiers are case-sensitive: `hello` is not the same as `hElLo`
- Probably should be *meaningful*
- We'll start variable names with lowercase letters usually

The purpose of variables is to hold data values for reference later on. This "obvious" point takes a while to click for most beginners. Variables aren't just things you throw in your program because whatever; their purpose is to remember data that you'll need later on.

To *create* a variable: simply write a line assigning it a value, of the following form:

In [ ]:
```
VARIABLE CREATION SYNTAX (do not run me):

<variable name> = <value>
```

After that, the variable name will be associated with that value, until it is assigned a different value.

(Note: In these notes, I will use `<this notation>` to represent a blank to be filled in. I use this to highlight ***syntax***: this refers to the order in which variables and operators are placed to make meaningful statements. For example, `x = 17` and `17 = x` are very different statements -- the latter causes a *syntax error*, because that's not the order you're supposed to write things in -- we'll discuss why soon.)

An ***expression*** is one or more literals or variables, connected with **operators** (e.g., `+`, `-`, etc.) and functions, which can be ***evaluated*** to produce a value. For example:

```
3 + 4.5
"Hello" + "Goodbye"
len("Hello")
"Hello" * 3
```

When these are encountered in a program, the interpreter respectively computes values of `7.5`, `"HelloGoodbye"`, `5`, and `"HelloHelloHello"` for the first four. (Notice that "value" doesn't mean *number* -- it means any type of data, and so strings are perfectly good end values.)

These are ***not*** (just) expressions:

```
x = 5 + 7
print(5+7)
```

Sure, `5+7` on its own is an expression -- it has a value. But `x = 5 + 7` and `print(5+7)` are really *instructions* about what to do with values, rather that *values* themselves.

Also, `3 + xyz` is also a (probably) valid expression. Since the `xyz` is **not in quotes, it is treated as a variable!**

To evaluate this expression, the interpreter reads the *current* value of the variable `xyz`, and (assuming it's numerical) produces that value plus 3. But: if

- `xyz` has a non-numeric value, or
- `xyz` has not previously been assigned a value

then this will cause an error!

Let's talk about the very basics of printing, which you may have caught on to.

```
In [ ]: SYNTAX FOR THE PRINT FUNCTION:

        print(<whatever>, <another>, <yet another>, <as many of these as you like>)
```

Each of those arguments (`<whatever>`, `<another>`, etc.) must be a single expression -- either a single literal, or a single variable, or a more complex formula. When your program reaches the `print` line, each of the expressions will be printed out on the same line, with their current values at that line, in order, with spaces between each value. The next `print` statement will be printed starting on a new line.

```
In [ ]: # EXAMPLE 6a: Basic print

        variable_x = 4 + 5
        print("Hello", True, variable_x, 5*variable_x  - (variable_x+2), "Goodbye")
        print("Next line")
        # That looks like two expressions to me, so there's got to be a comma between them!
        # If you put a comma in between, then this code will run.
        print("Here's a problem" variable_x)
```

You can also suppress the automatic newline at the end of a print by including `end = ""` at the end of the print statement, as follows.

```
In [ ]: # EXAMPLE 6b: Suppressing the automatic newline

        print(10)
        print(11)
        print(12, end = "") # This print statement won't be followed by an automatic newlin
        e...
        print(13)           # ...which means that this is displayed IMMEDIATELY after the 1
        2, not even a space in between.
        print(14)
```