Communion: Python Lesson One

These lessons are meant to give you an introduction to Python, not to teach you everything there is about it. Please do your own exploration if you're curious. This can be done through looking at the official Python tutorials, StackOverflow (a community driven QA site for coding), or just through experimentation.

Setting Up Shop

1. In order to use Python, you need to install it. You can get it here. Grab the newest version (at the time of writing is 3.9.4) as I may discuss old ways of doing things, but the newer ways are typically better in terms of security, readability, or otherwise. This installation gives you all of the standard libraries as well as core code to build/run Python scripts.

2. To write Python scripts, any text editing software can be used, including Notepad and Word, if you want; however, these solutions lack intelligent support/functionality for developers. Instead of using one of these *wonderful* solutions, install Visual Studio Code. Visual Studio Code (VS Code for short) is an open source integrated development environment (IDE) that has built in functionality for coding, such as syntax highlighting for different coding languages, and community developed extensions which further enhance every aspect of the environment. There are plenty of rudimentary how-tos on the internet to get started with Visual Stuido Code and Python, so I'll leave it to you to find and follow one of those.

Anatomy of Hello World

The most basic program in any language is `Hello World`. It's essentially just a print statement to your console to show that you have everything set up correctly and that you can get your machine to run the simplest of programs. In Python, it looks like this:

`Python`:

```
print('Hello World')
```

Comparatively, here are a few examples of `Hello World` in a few languages other languages:

1. `C` and `C++`:

```
#include <iostream>

int main(int argc, char *argv[]) {
    printf("Hello World");
    return 0;
}
```

`HTML`:

```
<body>
<p>Hello World</p>
</body>
```

`Malbolge` (my personal favorite as it's just so crazy that it took years to discover `Hello World`):

```
(=<`#9]~6ZY327Uv4-QsqpMn&+Ij"'E%e{Ab~w=_:]Kw%o44Uqp0/Q?xNvL:`H%c#DD2^WV>gY;dts76qKJImZkj
```

Python's implementation may seem trivial, and you would be right. Python was initially birthed as a scripting language (programming language geared toward quick development and simple use-cases), but has evolved into something that can handle substantially more complex structures/tasks.

The print Statement

The print statement is probably the most widely used debugging/feedback tool used in any programming language. In Python, it's simply `print()` with whatever you want to print within the parentheses. The best example is `Hello World` shown above.

In Python, however, the functionality does not end there.

- Multiple values: It's possible to print multiple things with a single print statement.
    - example: `print('123', '456')` will print the strings `123` and `456` on the same line, separated by a space.
- Non-string items: The object passed (the item inside the parentheses), does not need to be a string to print. Instead, the object just needs to define how to be treated when converted to a string (more detail on this upon request and maybe in a later lesson).
    - A few objects that print fine include:
        - string
        - integer
        - float
        - list
        - set
        - dictionary
- Additional arguments: In addition to the things that you want printed, there are some useful arguments that you can pass to the print method to modify how the printout is formatted.
    - `sep=<str>`: changes the separator between printed items when multiple are passed. The default value is a single space.
        - example: `print('123', '456', sep=' --> ')` will print out `123 --> 456`
        - example: `print('1', '2', '3', '4', sep=' & ')` will print out `1 & 2 & 3 & 4`
    - `end=<str>`: changes the final character(s) of the print statement. The default value is a new line character (i.e. the next print statement will be on the next line)
        - example: `print('hi there', end='!\n')` will print out `hi there!` followed by a new line.
        - example: `print('hi there', end='!')` will print out `hi there!` without a new line. This means that the next print statement will output abutted to this one.

Variables

A Variable in Python is a storage container for information that the user may want to use/manipulate.

A valid variable name may contain the following in any permutation:

- `[a-z]`: lower-case letters
- `[A-Z]`: upper-case letters
- `[0-9]`: any number
- `_`: underscore

Variables in Python, unlike other languages such as C and Java, do not need to be declared as a given type. Instead, the Python interpreter guesses (very well, but not perfectly in some cases) at runtime what the variable type is.

To define a variable, simply type the desired variable name followed by the assignment operator `=` (operators will be discussed in the next section) and the initial value. The interpreter guesses the type based on this initial value.

For example: `foo = 123` assignes the integer value `123` to a variable named `foo`. Spaces in cases like this are ignored and thus `foo = 123` and `foo=123` are equivalent, but it is considered good practice to use spaces in this regard since it promotes readability.

In addition to the variable name and use, there are many different variable types, but only a few key ones are discussed here:

- Integer:
    - As the name implies, integer values.
    - Can't really be "empty." Instead initialize with `0` if no specific value needed.
    - example: `myInteger = 123`
- Float:
    - Decimal numbers.
    - Can't really be "empty." Instead initialize with `0` if no specific value needed.
    - example: `myFloat = 1.23` and `myFloat = 1.0`
- String:
    - A list of characters that form "words". This is actually a pretty terrible way to describe it, but it's *technically* correct, I suppose.
    - Strings can be defined with single or double quotes. Note that the quotes can be used as part of the string if the encapsulating quotes are the other or if you use the escape character (discussed in the **other** section). Empty string denoted as `''` or `""`.
    - examples:
        - `myString = 'hi there'`
        - `myString = "hi there"`
        - `myString = 'a string "with quotes"'`
    - Strings can also be manipulated with a number of built in functions.
        - `<string>.upper()` and `<string>.lower()` converts the string (can be string literal or variable) to uppercase and lowercase, respectively.
            - example: `bar = foo.upper()` - Converts `foo` to all uppercase (where appropriate; integers ignored) and stores the result in `bar`.
        - `<string>.replace(<find_value>, <replace_value>)` finds a substring and replaces it with a replacement string in the given string (can be string literal or variable).
            - example: `bar = foo.replace(' ', '_')` - Finds all spaces in `foo` and replaces them with underscores and stores the result in `bar`.
        - f-strings or formatted strings: these are a way to define a string with variables within the string itself. I believe it's best explained through examples.
            - example: `foobar = f'hi, my name is {myName} and i am {myAge} years old.'` - This generates a string that inputs the predefined variables `myName` and `myAge` in the designated places

and stores the resultant string in `foobar`.
        - example: `foobar = 'hi, my name is {} and i am {} years old.'.format(myName, myAge)` - This is effectively the same as the above example but has all the variables stated in the `.format` at the end. Variables are added to the `{}` in respective order. This is the older style and has backward compatibility with `Python3 3.6.x` (when the above example was introduced) and before, I believe. Personally, it's not as readable and I shy away from it wherever possible.
    - `<string>.ljust(<num_chars>, <fill_char>)`, `<string>.rjust(<num_chars>, <fill_char>)`, and `<string>.center(<num_chars>, <fill_char>)` creates a string with a fixed width by justifying the given string or variable (left, right, and center respectively) and filling the remaining space with the given filler character.
- List:
    - A single variable name container that holds many items.
    - The items do not need to be the same type.
    - Defined using square brackets.
    - Empty list denoted as `[]`.
    - examples:
        - `myList = ['a', 'b', 'c']`
        - `myList = ['a', 1, 2.3]`
    - Items can be individually accessed through `<var_name>[<index>]`. Indexing in Python starts at 0.
        - example using `foo = ['a', 'b', 'c']`
            - `bar = foo[0]` will assign `'a'` to `bar`
- Dictionary:
    - Similar to lists, but instead of just storing items, items are stored with a key.
    - Individual items can be accessed by indexing with the proper key.
    - Defined using squigly braces.
    - Empty dictionary denoted as `{}`
    - examples:
        - `myDict = {'a': 1, 'b': 2}` - This is a dictionary containing the values `1` and `2` with the keys `'a'` and `'b'`, respectively.
        - `myDict = {'somelist': [1, 2, 3], 'somestring': 'abcd', 'anotherdict': {'a': 1, 'b': 2}}`
    - These are quite useful for storing multiple lots of data in a structured format.
    - They also export/import to/from `JSON` files very cleanly. (JSON is a data exchange file type -- more reading)

Operators

- `=` - Assign
    - assign right value to left
    - example: `foo = 3` assigns integer 3 to variable `foo`
- `==` - Equivalent
    - returns `True` if both sides have the same value, otherwise `False`
    - example: `1 == (2/2)` returns `True`
    - example: `1 == 1.0` returns `True`
    - example: `1 == '1'` returns `False`
- `>` - Greater than
    - returns `True` if left side value greater than right
- `>=` - Greater than or equal to
    - returns `True` if left side value greater than or equal to right
- `<` - Less than
    - returns `True` if left side value less than right

- `<=` - Less than or equal to
  - returns `True` if left side value less than or equal to right
- `!=` - Not equal to
  - returns `True` if two values are not equal
- `+=` - Self add
  - left value must be variable
  - adds right value to variable and stores back to the variable
  - example: `foo += 1` increments foo by 1
  - example: if `foo == 'abc'` and `foo += 'd'` is called, then `foo == 'abcd'`
- `-=` - Self subtract
  - left value must be variable
  - subtracts right value from variable and stores back to the variable
  - cannot work with strings (unlike `+`)
  - example: `foo -= 1` decrements foo by 1
- `and` - Logical and
  - returns `True` if both values are logically equivalent to `True` (there is a bunch of nuance in this when dealing with non-boolean values)
- `or` - Logical or
  - returns `True` if either value are logically equivalent to `True`
- `not` - Logical not
  - returns the inverse boolean value of the logical equivalent of the following value
- `in` - Existence in
  - checks to see if left value is in right
    - example: `1 in [1, 2, 3, 4]` returns `True`
    - example: `'c' in 'abcde'` returns `True`
  - also used in `for` loops but the operation is slightly different
- `+`, `-`, `*`, `/`, `%` - Arithmetic operators
  - add, subtract, multiply, divide, modulo, respectively
  - with the exception of the add operator, these behave as expected
    - for those of you that don't know modulo, it performs the division of the two numbers and returns the remainder
      - example: `5 % 2` returns `1`
      - example: `12 % 5` returns `2`
  - the add operator is often overloaded (a fancy word meaning defined in multiple locations for different instances) to operate in fancy ways
    - as expected, addition can be performed on integers and floats
    - addition can be performed on strings to merge them
      - example: `'abcd' + 'foobar'` returns `'abcdfoobar'`
    - addition can be performed on lists to merge them
      - example: `[1, 2] + ['a', 'b']` returns `[1, 2, 'a', 'b']`

Conditionals

Conditional statements allow the user to make divergent code. Code that does not do this is laregely useless and is merely a checklist of tasks.

Conditionals rely on evaluating logical statements and proceeding based on the boolean results.

Their general construction is `if <logical statement>:`. The executed code-block, should the logical statement hold true, follows the `if` statement on separate indented lines. Note that Python is an indentation based language rather than a braces language (like

Java and C++). Anything with the same indentation level will be grouped together in the same code block.

Example:

```
if 1 == 2:
    print('well, we have a problem here')
```

In this example, the `1 == 2` is our logical statement which dictates whether or not the code will evaluate the code within the `if` block.

To perform code in the event that the if statement returns `False`, we use `elif` to specify another case to check, or `else` to catch all other cases.

```
if 1 == 2:
    print('well, we have a problem here')
elif 1 == 3:
    print('not sure if this is more concerning')
else:
    print('okay, math still holds')
    foobar = 'everything is okay'
```

Sometimes you may want to assign a value to a variable if a logical statement is true and another if it is false. With how we've learned already, it would look something like this:

```
if foo == True:
    foobar = 1
else:
    foobar = 2
```

While correct, this can be a bit clunky and cumbersome to look at. This can be concatenated into a single line using what's called a ternary operator.

```
foobar = 1 if foo == True else 2
```

This has the exact same operation as well as being easier to read (and keeping your code-base smaller).

Not to make this more confusing, but the ternary operator can be repeated endlessly in a single line for multiple conditions (much like the `elif`).

```
foobar = 1 if foo == True else 2 if bar == True else 3 ...
```

Loops

There are two main loops to consider that are used in essentially every language (that I can think of): `for` and `while`. Other languages have things like the `do/while` loop and other such variants, but they're essentially the same thing with minor caveats.

The `while` loop

- The while loop is perhaps the easier of the two to conceptualize, so we'll start here
- These are very useful for cases when you are unsure how many times the loop needs to occur, but a clear exit condition can be set.
- Syntax of the while loop is similar to the `if` statement (without `elif` and `else`) in that it's the word `while` followed by a logical statement.
- The main difference is that the code block after the `while` statement is executed endlessly until the logical statement no longer holds.
- It is easy to create endless loops this way should no code be introduced in the execution block to change the logical statement's outcome.

`while` examples:

```
while 1:
    print('i think i made a mistake')
```

This example will enter an infinite loop, printing `'i think i made a mistake'` forever since the logical statement `1` evaluates to true.

```
success = False
foo = 0
while not success:
    print(foo)
    foo += 1
    if foo >= 10:
        success = True
```

This example will print the numbers 0 through 9 before exiting.

The `for` loop

- The `for` loop is used when the number of iterations is known, either by the user, or based on some known parameter within the code.
- While this operation can be achieved with the `while` loop, using a `for` loop eliminates the need to explicity write code to break out of the loop.
- Their general construction is `for <iterating variable> in <some list>:`
    - While the right hand side of the `is` is not always a `list`, it suffices for beginners to think as much.
    - If you're interested, a few exceptions are `generator statements` and `iterators`.

`for` examples:

```
for foo in ['a', 'b', 'c']:
    print(foo)
```

Here a variable `foo` is taking each value from the right hand list individually and executing the code block with that value.

```
for foo in [1, 2, 3]:
    temporaryVariable = foo * 10
    print(temporaryVariable)
```

This example is largely the same as the previous one, but shows multiple lines as part of the execution block.

```
bar = ['a', 'b', 1, 2, {}]
for foo in bar:
    if type(foo) == str:
        print(foo.upper())
    elif type(foo) == int:
        print(foo * 10)
    else:
        print(f'not sure what to do with {foo} ({type(foo)})')
```

This is a sligtly more complex example, but all of the pieces have been discussed previously and you should be able to figure out what would happen upon running. If not, try it.