Lesson 1: Hello World

Python, a Programming Language

Python

Python is a programming language. Like other languages, it gives us a way to communicate ideas. In the case of a programming language, these ideas are “commands” that people use to communicate specifically with a computer.

We convey our commands to the computer by writing them in a text file using a programming language (**.py** extension). These files are called *programs* or *scripts*. Running a program means telling a computer to read the file, translate it to the set of operations that it understands, and perform those actions.

Interpreted Language

Python is an *interpreted language*, which means that each line of code you write is translated, or *interpreted,* into a set of instructions that your machine can understand by the Python interpreter. This stands in contrast to compiled languages. For these languages (Fortran, C, C++, etc.), your entire code is translated into machine language before you ever run it. When you execute your program, it is already in machine language.

So, whenever you want to actually run your Python code, you give it to the Python interpreter first.

There are many ways to launch the Python interpreter. One way is to type into a command terminal

```
python
```

This launches the basic Python interpreter (if you installed Python 3, this is what you'll do to use it). We won't really this in the bootcamp, but instead JupyterLite via browser or JupyterLab's console.

Hello world.

Traditionally, the first program anyone writes when learning a new language is called `Hello world.` In this program, the words “Hello world.” are printed on the screen. The original `Hello world.` was likely written by *Brian Kernighan*, one of the inventors of Unix, and the author of the classic and authoritative book on the C programming language.

We'll first write and run this little program using a JupyterLab console or JupyterLite in browser. After launching JupyterLab, you probably already have the Launcher in your JupyterLab window. If not, you can expand the Files tab at the left of your JupyterLab window (if it is not already expanded) by clicking on that tab, or alternatively hit `ctrl+b` (or `cmd+b` on MacOS). At the top of the Files tab is a `+` sign, which gives you a Jupyter Launcher.

Once in the Jupyter Launcher, click the Python 3 icon under Console. This will launch a console, which has a large white space above a prompt that reads `In []:`. This is called a *code block* or a *cell* in JupyterLabs. You

can enter Python code into this block, and it will be executed.

To print `Hello world.`, enter the code

`print('Hello, world.')`

below in the code box (usually denoted as a grayed-out or gray-outlined box of text). To execute the code, hit `shift+enter` or click the "play" triangle symbol located along the bar at the top of the editor.

In [ ]:
```
print('Hello, world.')
```

Notice the additional text below the line of code. This is called an *output* and is the result of executed code. If this extra text reads "Hello, world.", congratulations! You did it! Well done.

The `print()` used in the code line above is called a **function**, and it is pre-built into Python. Words inside single quotations are outputted exactly as written from the `print()` function.

Preliminaries

Before we let you run off into the sunset with your firm understanding of `print()`, there's a few more things we have to cover. Then you can write your very own script.

Comments

Sometimes you may want to leave a note for yourself (or future programmers) within your code. Maybe you need to remember to add a line later on, or maybe what you're doing is complicated and needs a human-readable explanantion. Either way, what you need to use is a line of text that the machine won't interpret and try to execute: a **comment**.

> A **comment** is a piece of text within a program that is not executed. It can be used to provide additional information to aid in understanding the code. The `#` character is used to start a comment and it continues until the end of the line.
>
> Comments can be in line after/before code or on their own line.

In [ ]:
```
print('This is an in-line comment.') # This is just a print function
```

In [ ]:
```
# This is just a print function
print('This is a separate comment line.')
```

Variables

Whether you are programming in Python or pretty much any other language, you will be working with **variables**. While the precise definition of a variable will vary from language to language, the focus here is on Python variables. However, like many of the concepts in this bootcamp, the knowledge you gain about Python variables will translate to other languages fairly well, if not one-to-one.

A **variable** is used to store data that will be used by the program or script. This data can be a number, a string, a Boolean, a list or some other data type (there's lots of data types, which we'll review later). Every variable needs a name assigned to it to keep track ot it. This name can consist of letters, numbers, and the underscore

character `_`. The equal sign `=` is used to assign a value to a variable. After the initial assignment is made, the value of a variable can be updated to new values as needed.

> **Note:** As an aside, in Python variables are **objects**. In fact, everything in Python is an object. But more on that later.

So, a variable has two properties: a *type* and a *value*.

In [2]:
```
age = 19
```

In the example above, the variable `age` has been assigned or *instantiated* (initially assigned) to the value `19`, which is the data type 'integer'. To do this, we used the `=` character as an *assignment operator*.

For all intents and purposes, `age` is the exact same thing as `19` now. If you're not sure what type of variable you're working with, use the `type()` function in Python.

In [1]:
```
type(4)
```

Out[1]:
```
int
```

In [2]:
```
type(-6.79)
```

Out[2]:
```
float
```

In [3]:
```
type("Hello world.")
```

Out[3]:
```
str
```

For the above examples, `type()` tells us that `4` is an `int` (integer number), `-6.79` is a `float` (short for "floating point number", which basically translates to "a real number that isn't an integer"), and `Hello world.` is a `str` (short for "string").

> **Note:** If you want a number to appear as a string, all you need to do is put it inside a pair of quotations.

In [5]:
```
type('-4')
```

Out[5]:
```
str
```

> **Note:** Scientific notation also works for floats (floating point numbers).

In [9]:
```
type(-6.79e6)
```

```
-6790000.0
```

Out[9]:
```
float
```

And finally, if you needed it for, say, organization of data, you can extract the type from a variable using `type()` and output it somewhere using `print()`.

In [20]:
```
num = -6.79e6
type_num = type(num)
print('The variable value %r has data type %r' %(num, type_num))
```

```
The variable value -6790000.0 has data type <class 'float'>
```

Notice how variables values and types were inputted into the sentence above. `%r` is the general purpose inputter or placeholder, useful for if we don't know the type of data we're handling (like the above example). But there are also more specific placeholders:

- `%d` is for numbers (integers & floats)
- `%s` is for strings
- `%r` is for unknown (or changing) type variables

Integers & Floats

As an additional note to arithmetic operators, let's talk briefly about integers. An **integer** is a number that can be written without a fractional part (no decimal). An integer can be a positive number, a negative number or the number `0` so long as there is no decimal portion.

The number `0` represents an integer value but the same number written as `0.0` would represent a **floating point number** or **floats**. Even though we can agree zero is zero, these are different representations of zero, and both are useful depending upon what we want to do.

Arithmetic Operations

Python supports a variety of arithmetic operations. These are the primary arithmetic operations you may remember from grade school:

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `%` for modulus (returns the remainder)
- `**` for exponentiation
- `//` floor division

And like the regular ol' arithmetic operations from grade school, they can be performed on actual numbers `1`, `17`, `-39`, variables `x`, `age`, `solution`, or a combination of both.

> **Note:** `%` as an arithmetic operation is strictly different from when `%` is used inside the `print()` function to denote variable value placement within an outputted string.

In [ ]:
```
# my sibling was born in 2000, and I am 3 years older than them
sibling_age = 2023 - 2000
my_age = sibling_age + 3
```

> **Warning:** Do not use the `^` operator to raise something to a power. In Python, that symbol is reserved as the operator for bitwise `XOR`, which is outside the scope of this bootcamp.

Arithmetic Operators on `Int`

In [23]:
```
2*3
```

```
Out[23]: 6

In [24]: 6/3

Out[24]: 2.0

In [25]: 2**3

Out[25]: 8

In [26]: 8//6

Out[26]: 1
```

> **Note:** Keep in mind how, despite utilizing two integers, the result of line 24, `2.0`, is a `float`.

Arithmetic Operators on `Float`

```
In [32]: 2.1+3.2

Out[32]: 5.300000000000001

In [31]: 6.2/2.1

Out[31]: 2.9523809523809526

In [29]: 1.9*3.3

Out[29]: 6.27
```

> **Note:** Notice the result of line 32. We know that `2.1+3.2=5.3`, however a machine doesn't have that awareness to fall back on. Instead, this is an instance of *floating point error* or, more colloquially *rounding error*.

Floating Point Error

How many digits of `pi` can you hold in your head? You know, that number...

`pi = 3.141592657...`

Well, just like you, computers can only hold so much information at any one moment. And, for arithmetic operations, any inaccuracy in this held information (call it the machine's *working memory*) results in an error in calculated floating point numbers, denoted *floating point error* or *rounding error*. We won't get into this too much during this bootcamp, but it's good to keep in mind.

Order of operations

The order of operations is what you should be familar with from introducatory mathematics: Exponentiation comes first, followed by multiplication & division, floor division, and modulo. Last, as always, ss addition and subtraction. In order of precedence, this is

| **precedence** | **operator** | |:----------:|:----------:| | 1st | `**` | | 2nd | `*`, `/`, `//`, `%` | | 3rd | `+`, `-` | | |

Strings

And finally, let's take a look at strings. A **string** is a data type composed of a sequence of characters (letters, numbers, whitespace or punctuation) enclosed by quotation marks, can be either double quotation mark `"` or single quotation mark `'`.

If a string has to be broken into multiple lines, triple quotation marks `"""..."""` around the sentence, then hit enter at appropriate line breaks as normal to indicate to the machine that the string continues on the next line.

In [37]:
```
# examples of string formatting
user = "Full Name Here"
life_stage = 'young adult'

sentence = """This string is a lot longer, so it must
be broken several times to be
more easily read
on a small screen."""

print(sentence)
```

```
This string is a lot longer, so it must
be broken several times to be
more easily read
on a small screen.
```

> **Note:** You can use `print()` in combination with a string by excluding any quotations within `print()` that would normally be present.

Please proceed to Assignment 1.
