Python Bootcamp 1

This is a Jupyter Notebook. To run a gray code cell, click in the cell and either click on the "Run" arrow, or type shift+enter (or shift+return on a Mac).

```
print("Hello World!") #it's a print func
```

```
Hello World!
```

it's a print func

When you type in the gray code cell and hit shift+enter, you **run** the code. If something gets printed out to the screen after you run the code, we say it is **returned**.

Text cells like this one don't contain code. They are written in a language called Markdown. If you accidentally double click in a markdown cell, it won't cause any problems. Go ahead and double click on this text. Then you can run the cell, just like you would with a code cell, to return it to looking like nice formatted text. You can also type **bold**, *italic*, or underscored text.

Arithmetic operators

```
4 + 2
```

```
6
```

```
6 - 1
```

```
5
```

```
100 * 35
```

```
3500
```

```
40 / 6
```

```
6.666666666666667
```

To calculate an exponetial:

```
4 ** 2
```

```
16
```

```
3 ** 5
```

```
243
```

You might notice that I've included spaces around the operators: `4 + 2` instead of `4+2` This is best practice in Python because it is more readable. The math would work without spaces, but it's a good idea to get into the habit of writing "Pythonic" code now!

More division operators:

```
40 / 6
```

```
6.666666666666667
```

We gave it **integers** (whole number objects) and it returned a **float**. The `/` division operator always returns a float.

Get just the integer, without the remainder:

```
40 // 6
```

```
6
```

Get just the remainder:

```
40 % 6 # 40 mod 6
```

```
4
```

```
50.4 % 4
```

```
2.3999999999999986
```

Like many things you'll learn in this course, I'm not expecting you to memorize the operator for returning a remainder. If this is something you use a lot already in your own coursework, you might memorize it now. But if not, I just want you to hear about it so that if you ever need it months from now, you might remember that there is a way to do it so that you can google it and find the right code.

More practice:

```
40 / 5
```

```
8.0
```

```
40 // 5
```

```
8
```

```
40 % 5
```

```
0
```

```
30 // 4
```

```
7
```

Python follows the PEMDAS (Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right)) order of operations: (https://thehelloworldprogram.com/python/python-operators-order-precedence/)

```
40 / 5 + 2
```

```
10.0
```

```
40 / (5 + 2)
```

```
5.714285714285714
```

```
40 + 2 * 6 - 1
```

```
51
```

```
(4 + 2) * (6 - 1)
```

```
30
```

```
2.718281 ** 2 * 7 - 5
```

```
46.723361164727
```

```
a = 2
print("My sis is " , a)
#a =* (a + a)
#print(a)
```

```
My sis is  2
```

```
#%pip install numpy
import numpy as np # this library is needed for exp and log
np.exp(2) * (1 + 6) - 5
np.log(2.718)
```

```
np.float64(0.999896315728952)
```

```
2.718281
```

```
(np.log(5) - 3) / 2 + 1
```

```
0.30471895621705014
```

Why did some lines of code return integers (30, 51) and one returned a float (10.0)? The first line of code included division `/`, which you may remember always returns a float.

Try out some math of your own:

```
```

Comparison operators

These return a **boolean** object: True or False.

```
3 > 2
```

```
True
```

```
3 < 2
```

```
False
```

```
3 >= 2
```

```
True
```

```
3 <= 2
```

```
False
```

To check if two things are **equal**, Python uses a double equals sign:

```
3 == 3
```

```
True
```

```
# No space between = and =
3 = = 3
```

```
  File
"C:\Users\Vincent\AppData\Local\Temp/ipykernel_6632/1671315643.py",
line 2
    3 = = 3
        ^
SyntaxError: invalid syntax
```

To check if two variables are **not equal**, Python uses `!=`:

```
3 != 3
```

```
False
```

```
3 != 2
```

```
True
```

```
# No space between ! and =
3 ! = 2
```

```
  File "C:\Users\Vincent\AppData\Local\Temp\
ipykernel_7728\805721107.py", line 2
    3 ! = 2
      ^
SyntaxError: invalid syntax
```

Logical operators

These also return a **boolean**.

`and` means both sides must be True to return True:

```
3 < 4 and 4 == 3
```

```
False
```

`or` means only one side must be True to return True:

```
4 < 3 or 4 == 4
```

```
True
```

`not` is another logical operator. It reverses the boolean that follows it:

```
not 2 == 2
```

```
False
```

```
not 3 == 2
```

```
True
```

Before running these cells, try to guess if they will return True or False:

```
not 2 > 3 and not 2 == 2
```

```
False
```

```
not 2 > 3 or not 2 == 2
```

```
True
```

```
3 == 3 and not 3 != 3
```

```
True
```

String objects

A **string** is an object kind of like text. It is contained in double `"` or single `'` quotes (you can choose which you want to use).

```
"Hello World"
```

```
'Hello World'
```

Let's use our first function! It can be used on almost all classes of objects:

```
print("Hello World")
```

Strings can contain digits:

```
print("Hello World 2")
```

```
Hello World 2
```

Strings can contain **special characters**. The most commonly used are new lines `\n`:

```
print("Hello World\nHow are you?")
```

```
Hello World
How are you?
```

...and tabs `\t`:

```
print("Hello World\tHow are you?")
```

```
Hello World	How are you?
```

If you simply **run** the string, it will return the string how it looks to the computer:

```
"Hello World\nHow are you?"
```

```
'Hello World\nHow are you?'
```

...but if you use the `print()` function, it interprets the special characters and returns the string like this:

```
print("Hello World\nHow are you?\n123\n456\taaa")
```

```
Hello World
How are you?
123
456	aaa
```

What happens if you forget the quotes around a string?

```
print("Hello World")
```

```
  File
"C:\Users\Vincent\AppData\Local\Temp/ipykernel_6632/4293340409.py",
line 1
    print(Hello World)
                  ^
SyntaxError: invalid syntax
```

Our first error!! This is a common error - the `SyntaxError`. It often means that we forgot to type something.

```
pint("Hello World!")
```

```
---------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
Cell In[6], line 1
----> 1 pint("Hello World!")

NameError: name 'pint' is not defined
```

A `NameError` is another common error - it often means we've mispelled a function or variable.

What error do you think we will get if we run this line of code?

```
print("Hello World)
```

```
  File
"C:\Users\Vincent\AppData\Local\Temp/ipykernel_6632/2771583741.py",
line 1
    print("Hello World)
                       ^
SyntaxError: EOL while scanning string literal
```

Some arithmetic operators work with strings. To combine two strings:

```
print("Hello" + "World")
```

```
HelloWorld
```

```
print("Hello " + "World")
```

```
Hello World
```

```
print("Hello " + "World" + "!")
```

```
Hello World!
```

```
print("Hello World "*4)
```

```
Hello World Hello World Hello World Hello World
```

```
print("Hello World\n"*4)
```

```
Hello World
Hello World
Hello World
Hello World
```

Converting between data types

The `type()` function tells us the data type of an object (in Python, a data type is exactly the same thing as an object class):

```
type("Hello World")
```

```
str
```

```
type(9)
```

```
int
```

```
type(9.9)
```

```
float
```

```
type("9.9")
```

```
str
```

We can convert objects from one type to another, using the appropriate function:

```
float(9)
```

```
9.0
```

```
float("5")
```

```
5.0
```

```
int("5")
```

```
5
```

```
int(5.99)
```

```
5
```

Notice it doesn't round when it converts a float to an integer, it just drops off everything after the decimal.

```
str(5.005)
```

```
'5.005'
```

```
float("Hello World")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6632/49509932.py in <module>
----> 1 float("Hello World")

ValueError: could not convert string to float: 'Hello World'
```

Our third type of error! `ValueError` usually means you passed an argument to a function that it doesn't recognize. But don't spend any brainpower trying to memorize this error. Luckily the error message gives us information about what we did wrong.

Data types interacting

Before running each of these cells, try to guess what they will return:

```
4 * 4.0
```

```
16.0
```

```
"Hello World" * 4
```

```
'Hello WorldHello WorldHello WorldHello World'
```

```
"Hello World" + "4"
```

```
'Hello World4'
```

```
"Hello World" * 4
```

```
'Hello WorldHello WorldHello WorldHello World'
```

A `TypeError` means you tried to do something that doesn't work with the type of object you gave it. Sometimes `ValueErrors` and `TypeErrors` get thrown for similar errors.

```
"Hello World" + str(4)
```

```
"Hello World" + "4"
```

```
5 + True
```

```
6
```

```
5 + False
```

```
5
```

True evaluates to 1, False evaluates to 0

```
"5" + True
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6632/3803658572.py in <module>
----> 1 "5" + True

TypeError: can only concatenate str (not "bool") to str
```

Let's learn some new funcitons that work with integers and floats.

The `abs()` function returns the absolute value of a number:

```
abs(-19.6)
```

```
19.6
```

The `round()` function will round a float to a specified number of decimal places. It takes two arguments: the number to round and the number of decimal places.

```
my_number = 631.890382720
```

```
round(my_number, -2)
```

```
600.0
```

```
round(my_number, 2)
```

```
631.89
```

Some function arguments have default values, so if you don't include the argument, it will use the default value. Let's try `round()` without specifying a value for the second argument:

```
round(my_number)
```

```
632
```

```
int(my_number)
```

```
631
```

So the default value is to round to the closest integer.

`round()` can also round to places to the left of the decimal point:

```
round(18192948, -3)
```

```
18193000
```

You can also put functions inside other functions:

```
round(abs(-12.908))
```

```
abs(round(-12.908))
```

```
round(30 + 698, -1)
```

Variables

Variables allow you to store an object to use it later.

```
x = 3
```

Nothing is returned when you assign a variable. You can see the value of a variable by running the variable name or using the `print()` function

```
x
```

```
3
```

```
print(x)
```

```
3
```

The single `=` is called an **assignment operator**. Everything on the right side of the assignment operator is calculated first, and then that value is stored as the variable on the left of the assignment operator.

```
x = 3 - 1
```

```
x
```

```
2
```

```
y = x
```

```
y
```

```
2
```

```
z = 'aaa'
z
'aaa'
x = x**y
x
4
y
2
rabbit = 50
```

Variables can be used just like the objects they represent:

```
rabbit - 20
30
rabbit
50
greeting = "Hello World!"
greeting
'Hello World!'
print(greeting)
Hello World!
```

BACK TO THE SLIDES