Python is interpreted

Which means that the statements we type are interpreted and executed immediately.

....or in other words, we can use it as a calculator!

Given that the distance between Edinburgh and London is 403 miles, can we convert that to km?

In [ ]:

```
403 * 1.60934
```

Typing in numbers gets tedious quickly, how can we get around that?

Variables

are human-readable containers that can store numbers, characters, strings or any other value.

Let's try to calculate the distance between Edinburgh and London in km again:

In [ ]:

```
distanceToLondonMiles = 403
mileToKm = 1.60934
distanceToLondonKm = distanceToLondonMiles * mileToKm
distanceToLondonKm
```

That's neat! Now we can reuse the variable `mileToKm` for other conversions without having to type it in again!

Can we do the same conversion for a marathon, while keeping some of the variables above?

In [ ]:

```
marathonDistanceMiles = 26.219
marathonDistanceKm = marathonDistanceMiles * mileToKm
print(marathonDistanceKm)
```

Note that you can't add whitespaces to variable names, which means that we need combine multiple words into one. Possiblities:

```
allbundledtogether = "not readable"
CamelCase = "readable"
under_score = "readable"
```

Types

The values in these containers are stored as particular types. The important ones to remember are:

Type	Declaration	Example	Usage
Integer	int	`x = 124`	Numbers without decimal point
Float	float	`x = 124.56`	Numbers with decimcal point
String	str	`x = "Hello world"`	Used for text
Boolean	bool	`x = True` or `x = False`	Used for conditional statements
NoneType	None	`x = None`	Whenever you want an empty variable

How does all of that actually affect us? Well, let's try to add an Integer and a String:

In [ ]:

```
x = 10    # This is an integer
y = "20"  # This is a string
x + y
```

Whoops! Seems like we managed to break Python.

Luckily, Python offers an easy way to convert one variable to another. Let's try our example again.

In [ ]:

```
x = 10    # This is an integer
y = "20"  # This is a string
x + int(y)
```

We just converted the String variable y to an Integer! Actually, we can convert any type of variable to another type in a similar manner! For example, we can convert `y` back to a String with `str(y)`.

If you are ever unsure of what type a variable you can use the `type()` function to find out.

In [ ]:

```
type(mileToKm)
```

What if we make x and y Strings and add them up?

In [ ]:

```
x = "10"
y = "20"
x + y
```

That doesn't seem arithmetically correct but is correct in the world of Python. We effectively concatinated 2 strings!

Notice how the `+` operator behaves differently depending on the type of the variables. This is an imporant principle which spans across the whole Python language!

Exercise

1. **Create a variable `x` and assign the value `4` to it.**
2. **Create another variable `y` and assign to it the value of `x`**
3. **Change the value of `x` to be `"four"`**

Confirm your results by typing `x,y` at the end of the code cell. This will print out the values of the 2 variables.

In [ ]:

```

```

Arithmetical operators

Now let's see how we can make full use of our new calculator. Python supports a range of different arithmetical operators:

Symbol	Task Performed	Example	Result
+	Addition	4 + 3	7
-	Subtraction	4 - 3	1
/	Division	7 / 2	3.5
%	Mod	7 % 2	1

Symbol	Task Performed	Example	Result
%	Mod	7 % 2	1
*	Multiplication	4 * 3	12
//	Floor division	7 // 2	3
**	Power of	7 ** 2	49

Make note of how the different division operators work.

In [ ]:

```
16 ** 2 / 4
```

What happens if we have a lot of operators on the same line? It becomes difficult to read and increases your chances of making an error. Let's try to calculate `7 ** 2` again but this time represent 7 as 4 + 3.

In [ ]:

```
4 + 3 ** 2
```

That doesn't seem correct. Just like in actual Mathematics, we can put parenthesis when calculating.

Operator precendence in Python works the same way as it does in Mathematics!

In [ ]:

```
(4 + 3) ** 2
```

Exercise

Perform the followig calculations:

1. **Multiply 549 by 72108**
2. **Add 48 to it**
3. **Divide it by 31**

and find the whole-number part of the result.

(Hint: you may at some point find floor division useful)

In [ ]:

```

```

Boolean logic

It would be useful if we can compare values while using them. To do that we can use comparison operators:

Operator	Output
x == y	True if x and y have the same value
x != y	True if x and y don't have the same value
x < y	True if x is less than y
x > y	True if x is more than y
x <= y	True if x is less than or equal to y
x >= y	True if x is more than or equal to y

Make note that these operators return Boolean values (ie. `True` or `False`). Naturally, if the operations don't return `True`, they will return `False`. Let's try some of them out:

In [ ]:

```
x = 5        # assign 5 to the variable x
x == 5       # check if value of x is 5
```

Note that `==` is not the same as `=`

In [ ]:

```
x > 7
```

That is nice! How can we extend this to link multiple combinations like that? Luckily, Python offers the usual set of logical operations:

Operation	Result
x or y	True if at least on is True
x and y	True only if both are True
not x	True only if x is False

Here are some examples of them:

```
True  and True  is True
True  and False is False
False and False is False

False or False is False
True  or True  is True
True  or False is True

not True  is False
not False is True
```

With this knowledge, we can now chain different boolean operations. The simplest of which is to check if a number is within a range.

In [ ]:

```
x = 14
# check if x is within the range 10..20

( x > 10 ) and ( x < 20)
```

As seen, parenthesis are helpful here as well and make the code more readable! That being said, what happens if we have a really complicated boolean logic?

In [ ]:

```
# check if x is a multiple of 2
# check if x is a multiple of 3
# check whether x is not divisible by 6 = 2*3.
x = 14

not (( x % 2 == 0 ) and ( x % 3 == 0))
```

and it became a mess...

To make it more understandable we can introduce intermediate variables like previously:

In [ ]:

```
x = 14

xDivBy2 = ( x % 2 ) == 0 # check if x is a multiple of 2
xDivBy3 = ( x % 3 ) == 0 # check if x is a multiple of 3
```

```
not (xDivBy2 and xDivBy3)
```

Exercise

The code below generates a random number between -50 and 50. Complete the code block using boolean logic to check if `x` lies in the range [0, 10]. (Recall that x in $[a, b]$ means $a \leq x \leq b$.)

In [ ]:

```
import random
x = 100*random.random() - 50

# Finish this block
```

Strings

Strings are very powerful in Python and offer a lot of functionality. For one, they can be added and multiplied. That sounds a bit absurd, how can you add and multiply words? Well, they can and actually they can do much more.

In [ ]:

```
x = "Python"
y = "rocks"
x + " " + y
```

In [ ]:

```
x = "This can be"
y = "repeated "
x + " " + y * 3
```

Strings also have som of built-in functions which alter them directly. Here are some of them:

In [ ]:

```
x = "Edinburgh"
x = x.upper()

y = "University Of "
y = y.lower()

y + x
```

To find out its full capabilities you can use *Tab Completion*. Simply type out the string variable followed by a `.` and then press the Tab character on your keyboard. A dropdown menu should appear showing you all of the methods of strings.

This Tab completion functionality extends to everything else in Python (eg. variables, functions, methods), however, it our case it is a feature provided by Jupyter Notebooks.

In [ ]:

```
x.
```

but what if we want to include numbers in strings? Surely we can keep a number stored as a String (ie. `x = "20"`) but we can't use that number for actual calculations.

The right way to do this is to keep numbers represented by numbers (ie. integers, floats, etc.) and only convert them to Strings whenever needed. To show how to do that let us compute the answer to the universe.

In [ ]:

```
x = 6
x = ( x * 441 ) // 63
"The answer to Life, the Universe and Everything is " + str(x)
```

Exercise

The code below assigns the current day, month and year to variables (you don't need to worry about how this works for now). Complete the block with a string stating `"Today is the Dth day of the Mth month of the Yth year"`, where `D`, `M`, `Y` are given by the variables below.

In [ ]:

```
import datetime
today = datetime.date.today()
day = today.day
month = today.month
year = today.year

#Create your string
```

Printing

When writing your own script you won't have the luxury of printing the value of a variable simply by typing its name(like we have been doing until now), you will have to use `print()` to check any values. Luckily it's quite versatile and powerful but also has a few quirks.

In [ ]:

```
print("Python is powerful and versatile!")
```

In [ ]:

```
str1 = "The string class has"
str2 = 76
str3 = "methods!"
print(str1 + str2 + str3)
```

This doesn't really work and it is not an issue of `print`. It is actually because we are trying to add a String to an Integer. As seen previously, we can convert this to a string using `str()`:

In [ ]:

```
str1 = "The string class has "
str2 = 76
str3 = " methods!"
print(str1 + str(str2) + str3)  # You will have to convert str2 to a string before you ca
n print it
```

Surely Python would have a more elegant solution, right? Correct!

You can use `print` in the format `print(argument1, argument2, argument3, .. ,argumentN)` which allows us to print nearly every type of variable, which is quite useful and doesn't require us to do type conversions all the time.

In [ ]:

```
str1 = "which means it has even more than"
str2 = 76
str3 = "quirks"
print(str1, str2, str3)
```

An alternative to that is the `format()` method which can be applied to a string. For it to work you need to include ordered placeholders `{}` within you string.

In [ ]:

```
history = "The name of {0} is actually a reference to the 1970s BBC show {1}"
print(history)

formatted_string = history.format("Python", "Monty Python's Flying Circus")
print(formatted_string)
```

Exercise

The code below calculates the circumference of the Earth at the equator. Complete the block to print this circumference with a statement of how it was calculated (e.g "Earth's diameter at equator: 12756 km. Equator's circumference: pi * 127556 = __ km").

Print this statement 3 times, using a different method for printing each time.

In [ ]:

```
pi = 3.14159 # Pi
d = 12756 # Diameter of eath at equator (in km)
c = pi*d # Circumference of equator

#Print using +, and casting

#Print using several arguments

#Print using .format
```

Commenting

When writing code you can also write a human-readable explanation of your code in the form of a comment. This can be done by typing in `#` and then writing extra information after it. For example:

- `print(totalCost)` **is ambiguous and we can't exactly be sure what** `totalCost` **is.**
- `print(totalCost)  # Prints the total cost for renovating the Main Library` **is more informative**

Try running the following code. It should fail. Comment out the line of code so that the cell runs without an error (the cell should do nothing).

In [ ]:

```
broken code
```

Exercises

1. Error Spotting

There seem to be a couple of issues with the code below. Can you make it run?

In [ ]:

```
x = "Calculating 252 // 6
print(x)
aswer = 252 // 6
print("The answer is", answer)
```

Run this block to check your answer.

In [ ]:

```
################ Checking code ##########################
```

```
################ Checking code ########################
# Please don't edit this code
if (type(answer) == type(12) and type(x) == type("string")):
    print ("Answer is correct. Good job!")
else:
    print("Wrong answer, please try again.")
```

2. 3D print yourself

Calculate how much it would cost to 3D print a clone of yourself.

1. **Create a variable `pricePerKg` and assign to it the value of 3D printing material. (You can find that out on Amazon).**
2. **Create a variable `myWeight` and assign your weight to it.**
3. **Calculate the total cost, assign it to the variable `total` and print it.**

In [ ]:

```

```

The code below will check if your answer is in the right ballpark.

In [ ]:

```
################ Checking code ########################
# Please don't edit this code
if (40.0 * 10.0 < total < 120.0 * 20.0):
    print ("Answer seems correct. Good job!")
else:
    print("Wrong answer, please try again.")
```

3. Odd or even

First, assign an integer to a variable. Then write code which checks whether the number in the variable is odd or even. The code cell should output `True` if the number is even and `False` if it is odd. At the end of the code cell, you should just end up with a *boolean*!

Note: Don't use if-else statements here, they will be covered later in the course

Hint: how does an even / odd number react differently when divided by 2?

In [ ]:

```

```