Session 1: Introduction to Python

The first thing that you'll need to do is make sure you have Python installed on your computer. You can find information on how to do that here.

Throughout these documents note that text in a `monospaced font` is Python code.

Information about CompSoc

The Oxford University Computer Society was founded in 1978, making it one of the oldest university computing societies in the country.

This year, we will be holding Thursday Talks from both academics and sponsor / Workshops on Thursdays (Weeks 2-8 5-6pm). Learn to code and socials on Saturdays (Weeks 1-7 7pm).

Refer to our website for more and latest details of all events.

IDLE

Once you have Python installed you'll be able to launch a piece of software called IDLE. When you first launch it you will be greeted with a prompt where you can type in statements or expressions in Python. The IDLE is used for learning and experimenting purposes.

Text Editor

The IDLE is not suitable for writing actual large programs. Instead, we would use the text editor method. We can create a new file by clicking **File > New File** at the top menu bar of the IDLE.

Problems installing Python or using IDLE

If you can't get Python installed on your computer, don't worry! Instead, please go to this website, which allows you to run Python programs online. All our examples (until the final session) should run fine in either the online version or IDLE.

Our first program - Hello, world!

The programs that we are going to be writing in this course will all be text-based, i.e. they can take text that you enter as input and produce text as output. Although this might seem limited in the age of animated, interactive apps they provide us with a strong foundation for programming.

One of the most basic programs we can write is one that just outputs some text. In a new file in IDLE (or empty editor on `repl.it`) enter the following Python program:

```
print("Hello, world!")
```

In IDLE, then click **Run > Run Module** (or F5) or click the run button on `repl.it`. IDLE will then run the program in the prompt window, and you should see the output `"Hello, world!"`.

There are three elements to consider here:

- `print` is the name of a **function**. Functions are pieces of code that do something and/or calculate something. The `print` function outputs text (historically, the some of the earliest computers used physical printers as their primary form of output)
- `( )` (parentheses) are used to indicate that we are **calling** the function `print`, i.e. that we are telling Python to execute the `print` function
- `"Hello, world!"` is the **argument** to the function `print`, i.e. the text that we would like the code inside the `print` function to output. You'll notice that when the text is printed the quote marks are missing - this is because we have to indicate to Python that the argument is text, rather than other symbols. We call text that appears in programs like this **strings**

Comments

Throughout the next examples, you will see some text written after a `#` symbol. These are called **comments**, and we will use them to explain and document our code. In Python, any text after a `#` on that line is not considered as part of the program. So we could have written our first program as:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
print("Hello, world!") # Prints Hello, world!
```

Variables

Python programs store data in **variables**. A variable is a bit like a box that has a name, and inside the box we can store values. The value stored could be changed based on user input, or as the program moves on. In Python, we have to put some value in our box when we create it. You can try:

```
greeting1 = "Hello!" # The box named 'greeting1' now contains the string
"Hello!"
my_favourite_number = 3 # The box named 'my_favourite_number' now contains
the integer 3
pi = 3.14 # The box named 'pi' now contains the floating point number 3.14
_underscore = "_" # The box named '_underscore' now contains the string "_"

print(greeting1) # Prints 'Hello!'
print(my_favourite_number) # Prints '3'
print(pi) # Prints '3.14'
print(_underscore) # Prints '_'
```

This created four variables, called `greeting1`, `my_favourite_number`, `pi`, and `_underscore` and then printed the values they store.

To assign a value to a variable we write `name = value` where `name` can be replaced with any piece of text that Python allows as a variable name and `value` can be replaced with any Python value. Then, to get the

value associated with a variable we just write its name - this is exactly the same as if we'd written the value in the first place. The following program functions in exactly the same as the Hello World program we wrote:

```
message = "Hello, world!"
print(message)
```

The way to read = in Python is as *becomes* rather than as a comparison or as a statement (e.g. "my_favourite_number *becomes* 3" rather than "my_favourite_number" *is equal to* 3). This is because values of variables can change over the course of the program.

To see this you can try the following program:

```
my_favourite_number = 3
print(my_favourite_number) # Prints 3
my_favourite_number = 4
print(my_favourite_number) # Prints 4
```

Variable names in Python begin with a letter or an underscore, followed by a sequence of letters, numbers, or underscores. Letters should be restricted to the Latin letters (a-z) and (A-Z), though Python will allow you to use letters such as щ, Æ, ß, and most other Unicode characters (but please don't). One important restriction is that variable names cannot be the same as any of the **reserved** words in Python (also known as **keywords**), which are the following 35 words:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

For your own variables, try to stick with names which are descriptive of what the variable is supposed to contain.

Arithmetic Operations

We can use Python to calculate values. Some examples are given below.

```
print(3 + 4) # Addition
print(3 - 4) # Subtraction
```

```
print(3 * 4) # Multiplication
print(3 / 4) # Division
print(3 // 4) # Integer division
print(3 ** 4) # Exponentiation (i.e. calculates 3 to the power of 4)
```

We can also use variables in the place of numbers, e.g.

```
three = 3
four = 4
print(three + four)
print(3 - four)
print(three * 4)
# ... etc.
```

The order of operations is the same as in mathematics, e.g.

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

is the same as

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

However, you can always use parentheses if you are not sure about something.

Data Types

Each variable in Python has a certain type. Some of the most common types are **strings**, **integers** and **floating point numbers** (**float** for short).

Type	Description	Examples
Integer	Number without decimal places	`1`,`5`, `15`, `15404505`, `-455`, `0`
Float	Number with decimal places	`1.5`, `40.4550590`, `1.0`, `53.4`, `-0.588`, `-458.5`
String	Piece of text	`"a"`, `"boy"`, `"I am a boy!"`, `"I like basketball"`, `"Hello world"`, `"Test"`

Integers and Floats

We can change constants from an integer to a float by appending `.0` at the end.

The result of division using `/` will always result in a float. While division using `//` will result in an integer, the result is **rounded down**.

```
print(7 / 4) # 1.75
print(7 // 4) # 1 (rounded down)
print(8/4) # 2.0 (a float)
print(8//4) # 2 (an integer)
```

For other arithmetic operators mentioned above, they give floats when any of their arguments are floats, while they give integers when all arguments are integers.

```
print(4+4) # 8 (an integer)
print(8.0-4) # 4.0 (a float)
print(3.8*4) # 15.2 (a float)
print(8**4) # 4096 (an integer)
print(16**0.5) # 4.0 (a float)
```

You could convert integers to floats explicitly using the function `float`, and concert floats to integers explicitly using the function `int` (again, the value is **rounded down**). As demonstrated by the examples here:

```
float(40) # 40.0
int(48.9) # 48 (rounded down)
```

On first sight, floats seem to be more powerful than integers, as they can store values with decimal places while integers cannot, and both of them can be used to store integral values.

However, this is not the case. Floats cannot store numbers exactly and will result in rounding errors. It is more reliable to use integers whenever possible.

```
30000000000000000000000001-30000000000000000000000000 # 1
30000000000000000000000001.0-30000000000000000000000000.0 #0.0

int(30000000000000000000000001.0) #30000000000000000570425344
```

Strings

We called `"Hello, world!"` a **string**, i.e. a piece of text that appears in our source code that we can either print or manipulate.

We are surrounding strings in double/single quotes so as to distinguish strings from keywords in Python.

One such example is that we can split this string in two:

```
print("Hello, " + "world!")
```

Here Python will join (`+`) the two strings together to form one string that is then given to the `print` function.

You could convert strings to integers and floats using the `int` and `float` functions respectively. You could also convert integers and floats to strings using the function `str`.

```
int("4") # 4
str(4096.5) # '4096.5'
int("Hello!") # This will cause Python to crash. "Hello!" is not a number!
```

Reading input

We often want our programs to take some kind of input from the user. For example, lets create a program, which prints a personalised greeting. Create a new file and try the following program:

```
name = input("Please enter your name: ")
print("Hi, " + name + "!")
```

Try running this in IDLE, then typing in your name, and pressing Enter. You should get a personalised greeting. Here is what is happening: when you call the `input` function, it will print the message you give it as an argument, and then wait for the user to enter some kind of input. Note that this input is treated as a **string**. In this case, we choose to store this input in the variable called `name`. After that, we just join the strings we received, and print the result.

Now, lets create a program, which multiplies your favourite number by 2. This would look like:

```
your_favourite_number = input("Please enter your favourite number: ")
print("Your favourite number times two is: " + 2 * your_favourite_number)
```

The code looks alright, but something weird happened when you try to run it...

```
Please enter your favourite number: 4096
Your favourite number times two is: 40964096
```

Instead of multiplying the number by two, it repeats the number two times. To understand that, you may try to run

```
"text" * 2 # 'texttext'
```

Python actually defined "multiplcation" of strings as repeating the string for the specified number of times. (And also string "addition", as you have seen earlier)

When the `input` function receives a value it treats it like a **string**. So in order to perform arithmetic operations on our value, we have to tell Python to **convert** it to the appropriate type - in this case **integer**.

To be safe we also should covert intergers and floats to strings using `str` when printing them out

```
your_favourite_number = int(input("Please enter your favourite number: "))
print("Your favourite number times two is: " + str(2 *
your_favourite_number))
```

Worked example: Celsius to Fahrenheit

°F = 9/5 * °C + 32

```
c = float(input("Input temperature in Celsius: "))
f = 9 / 5 * c + 32
print(str(c) + " Celsius = " + str(f) + " Fahrenheit")
```

Sneak Peak - If Statements

We often want our programs to adapt their behaviour based on the input they receive. For instance, lets say we want to create a program, which tells our user if the number they input is small (e.g. less than 10). This is what the next program does:

```
number = int(input("Please enter a number: "))
if number < 10:
  print(number + " is a small number.")
```

We could also have our program say something if the number is large as well:

```
number = int(input("Please enter a number: "))
if number < 10:
  print(number + " is a small number.")
else:
  print(number + " is a large number.")
```

We will discuss more on if statements in the next session.

Exercises

When attempting these exercises, I suggest you type out the code by hand, rather than copy-pasting it. Pay attention to all of the symbols you are typing and see if you can recall why they are there.

1. Now try to combine the last two programs, so that you print out the person's name, as well as their favourite number multiplied by two.

2. Try to write a similar program that converts Fahrenheit to Celsius

   (°F − 32) × 5/9 = °C

3. **Average of two numbers:** create a program which takes as input two numbers and computes their average. We will generalise the program to calculate the average of more than two numbers in the next few sessions.

Good resources to look at

- Khanacademy - good for learning about computer science and computational thinking
- Codecademy - excellent introductory courses in different programming languages. Note that they currently use Python 2.x, whereas we are using Python 3.x
- Project Euler - programming challenges, probably a good place to practice after a couple of weeks
- Learn Python the Hard Way