

# MCSE 6309 and EMoS 6309

## MACHINE LEARNING

### **Tutorial 1: Introduction to Python**

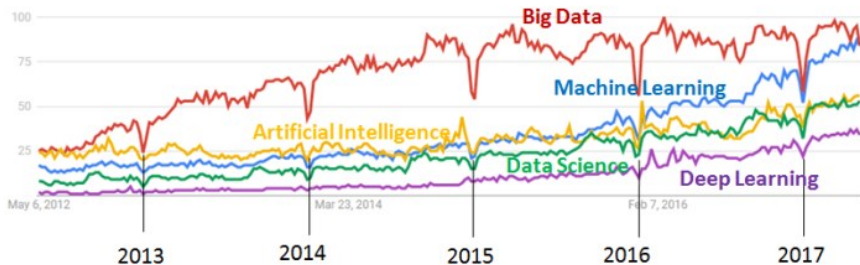
*Acknowledgment:*

Dr. Colin Torney,  
Senior Lecturer in Mathematical Biology & Ecology  
School of Mathematics and Statistics  
University of Glasgow, UK

# Motivation

## Google Trends, May 2012 - April 2017, Worldwide

Big Data, Machine Learning, Artificial Intelligence, Data Science, Deep Learning



# Motivation

All the software is free!

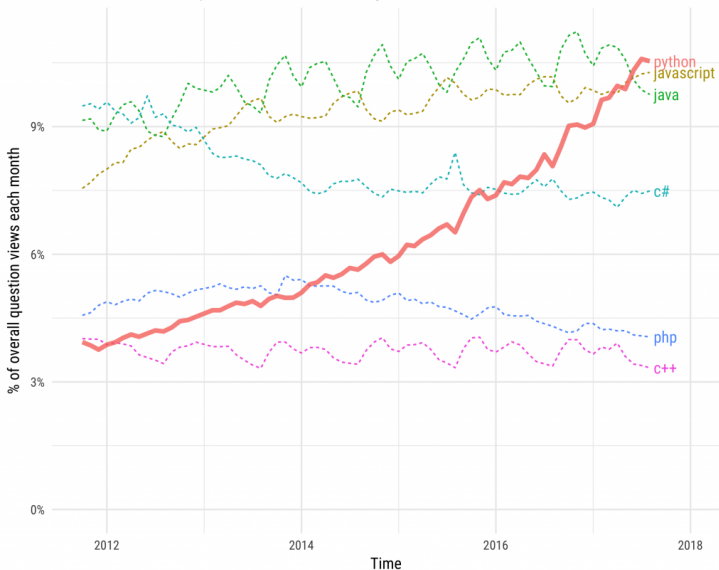


# Introduction to Programming and Python

- A computer is a simple instrument, that takes an input, stores and transforms them according to a program, and produces an output.
- Programs are written in a variety of languages to provide instructions to the computer, telling it what it must do.
- Python is an easy to learn programming language with readable code.
- Python is flexible enough you can use it for many different purposes easily.

## Growth of major programming languages

Based on Stack Overflow question views in World Bank high-income countries



# Ways to use Python

Python is primarily an **interpreted** language. This means that the instructions you enter in the Python language are interpreted by a second program (an interpreter) into machine-code, the internal language used by the computer, every time the program is run.

Source code:  
hello.c



COMPILER



Machine code:



result



Program (also  
called binary,  
executable ...)

Source code:  
hello.py



INTERPRETER



result



# Ways to use Python

This means that you can write instructions in Python in several different ways:

- **Immediate mode:** You type an instruction in Python, it is immediately interpreted and run, with the results displayed straight away. Generally, this is used for simple testing of how the language will work and for quick results you don't need to re-use.
- **Scripts/programs:** You write a series of instructions which are saved into a file. This is usually called a program or a script. This can be re-used whenever you need it and is how Python is generally used to solve problems.

# Jupyter Notebooks

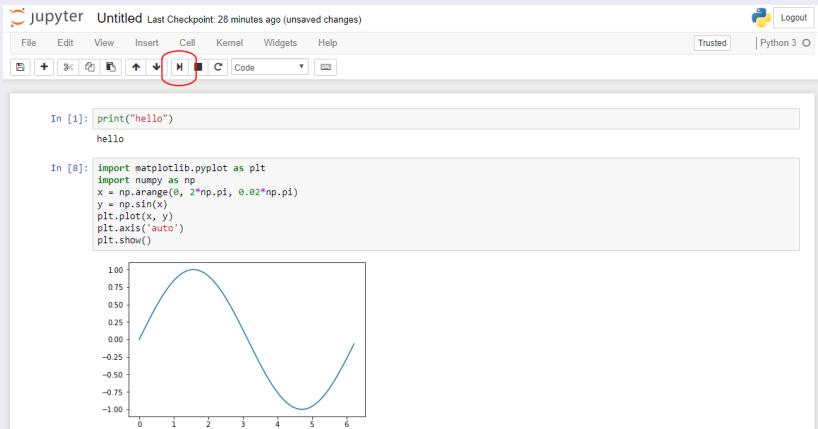
For the duration of this course, we will use a program called **Jupyter Notebook** as our interpreter for Python. Jupyter Notebook allows you to interact with a webpage to execute Python commands and run Python scripts. To start the application, click the relevant icon and wait for the Jupyter Notebook window to open in your web browser - this might take a while. Once it is open, you will see the Notebook Dashboard (as below). To open a blank Python workbook, click the New button (highlighted below) and pick the Python selection.





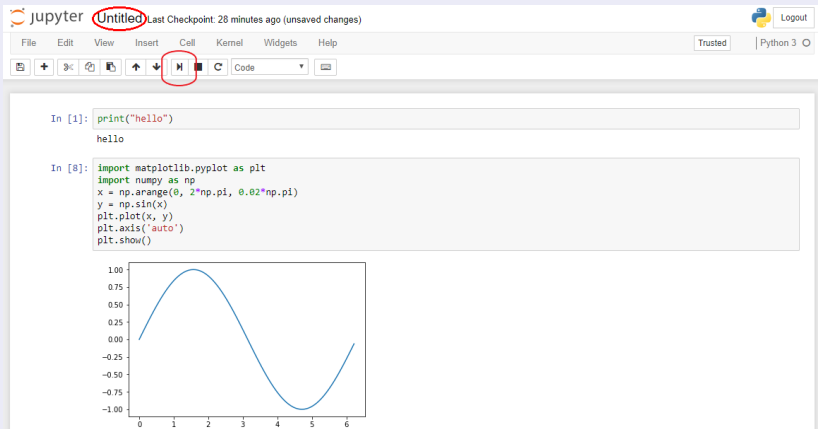
# Jupyter Notebook Cells

When you open a blank Python notebook, you will see a screen like the one below. You can type commands into a cell and then click the "Run Cell" icon (highlighted) to execute the command.



# Renaming a Notebook

To rename the notebook click where it says *untitled* and type a new name. Call your notebook presentation1.



The screenshot shows the Jupyter Notebook interface. The top bar displays the Jupyter logo, the word "Untitled" (circled in red), and the text "Last Checkpoint: 28 minutes ago (unsaved changes)". The top menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. The top right corner shows a Python logo, a "Logout" button, and "Python 3". The toolbar contains icons for file operations, a dropdown menu set to "Code", and a "Trust" button. The main area shows two input cells. The first cell contains the code `print("hello")` and the output "hello". The second cell contains the code for plotting a sine wave: `import matplotlib.pyplot as plt`, `import numpy as np`, `x = np.arange(0, 2*np.pi, 0.02*np.pi)`, `y = np.sin(x)`, `plt.plot(x, y)`, `plt.axis('auto')`, and `plt.show()`. Below the code is a plot of a sine wave with the x-axis ranging from 0 to 6 and the y-axis ranging from -1.00 to 1.00.

# Closing a Jupyter Notebook

When you want to close Jupyter Notebook, do not simply close the browser window. Instead, first go into the File menu at the top left of the Jupyter window and select "Close and Halt". This ensures that the kernel which is running the Python interpreter closes properly.

You can then logout of the Notebook Dashboard via the button at the top right. Remember to close the terminal window which is running Notebook behind the scenes too.

## An example program

We begin with a very simple program, to show how we perform the most basic task in Python. Things get more complicated later! First open your notebook presentation1. Then type:

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

Run the cell to produce the very simple result:

```
Hello world!
```

In this very simple program, `print()` is a Python **function**. Our instruction calls this function with the **parameters** "Hello world!" in brackets, which tells it what to put on the screen.

# Comments

Sometimes we want to write something in a program which will not produce any instructions when the interpreter runs it.

- We might want to write a comment so that someone else examining the program can understand more easily how it works.
- We might want to mark a line of code we have written to be skipped over by the interpreter, perhaps because we think it isn't working properly.

To make a comment or mark a line to be skipped over, we just insert a # character at the start of the line.

```
# This is a Hello World program  
print("Hello world!")
```

Writing clear comments is a good habit to pick up as a programmer - your programs are much easier for other people to read if you comment well.

# Variables

A **variable** is a label we assign to a section of the computer's memory so we can store values in it, manipulate them, and recall them whenever we need.

They are vital to almost all programs, for Python or for any other language. Here is a simple example using the program we typed before:

```
print_this = "Hello world!"  
print(print_this)
```

The first line here is an **assignment** statement. This assigns the value "Hello world!" to the variable `hello_world`.

We can then use that variable as our parameter for the `print()` function call, and it produces the same results as our earlier program.

# Data Types

Any piece of data stored by a Python program is treated as being a certain **data type**. Some simple examples are shown below:

- **String:** "Hello world!" is a string. Strings are usually enclosed in quotation marks (single or double). They are strings of characters, which can be letters, numbers, punctuation marks or anything else.
- **Integer:** An integer is a whole number without any fractional component - 1, 7, 912.
- **Float:** A float is a number which may have a fractional component (but doesn't have to) - 1.213, 7.0, 912.0001.

Unlike some programming languages, you usually do not need to declare a variable as having a particular data type. Python will figure this out for you. It is important to remember that the same data might behave differently depending on what type it is - "7" is not the same as 7.

# User Input

A good way to illustrate how variables might be used in a program is to show a simple example of user input. We can see this in the program below.

```
user_name = input("Type in your name: ")
print("Hello " + user_name)
num_var = int(input(user_name + ", input a number: "))
num_var = num_var + 4
print("Your number plus 4 is " + str(num_var))
```

There are three new functions here, `input()`, `int()` and `str()`. `input()` prints a string then waits for the user to type in something, passing it back to the program.

The last two functions **cast** the data that is passed to them into different data types - integer and string respectively.



# Calculations using Operators

We saw in the last slide how the + sign was used to both concatenate two strings into one, and also perform an arithmetic addition. These kinds of calculations are performed via **operators**.

There are several different kinds of operators, and they can treat data of different types in different ways (for example, + will concatenate strings but add integers). Here are some simple operators and quick descriptions of what they do:

- **Arithmetic:** +, -, \*, / etc.
- **Comparison:** ==, <, >, != etc. These compare values and return True or False.
- **Assignment:** = etc. These assign a value to a variable.
- **Logical:** and, or, not etc. These take logical True or False values and combine them according to basic logical rules to determine if the final result should be True or False.

# Using conditional statements to control program flow

Conditional statements in Python are statements which control the flow of the program. The simplest example is the `if` statement.

```
a = 2
if a < 5:
    print("a is less than 5")
print("Finished!")
```

This program produces this output:

```
a is less than 5
Finished!
```

The indentation and the colon after the `if` statement are very important and required. `if` will execute all statements indented immediately after it, assuming the condition it checks is true.

## More about if

```
if condition:  
    statement_block_1  
else:  
    statement_block_2
```

Above we see the structure of a slightly more complicated version of `if`, the `if...else` statement. If the condition is `True`, statement block 1 executes, otherwise statement block 2 executes.

Each block can consist of multiple lines, as long as every line in the block is indented. Indentations can be tabs or spaces (as long as they are consistent in a single file). The usual choice is 4 spaces.

You can nest multiple `if` statements, as long as you remember to indent each nested statement block multiple times.

## Program flow using loops

We may want to repeat an instruction several times in a program (often called **iterating**). We could do this by writing out the instruction several times, but the more usual way to do this is by using a **loop**. The simplest loop is a `while` statement.

```
x = 0
while x < 5:
    print(x)
    x = x + 1
```

Each time the statement block indented after a `while` statement ends, the program returns to the `while` statement and checks it again. If the condition is still true, the whole statement block repeats. This continues until the statement is false. So the program above will print the numbers 0 to 4 on the screen, then end.

Mistakes in programming can cause a program to be caught in an infinite loop. You would have to forcibly stop the program.

# Packages and Modules

If you are developing a large, complex program, it is normal that you will need certain complex functions over and over in your program. For example, you might need to calculate the area of a circle multiple times. You would normally want to write a function to do this; but calculating the area of a circle might be useful for other programs too. It is the kind of function that would be suitable to place in a **module**.

A module is a collection of Python code that you can **import** into any program, and call on its functions and data types easily.

Modules are often stored in **packages**, collections of modules that you can import together or individually. Dozens of packages are available to be downloaded and installed from online repositories.

# Importing and Using Modules

Using a module in your program is easy - you need to import it before using any of it in your code:

```
import datetime
```

You can then use functions in the `datetime` module by prefixing them with the name of the module. For example, you can use the `time` function like this:

```
t = datetime.time(1,2,3)
```

Typing the full module name every time you use one of its functions can be repetitive, so it is possible to give the module an alias to make it easier to refer to:

```
import datetime as dt  
t = dt.time(1,2,3)
```

## Practical Exercises

# Lists and variables

We have seen before how a variable is used to give us a label for a section of memory so we can refer to it and manipulate it in a program.

What if you want a variable to hold a collection of items? For example, if you wanted a shopping list, you might want to contain “milk”, “salt”, and “eggs”. You can have a separate variable for each, but you might want to have a single variable so you can refer to them all more easily.



# Lists and variables

In Python, we do this using a **list**.

```
shopping_list = ["milk", "salt", "eggs"]
```

To assign the list, you enclose everything to be assigned in square brackets. You then refer to **elements** of the list using a number in square brackets which counts down the list beginning at 0. So,

```
shopping_list[0] = "milk",
```

and

```
shopping_list[2] = "eggs".
```

# Lists and variables

You can get more than 1 element from a list by **slicing** it. If you use a colon like "0:2" when referring to it, it pulls back any element from 0 to just before 2

```
shopping_list[0:2] = ["milk","salt"]
```

# Strings and lists

Strings are a very important data type, and it is possible to treat a string in Python as if it were a list of characters.

```
name = "Universal Income"  
print(name[0])  
print(name[10:])
```

This produces the results:

```
U  
Income
```

Note that you cannot assign parts of a string in this way - `name[0] = "V"` will not work, for example.

## Iterating through a list

Let us suppose you want to perform a simple instruction on every element of a list. You might not know what the list contains, so you obviously want to use a loop to do so rather than trying to write code for everything specifically. But a `while` loop is not the best idea for this.

```
for item in shopping_list:  
    print("Remember to buy " + item + "!")
```

This program simply takes every element in `shopping_list`, assigns it to the variable `item`, and then executes its statement block. It repeats this for every item in the list, starting at item 0.

This is a powerful tool and makes many kinds of programs much easier to write. It also makes it much quicker, if we simply want to repeat a statement block a set number of times, to use a `for` loop than a `while` loop.

# Iterating using the range function

The range function is a relatively simple function that generates a list of integers that a for loop can then iterate through.

```
for x in range(4):  
    print("x is " + str(x))
```

This program produces the following results:

```
x is 0  
x is 1  
x is 2  
x is 3
```

Range has the syntax `range(start, end, step)`. It will generate integers beginning at *start*, adding on *step* each time it iterates, and terminates when it is equal to or passes *end*.

# The numpy module and arrays

numpy is a Python module that contains code to handle data in a new and useful way - an **array**. It is possible to think of an array as like a list - it is a collection of data - but it can have multiple dimensions, not just one.

This makes it ideal to represent images, which are two-dimensional. A 20x20 small image can be represented by a 20 by 20 array, each element of which represents the colour in an individual pixel in the image.

As we are going to be exploring image analysis in detail, understanding how to represent an image as an array, and how arrays work, is vital.



4



# Creating arrays I

To use arrays, the first thing you have to do is to import `numpy`. Once that is done, creating an array can be easy. Here is an example:

```
import numpy as np
a = np.array([[1,1],[1,1]])
```

In this example, `a` is a 2 by 2 square array with every element equal to one. When you create an array, it is a very good idea to make sure it is the size you will need it to be; it is easier for the computer to create an array the correct size first than it is for it to adjust the size later on.

There are many other ways to create arrays directly from images or via combining variables or lists. We discuss these as they come up.

## Creating arrays II

Creating a very large array is sometimes required; in these cases you probably don't want to type in every element as that would be impractical for an array that might possess several million elements. It is much easier to use some inbuilt `numpy` functions to create them:

```
import numpy as np
a = np.ones((2,2,2))
b = np.zeros((2000,2000))
c = np.arange(10)
```

In this example, `a` is a 2 by 2 by 2 cubic array with every element equal to one, `b` is a 2000 by 2000 square array with every element equal to zero, and `c` is a one dimensional array consisting of integers from 0 to 9.

Note the double brackets when we assign to `a` and `b`. Our parameter is a **shape** defining the size of the array we are looking for. So, `(2,2,2)` is a shape representing a three-dimensional array which is 2 elements in size in every dimension.

## Handling arrays I

Now that you have an array, one which might be very large, it is natural to ask what we can do with it. Let's assume that we have a 150x150 image which has been loaded into the array `image_array`, and we want to know how bright the 20x20th pixel in the image is. How do we do this?

```
print(my_array[19,19])
```

What if we have loaded the image into an array, but we don't know what size the image is? Numpy arrays have an attribute called "shape" which we can call on to get a picture of their dimension and how big the array is in each.

```
>>> print(my_array.shape)
(150,150)
```

## Handling arrays II

Note that the `shape` attribute returns a **tuple**. This is a collection (of integers in this case), like a list. Unlike a list, you cannot change it. Like a list, you can pull individual elements out if you are only interested in how big an image is in its x or y dimension.

```
>>> print(my_array.shape[0])  
150
```

We can combine this with loops and the `range` function if we want to do something across an entire array. For example, the following program will fill the array with float numbers between 0 and 1, increasing as the array indices increase.

```
(ysz, xsz) = my_array.shape  
for i in range(ysz):  
    for j in range(xsz):  
        my_array[i,j] = 1.0 * ((i + j) / float(ysz + xsz))
```

# Array based operations

Numpy contains several functions for performing aggregations and calculations across arrays. When working with arrays, it is better whenever possible to act on the whole array at once instead of iterating through it. This is much better for performance.

```
my_array = my_array + 10
```

This adds 10 to the contents of every element in `my_array`.

```
np.amin(my_array)
```

Returns the minimum value in `my_array`.

There are many other numpy functions and Python operators which can be applied to an entire array, or to an entire sliced section of an array, at once. Whenever possible, do this rather than iterate through it.

# Slicing arrays

Sometimes, if you are working with an array, you want to pull back a section of it rather than just a single entry. You do this by **slicing** the array. This works just as it does with lists. For example:

```
>>> a[0,3:5]
array([3,4])
```

```
>>> a[4:,4:]
array([[44, 45],
       [54, 55]])
```

```
>>> a[:,2]
array([2,12,22,32,42,52])
```

```
>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

# Matplotlib

Sometimes you will want to plot a graph or show an image as part of your program. A module exists called `matplotlib.pyplot` which makes doing this much simpler. It must be imported and is usually given the alias `plt`.

Once imported, you can build a plot with a variety of functions and then display it using the `plt.show()` function. Here is a simple example:

```
import matplotlib.pyplot as plt
import numpy as np
plt.style.use('ggplot')
x = np.arange(0,2*np.pi,0.02*np.pi)
y = np.sin(x)
plt.plot(x,y)
plt.axis([0,6.3,-1.1,1.1])
plt.show()
```

# Matplotlib

