LECTURE 1

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in (1991).

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Good to know

- The most recent major version of Python is Python 3. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Python Install

Many PCs and Macs will have python already installed.

To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name>python --version
```

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: https://www.python.org/

Python Quickstart

Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

The way to run a python file is like this on the command line:

```
C:\Users\Your Name>python helloworld.py
```

Where "helloworld.py" is the name of your python file.

Let's write our first Python file, called helloworld.py, which can be done in any text editor.

```
helloworld.py print("Hello, World!")
```

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

```
C:\Users\Your Name>python helloworld.py
```

The output should read:

```
Hello, World!
```

The Python Command Line

To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.

Type the following on the Windows, Mac or Linux command line:

```
C:\Users\Your Name>python
```

From there you can write any python, including our hello world example which will write "Hello, World!" in the command line:

C:\Users\Your Name>python Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32 Type "help", "copyright", "credits" or "license" for more information. >>> print("Hello, World!") Hello, World!

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

exit()

Python Indentation

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code.

In [1]:

```
if 5 > 2:
  print("Five is greater than two!")
```

```
Five is greater than two!
```

Python will give you an error if you skip the indentation:

In [2]:

```
if 5 > 2:
print("Five is greater than two!")
```

```
  Cell In[2], line 2
    print("Five is greater than two!")
    ^
IndentationError: expected an indented block after 'if' statement on line 1
```

The number of spaces is up to you as a programmer, the most common use is four, but it has to be at least one.

In [3]:

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

```
Five is greater than two!
Five is greater than two!
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

In [4]:

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

```
  Cell In[4], line 3
    print("Five is greater than two!")
    ^
IndentationError: unexpected indent
```

Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.
- Comments starts with a #, and Python will ignore them:

In [5]:

```
#This is a comment
print("Hello, World!")
print("Hello, World!") #This is a comment
#print("Hello, World!")
print("Cheers, Mate!")
```

```
Hello, World!
Hello, World!
Cheers, Mate!
```

Python Variables

- Variables are containers for storing data values.
- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.

In [6]:

```
x = 5
y = "John"
print(x)
print(y)
```

```
5
John
```

Variables do not need to be declared with any particular type, and can even change type after they have been set.

In [7]:

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

```
Sally
```

If you want to specify the data type of a variable, this can be done with casting.

In [8]:

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- A variable name cannot be any of the Python keywords.

Assign Values

Python allows you to assign values to multiple variables in one line:

In [9]:

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

```
Orange
Banana
Cherry
```

And you can assign the same value to multiple variables in one line:

In [10]:

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

```
Orange
Orange
Orange
```

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called unpacking.

In [11]:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

```
apple
banana
cherry
```

Python Data Types

Built-in Data Types:

- Text Type: str
- Numeric Types: int, float, complex
- Sequence Types: list, tuple, range
- Mapping Type: dict
- Set Types: set, frozenset
- Boolean Type: bool
- Binary Types: bytes, bytearray, memoryview
- None Type: NoneType

You can get the data type of any object by using the type() function:

In [12]:

```
x = 5
print(type(x))
```

```
<class 'int'>
```

Python Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can assign a multiline string to a variable by using three quotes or three single qoutes:

In [13]:

```
a = """Welcome to first lecture,
Programming languges,
4th year,
IEF."""
print(a)
```

```
Welcome to first lecture,
Programming languges,
4th year,
IEF.
```

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

In [14]:

```
a = "Hello, World!"
print(a[1])
```

```
e
```

Since strings are arrays, we can loop through the characters in a string, with a for loop.

In [15]:

```
for x in "banana":
  print(x)
```

```
b
a
n
a
n
a
```

To get the length of a string, use the len() function.

In [16]:

```
a = "Hello, World!"
print(len(a))
```

```
13
```

To check if a certain phrase or character is present in a string, we can use the keyword in.

In [17]:

```
txt = "The best things in life are free!"
print("free" in txt)
```

```
True
```

In [18]:

```
txt = "The best things in life are free!"
if "free" in txt:
  print("Yes, 'free' is present.")
```

```
Yes, 'free' is present.
```

In [19]:

```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

```
True
```

In [20]:

```
txt = "The best things in life are free!"
if "expensive" not in txt:
  print("No, 'expensive' is NOT present.")
```

```
No, 'expensive' is NOT present.
```

Slicing Strings

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

In [21]:

```
b = "Hello, World!"
print(b[2:5])
```

```
llo
```

In [22]:

```
b = "Hello, World!"
print(b[:5])
```

```
Hello
```

In [23]:

```
b = "Hello, World!"
print(b[2:])
```

```
llo, World!
```

In [24]:

```
b = "Hello, World!"
print(b[-5:-2])
```

```
orl
```

Modify Strings

Python has a set of built-in methods that you can use on strings.

In [25]:

```
a = "Hello, World!"
print(a.upper())
```

```
HELLO, WORLD!
```

In [26]:

```
a = "Hello, World!"
print(a.lower())
```

```
hello, world!
```

In [27]:

```
a = " Hello, World! "
print(a.strip()) # removes any whitespace from the beginning or the end
```

```
Hello, World!
```

In [28]:

```python
a = "Hello, World!"
print(a.replace("H", "J"))
```

```
Jello, World!
```

In [29]:

```python
a = "Hello, World!"
print(a.split(","))
```

```
['Hello', ' World!']
```

In [30]:

```python
a = "Hello"
b = "World"
c = a + b
print(c)
```

```
HelloWorld
```

In [31]:

```python
age = 38
txt = "My name is John, and I am {}"
print(txt.format(age))
```

```
My name is John, and I am 38
```

In [32]:

```python
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

```
I want 3 pieces of item 567 for 49.95 dollars.
```

In [33]:

```python
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

```
I want to pay 49.95 dollars for 3 pieces of item 567.
```

String Methods

Python has a set of built-in methods that you can use on strings.

All string methods return new values. They do not change the original string.

- capitalize() Converts the first character to upper case
- casefold() Converts string into lower case
- center() Returns a centered string
- count() Returns the number of times a specified value occurs in a string
- encode() Returns an encoded version of the string
- endswith() Returns true if the string ends with the specified value
- expandtabs() Sets the tab size of the string
- find() Searches the string for a specified value and returns the position of where it was found
- format() Formats specified values in a string
- format_map() Formats specified values in a string
- index() Searches the string for a specified value and returns the position of where it was found
- isalnum() Returns True if all characters in the string are alphanumeric

- isalpha() Returns True if all characters in the string are in the alphabet
- isascii() Returns True if all characters in the string are ascii characters
- isdecimal() Returns True if all characters in the string are decimals
- isdigit() Returns True if all characters in the string are digits
- isidentifier() Returns True if the string is an identifier
- islower() Returns True if all characters in the string are lower case
- isnumeric() Returns True if all characters in the string are numeric
- isprintable() Returns True if all characters in the string are printable
- isspace() Returns True if all characters in the string are whitespaces
- istitle() Returns True if the string follows the rules of a title
- isupper() Returns True if all characters in the string are upper case
- join() Joins the elements of an iterable to the end of the string
- ljust() Returns a left justified version of the string
- lower() Converts a string into lower case
- lstrip() Returns a left trim version of the string
- maketrans() Returns a translation table to be used in translations
- partition() Returns a tuple where the string is parted into three parts
- replace() Returns a string where a specified value is replaced with a specified value
- rfind() Searches the string for a specified value and returns the last position of where it was found
- rindex() Searches the string for a specified value and returns the last position of where it was found
- rjust() Returns a right justified version of the string
- rpartition() Returns a tuple where the string is parted into three parts
- rsplit() Splits the string at the specified separator, and returns a list
- rstrip() Returns a right trim version of the string
- split() Splits the string at the specified separator, and returns a list
- splitlines() Splits the string at line breaks and returns a list
- startswith() Returns true if the string starts with the specified value
- strip() Returns a trimmed version of the string
- swapcase() Swaps cases, lower case becomes upper case and vice versa
- title() Converts the first character of each word to upper case
- translate() Returns a translated string
- upper() Converts a string into upper case
- zfill() Fills the string with a specified number of 0 values at the beginning

Python Lists

Lists are used to store multiple items in a single variable. .

Lists are created using square brackets:

In [34]:

```python
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

```
['apple', 'banana', 'cherry']
```

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

To determine how many items a list has, use the len() function:

In [35]:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

```
3
```

List items can be of any data type:

In [36]:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
list4 = ["abc", 34, True, 40, "male"]
```

It is also possible to use the list() constructor when creating a new list.

In [37]:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

```
['apple', 'banana', 'cherry']
```

List items are indexed and you can access them by referring to the index number:

In [38]:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

```
banana
```

Negative indexing means start from the end

-1 refers to the last item, -2 refers to the second last item etc.

In [39]:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

```
cherry
```

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

In [40]:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

```
['cherry', 'orange', 'kiwi']
```

In [41]:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

```
['apple', 'banana', 'cherry', 'orange']
```

In [42]:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

```
['cherry', 'orange', 'kiwi', 'melon', 'mango']
```

In [43]:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

```
['orange', 'kiwi', 'melon']
```

To determine if a specified item is present in a list use the in keyword:

In [44]:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

```
Yes, 'apple' is in the fruits list
```

To change the value of a specific item, refer to the index number:

In [45]:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

```
['apple', 'blackcurrant', 'cherry']
```

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

In [46]:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

In [47]:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'cherry']
```

If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

In [48]:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

```
['apple', 'watermelon']
```

To insert a new list item, without replacing any of the existing values, we can use the insert() method.

The insert() method inserts an item at the specified index:

In [49]:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

```
['apple', 'banana', 'watermelon', 'cherry']
```

To add an item to the end of the list, use the append() method:

In [50]:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

```
['apple', 'banana', 'cherry', 'orange']
```

To append elements from another list to the current list, use the extend() method.

In [51]:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

```
['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
```

The remove() method removes the specified item. If there are more than one item with the specified value, the remove() method removes the first occurance:

In [52]:

```
thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)
```

```
['apple', 'cherry', 'banana', 'kiwi']
```

The pop() method removes the specified index. If you do not specify the index, the pop() method removes the last item.

In [53]:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

```
['apple', 'cherry']
```

The del keyword also removes the specified index:

In [54]:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

```
['banana', 'cherry']
```

The clear() method empties the list. The list still remains, but it has no content.

In [55]:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

```
[]
```

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy().

In [56]:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

```
['apple', 'banana', 'cherry']
```

Another way to make a copy is to use the built-in method list().

In [57]:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

```
['apple', 'banana', 'cherry']
```

Sort Lists

List objects have a sort() method that will sort the list alphanumerically, ascending, by default:

In [58]:

```python
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

```
['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

In [59]:

```python
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

```
[23, 50, 65, 82, 100]
```

To sort descending, use the keyword argument reverse = True:

In [60]:

```python
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

```
['pineapple', 'orange', 'mango', 'kiwi', 'banana']
```

You can also customize your own function by using the keyword argument key = function.

The function will return a number that will be used to sort the list (the lowest number first):

In [61]:

```python
def myfunc(n):
  return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

```
[50, 65, 23, 82, 100]
```

By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters:

In [62]:

```python
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```

```
['Kiwi', 'Orange', 'banana', 'cherry']
```

Luckily we can use built-in functions as key functions when sorting a list.

So if you want a case-insensitive sort function, use str.lower as a key function:

In [63]:

```python
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

```
['banana', 'cherry', 'Kiwi', 'Orange']
```

The reverse() method reverses the current sorting order of the elements.

In [64]:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

```
['cherry', 'Kiwi', 'Orange', 'banana']
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a for statement with a conditional test inside:

In [65]:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

```
['apple', 'banana', 'mango']
```

With list comprehension you can do all that with only one line of code:

In [66]:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

```
['apple', 'banana', 'mango']
```

Syntax

newlist = [expression for item in iterable if condition == True]

The return value is a new list, leaving the old list unchanged.