SESSION 1.0: Introduction to Python 3.x

WHY PYTHON?

- Works on multiple platforms
- Has a really easy and comprehensible syntax
- You can write stuff in smaller number of lines compared to other languages
- Its an interpreter language, and can be used in multiple ways (procedural, OOP, functional)
- It's FREE and open, has a very rich library
- But in the end, its just a language.

1.0 BASICS

1.1 Keywords

These following keywords are reserved words that cannot be used as identifiers or variables.

- `and`
- `as`
- `assert`
- `async` [note 1]
- `await` [note 1]
- `break`
- `class`
- `continue`
- `def`
- `del`
- `elif`
- `else`
- `except`
- `False` [note 2]
- `finally`
- `for`
- `from`
- `global`
- `if`
- `import`
- `in`
- `is`
- `lambda`
- `None`
- `nonlocal` [note 2]
- `not`
- `or`
- `pass`
- `raise`
- `return`
- `True` [note 2]
- `try`
- `while`
- `with`
- `yield`

EXERCISE: Start off with something basic. Just use the 'print' command and tell us about yourself.

Example:

```
print("Why am I studying physics???")
```

1.2 Indentation

Python uses whitespace to define code blocks (instead of punctuation of keywords, like in other languages). Indentations are then important as they indicate blocks of code. A wrong indentation will give out an error. Indentation is also useful to make your code readable.

1.3 Comments

When you put '#' in front of a line, the line will be 'commented out'. So, when you later run it, Python will not read it and skip to the next one. Comments are useful from debugging or even

straight out putting notes. You might not remember why you even wrote that line of code after a few days, so doing this helps out a lot. Another way of writing comments is shown below.

Example:

```
print("This line is run as usual")
#print("This line is skipped")
print("This line is run as usual")

"""
Anything between here is a comment as well
print("Don't trust me?")
"""
print("How about now?")
```

1.4 Variables

Variables are essentially containers to store data values or objects. You can write out the name of the container however you want using letters, numbers and underscore. (Of course, there's some sort of convention on how to write them, but we'll learn that later.)

You don't have a specific command to create variables, it is created as you 'assign' a value/object to it. Variables can also be 'redefined' and change types using specific functions.

NB: Python is duck typed, as in it knows automatically what data type or object referenced is stored in a container when you use them. This makes writing code easy, but you tend to make mistakes as your code grows larger and larger.

EXERCISE: Just like the first exercise you did, now instead of giving out full statements, put them inside variables.

Example:

```
name   = Uvuvwevwevwe Onyetenyevwe Ugwemuhwem Osas
age      = 21

print("My name is", name)
```

One can also learn what data type is stored in a variable using the function type(). Try it out.

Example:

```
someVariable = “someWeirdStringButOkay”
print(type(someString))
```

1.4.1 Global Variables

Later, as you learn to write using functions, you’ll see that some variables may need to be declared ‘globally’. A variable that is declared inside a function will only be able to be used in that function-space. So, if you have multiple functions in a script and also importing other functions from libraries then you may have to declare one globally if it is used in multiple cases.

EXAMPLE:

```
someVar = 3.14

def someFunc(radius):
    print(“radius =”, 2*someVar*rad, “metre(s)”)
```

1.4.2 Naming Conventions

There are certain ways to write out the name of the variables. As mentioned before, the variable can be made of any letter, numbers and underscore but it must start with a letter. It really depends on you how to write it out, but if you’re collaborating on a single project its good to use something that everyone can understand quickly.

You can either separate two words using an underscore or capitalize the latter of the two.

EXAMPLE:

```
someNumber  = 5
some_number = 6
```

1.5 Input

You can place in a prompt inside your code to take in inputs from the user using the input() method.

EXAMPLE:

```
number = input(‘Enter a number:’)
print(num)
```

2.0 DATA TYPES & OPERATORS

Python 3.x has multiple built in data types that are commonly used throughout and each comes with a set of functions and operators that do specific things. We will go through the basic ones and leave the rest for you to explore later.

2.1 Built-in Data Types

2.1.1 Boolean (bool)

Booleans are of either two values: True or False

They are useful in flow control when you just want to pass a flag on whether a command is to be executed or not. The bool() function is used to evaluate other values and can also give out True or False depending on what is passed.

Most values are True, like non-empty strings/lists/dictionaries and non-zero numbers.

EXAMPLE:

```
v = 15
w = 0

print((bool(v))
print((bool(w))
```

2.1.2 Strings (str)

Anything that is surrounded by single/double quotation marks are automatically strings. Strings are just a collection (array) of characters in a certain order. You can also write multiline strings by using three quotes. One of Python's strong points is string manipulation, which is far easier than certain other languages.

Example:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""

print(a)
```

2.1.2.1 String Manipulation Methods

As mentioned before, strings are arrays of characters hence each character can be accessed given their position index. This method leads to the ability to ‘slice’ strings according to what you want. The same function is used to access the number of characters in a string and the number of members in a given list (which you will learn later).

NB: Index always starts with 0.

EXAMPLE:

```
someString = “Map of the Soul”

print(someString[3:7])
print(someString[-5:-3])
print(len(someString[2:-7])
```

Other methods, functions that you should check out are:

strip(), lower(), upper(), replace(), split(), capitalize(), casefold(), center(), count(), encode(), endswith(), expandtabs(), find(), format(), format_map(), index(), isalnum(), isalpha(), isdecimal(), isdigit(), isidentifier(), islower(), isnumeric(), isprintable(), ispace(), istitle(), isupper(), join(), ljust(), lstrip(), maketrans(), partition(), rfind(), rindex(), rjust(), rpartition(), rsplit(), rstrip(), splitlines(), startswith(), strip(), swapcase(), title(), translate(), upper(), zfill(),

In-line methods when printing out strings are also available: \", \', \\, \n, \r, \t, \b, \f, \ooo, \xhh.

2.1.2.2 String Combination, Check

An easy way to check if a substring is inside a given string, one can use the keyword ‘in’, which will return a boolean referring to whether it is there or not. You can also add 2 strings together into a single string by using the ‘+’ operator, and this method does not work for other data types besides strings.

EXAMPLE:

```
someString  = “Boruto is just a Naruto reboot”
otherString   = someString+ “ and Sakura was best girl”
subString     = “Naruto”
check          = subString in otherString
print(check)
```

2.1.3 Numbers (int, float, complex)

When you assign numbers to variables, a variable of numeric type is created. ‘Int’ are integers, whole numbers that can be positive or negative. ‘float’, or floating point numbers are basically integers that have decimals. ‘complex’ on the other hand are numbers with the imaginary part written with a ‘j’. You can easily convert one numeric type to another using the functions int(), float(), complex().

Example:

```
x  = 15
y0 = 15.
y1 = 15.0202020202000
z  = 15 + 2j

print(type(x))
print(type(y0))
print(type(int(y0)))
print(type(y1))
print(type(z))
```

2.1.4 Sequence (list, tuple, dictionary)

There are a few sequence data types, but the one that you’ll use the most will be list and dictionaries, depending on what you are doing.

2.1.4.1 Lists

Lists, or sometimes called arrays are one of the most useful data types in Python. Lists are written using square brackets and each member is separated by a comma ([.., .., ..]). We mentioned that strings are essentially arrays of characters, so accessing each member/value is the same. Just give in the index, and it’ll pop out what’s stored there if any. Lists and their members can also be redefined easily (i.e. mutable data type, unlike strings and tuples)

EXAMPLE:

```
someList = [ “stop”, “i”, “could”, “have”, “dropped”, “my”, “croissant”]
print(someList[0])
print(someList[2:4])
```

To add something in a list, we use the append() method. Consequently, to remove something from it, we use the the del command or the remove() method.

EXAMPLE:

```
gravityReading = [10.1, 8.5, 9.8, 16.3, 9.95]
del gravityReading[0]
gravityReading.remove(16.3)
print(gravityReading)
```

NB: To copy a list, you cannot just assign a new variable to an existing list. Any changes made to the new variable will affect the older variable, as it points to the same object. To make a copy, just use newList = someList[:]. The copy() method works as well.

2.1.4.1.1 List Operations

Just like strings, the len() method is used to know the number of members while you can combine two lists into one using '+'. A useful way to iterate through a list is using the for..in.. expression.

EXAMPLE.

```
mean = 0
For reading in gravityReading: mean += reading
mean = mean/len(gravityReading)
print(mean)
```

Some other functions that you should try out are: cmp(), max(), min(), list(), count(), extend(), index(), insert(), pop(), reverse(), sort(), clear(), any(), all(), ascii(), bool(), filter(), iter(), map(), reversed(), sum(), zip().

A notable method is enumerate(). It is useful in many cases, so try enumerate(someList) and print the output. See what you get.

2.1.4.1.2 Nested Lists

You can literally put list inside another list inside another list inside another list inside another list. In a technical sense, they're called nested lists. This way, you can do computations that involve dimensions more than one.

EXAMPLE:

```
trajPoints = [ [ 2 , 3, 4, 5], [ 3, 6, 6, 3] ]
for dim in trajpoints: print(dim)
```

2.1.4.2 Tuples, Sets

Tuples, in a way, work like lists, but instead they use the round brackets ((.., .., ..)) and are immutable (i.e. you cannot reassign/redefine the values given prior) but they can be made up of mutable data types. One workaround to add a member/value to tuples is to change them into a list using list(), append the new value/member and then change it back to a tuple again.

Sets are unordered, and they do not contain duplicate elements unlike other sequence data types. You tend to use them in string manipulation.

Don't bother, you would not be using them much unless the case comes up.

But then again, there's no harm in exploring, right?

2.1.4.3 Dictionaries

Another very useful sequence data type is dictionary. Instead of being referred to by their position index, dictionary values are associated with keys. Those keys can be anything from strings or numbers. The list() method is used to extract the keys used in a dictionary.

EXAMPLE:

```
age = {
        'amir'   : 22,
        'ali'    : 24,
        'auzan' : 25,
}

print(list(age))
for names in age: print(names, "=", age[names])
for name, ageNumber in age.items(): print(name, ageNumber)
```

2.1.4.3.1 Nested Dictionaries

Just like lists, one can make a dictionary inside a dictionary. You can also place a list inside a dictionary too.

EXAMPLE:

```
Students = {
        'amir': {
                'age' : 22.
                'major' : "Physics",
```

```
        },
‘ali’: {
        ‘age’ : 24.
        ‘major’ : “Engineering”,
        },
‘auzan’: {
        ‘age’ : 25.
        ‘major’ : “Physics”,
        },
}
print(Students[“amir”][“major”])
```

2.1.4.4 Range

The range type is an immutable sequence of numbers usually used in loops. To create a range, the range() method is used.

EXAMPLE:

```
print(range(5))
print(range(2, 6, 0.1))
```

2.1.5 Common Operators

Since our main reason for using Python revolves around processing numerical values, mathematical operators are a must know.

2.1.5.1 Arithmetic Operators

They return the more ‘general’ type of numeric data types from the operands.

Addition	:	+
Subtraction	:	-
Multiplication	:	*
Division	:	/
Modulus	:	%
Exponentiation	:	**
Floor division	:	//

2.1.5.2 Comparison Operators

These operators return boolean values.

Greater : >, >=
Less : <, <=
Equal : ==
Not equal : !=

2.1.5.3 Logical Operators

The boolean value they return depends on the operands.

and : True if both operands/statements are True
or : True if either of them are True
not : True if False, False if True

2.1.5.4 Identity Operators

The ‘is’ operator is used when you want to check if the two variables/object refer to the ‘same’ value or not. They return boolean values, but work a bit differently from the ‘==’ operator. The first is to check identity, while the latter is to check equality.

‘is’ can also be coupled with ‘not’ to make ‘is not’.

2.1.5.5 Membership Operators

The ‘in’ operator we encountered in the String subsection. They can be coupled with ‘not’ as well and can be used to check on other data types too, i.e. lists, tuples, sets and dictionaries.

Each operator can also be written in a succinct form when you want to update a value pf a variable, or in a technical sense, are also known as assignment operators.

EXAMPLE:

```
x += 5
y *= 6
```

There are bitwise operators that involve manipulating the bit value of the variables such as &, |, ~, ^, >> and << that you can check out.

EXERCISE: Given the equation 0 = 8x^2 - 6x + 1.125 , find the root using the quadratic formula. Update the root value by adding 6.66 and plug it in to the equation. Translate it into programming.

EXERCISE: Write a Python program to calculate the surface area and the volume of a sphere, with the input given by the user.