

Basic Python

Course Outline

- Introduction to Python
- What can Python do
- Setup and Installation
- Take Note
- Coding proper
- Variable
- Input
- Data Types (Integer, String, Boolean, Float)
- Operators in Python(Logical and Arithmetic operator)
- Flow Control (If statement, try and except and Loops)
- Data structure in Python
- Functions
- Lambda

Introduction to Python

- Python is a multipurpose high level programming language, developed by Guido Van Rossum in 1991, with aim of providing a simple, readable and and performance base programming language.
- Python support multiple programming paradigms including procedural, object oriented, and functional programming style.
- Python is widely used in many field mostly in the STEM(Science Technology, Engineering and Mathematic) organizations. Companies like MANG (Meta, Amazon, Netflix and Google) and other tech organizations uses python as part of it's tech stack.

What can Python do?

- Python is widely used in web development with the availability of packages like Django Flask and FastAPI etc. which also come with a very large and active community.
- Python is also used in Data Science and Machine Learning
- Python can also be use in web scraping, web automation and Scripting Mobile and Desktop App development,

Setup and Installation😊

- To run Python on your local machine(PC i.e Personal computer which can be laptop or even Desktop), you need to install the python IDE from the python official website.

Download ☐ <https://www.python.org/>

- For the course, we'll be using Visual Studio Code (Vscode)

Download ☐

<https://code.visualstudio.com/download>

- Also we'll be making use of Git Bash as our terminal where we would see the outcome of our code

Download ☐ <https://git-scm.com/downloads>

Take Note of the following

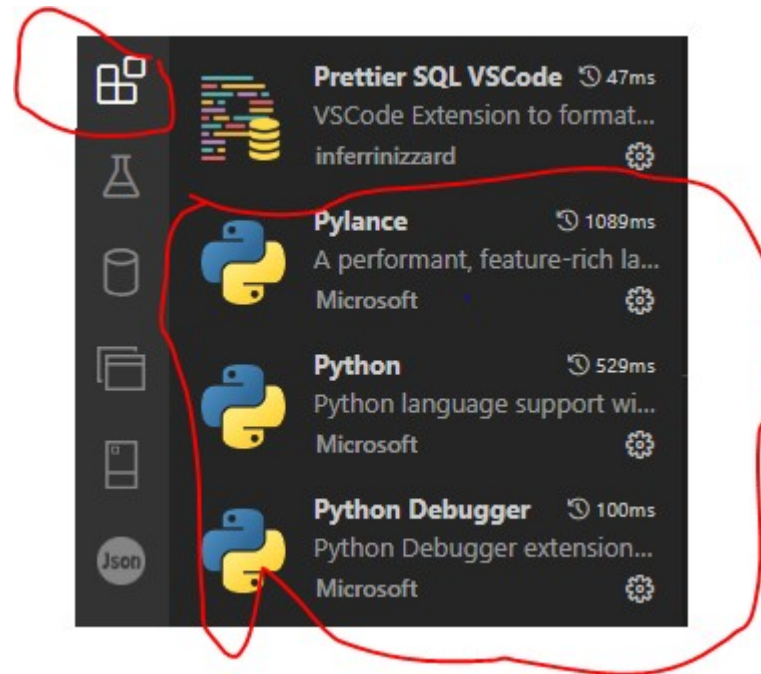
After you've downloaded python install it, but after tapping on the .exe file you'll see a pop-up, tick the "add python to exe....."

In the course bear in mind that python is dynamically typed and reads from top to bottom and left to right.

Lastly Open your VSC, in the top left you'll see the extension as displayed, tap on it



After that download the following by tapping the search like bar at the top and search and download the following Pylance, Python, and Python debugger




Coding Proper

- Now that we have Python IDE, Vscode (i.e our code editor) and GitBash as our terminal we can now start coding
- The first Thing we'll be doing is the log data on our terminal
- Before that create a folder and open the folder in your vscode, then create a .py file in the folder after the vscode is opened
- **PRINT statement**
 - In python, "print" is a syntax or command used to display messages or output to the user via the terminal.
 - It allows you to show information on the screen or console.
 - Example
 - `print("Hello, World!")`**


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

Variable

- A variable is a placeholder for storing data in a program.
- - Variables have names and values
 - Example:
age = 25
name = "John"



```
age = 25  
name = "John"
```

A variable can also be set as a value for a variable for example

```
name = input()
```

What this means is that our variable name “name” has a value of an input, which we can print back on the terminal

```
name = input()
```

```
print(name)
```

output: So let's say after you run the code the typed in “Adeswa”

And you click on enter it still console Adesewa

 app.py x

 app.py > ...

```
1  name = input()  
2  print(name)
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

USER@DESKTOP-5F4T6S4 MSYS ~/Desktop/onazi/bot

\$ python app.py

Adesewa

Adesewa

- Do you know that you can print that variable by:

```
age = 25
```

```
print(age)
```

Now you'll observe that we didn't have to add "" this time, why because it's a variable is not a data type.

Data Types in Python

We have four standard data types in python which are:

- Strings - identified by text in between " " e.g "Hello world"
- Integers – They are numbers in python e.g 0 – 9, they do not require "".
- Floats – These are decimal number e.g 18.99, 45.99 etc
- Booleans – These are True or False statement in python written as **True** and **False** no inverted comma needed

Input

- Now that we've learnt how to log data to our terminal (i.e the print statement), we also need to learn how to collect or accept input from the terminal.
- By default every input from our terminal is a string except if declared
- To collect input from the terminal we'll use the syntax called "input" followed by parenthesis "()"

E.g input()

When we run this we'll see that our terminal allows us to type in via the terminal

We can Take full control of our input by declaring what of data and how the data should be collected.

String input:

An input can be assigned to accept only string and nothing more by adding **str()** and the **input()** in between the str parenthesis e.g

```
username = str(input())
```

Integer input:

An input can be assigned to accept only integers and nothing more by adding **int()** and the **input()** in between the int parenthesis e.g

```
age = int(input())
```

Float input:

An input can be assigned to accept only floats and nothing more by adding **float()** and the **input()** in between the int parenthesis e.g

```
price = float(input())
```


Boolean input:

An input can be assigned to accept only Boolean and nothing more by adding `bool()` and the `input()` in between the `int` parenthesis e.g

`confirm = bool(input())`

Input attribute

Another method of control is by declaring if your string is either an uppercase (by adding `.upper()` after the last bracket of our input), lowercase (by adding `.lower()`) or if only you want to capitalize the first letter of your string by using `capitalize()`

- `.upper()`: The attribute convert lower case input to upper case.

e.g

```
name = str(input()).upper()
```

the code output

 app.py X

 app.py > ...

```
1  name = str(input()).upper()  
2  print(name)
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
USER@DESKTOP-5F4T6S4 MSYS ~/Desktop/onazi/bot  
$ python app.py  
hello  
HELLO
```

- `.lower()`: The attribute convert upper case input to lower case.

e.g

```
name = str(input()).lower()
```

the code output

app.py X

app.py > ...

```
1 name = str(input()).lower()  
2 print(name)
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

USER@DESKTOP-5F4T6S4 MSYS ~/Desktop/onazi/bot

\$ python app.py

HAPPY

happy

- `.capitalize()`: The attribute convert the first character of the input to upper case and retain the rest as lower case character.

e.g

```
name = str(input()).capitalize()
```

the code output

 app.py X

 app.py > ...

```
1 name = str(input()).capitalize()  
2 print(name)
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
USER@DESKTOP-5F4T6S4 MSYS ~/Desktop/onazi/bot  
$ python app.py  
hello  
Hello
```

- Other attributes are:

`.split()`

`.strip()`

`.join()`

Operators in Python (Logical and Arithmetic operators)

- Operators in Python:

Operators in Python are indeed used for performing specific types of logic. They are symbols or special keywords that operate on operands (values or variables) to produce a result.

Types of Operators:

Arithmetic Operators:

- Addition (+): Adds two operands.
- Subtraction (-): Subtracts the second operand from the first.
- Multiplication (*): Multiplies two operands.
- Division (/): Divides the first operand by the second.
- Exponentiation (**): Raises the first operand to the power of the second.
- Modulus (%): Returns the remainder of the division if it exist.
- Greater than sign(>)
- Less than sign(<)
- Greater than or equal to (>=)
- Less than or equal to (<=)
- Equivalent to (==)
- Not equal to (!=)

•

Logic Operators:

AND (and): Returns True if both condition are True.

OR (or): Returns True if at least one of the condition is True.

None : Return True an object requested is not found

In: Return True if object exist in set of data

Is

not

any

FLOW CONTROL (IF STATEMENT AND LOOPS)

If

- If statement
- Nested if

Loops

- for loops
- While loops
- Nested loops

IF STATEMENT

. If Statement:

- The "if" statement is used to make decisions in code based on conditions.

- It executes a block of code only if the condition is true.

- Exam

```
age = 18
if age >= 18:
    print("You are an adult.")
```

It doesn't just end there, we have the **else** and **elif** statements, which is used to handle otherwise condition. For instance let's say we have a form that collect basic information like name age and gender, and we want the program to address each user according to their gender we'll use the else and **elif**

e.g.

```
name = str(input("What's your name ")).capitalize()
age = int(input("What's your age "))
gender = str(input("What's your gender ")).lower()
if gender == "male":
    print("Hello sir")
elif gender == "female":
    print("Hello ma")
else:
    print("Invalid gender")
```

From the example in the previous slide, we can see that our program flow will respond to our users genders. You can see that We used the elif because we have to gender(i.e. gender to satisfy)

And we used the else to handle condition that our user enters an invalid gender.

With this you use if and else only if you have just One option the else is mainly used to respond to non existing conditions outside the given condition

But we use elif if we have more than one options like our example

So if you have 5 condition you have 1 if 4 elifs and 1 else.

You can't have more than 1 else in a control

LOOPS

Loops in Python

In programming, loops are used to execute a block of code multiple times. They are essential for automating repetitive tasks and iterating over collections of data.

for Loops

For Loop:

- A "for" loop is used to iterate over a sequence (like a list, tuple, or string) or other iterable objects.
- It executes a block of code for each item in the sequence.
- Example:

```
for i in range(5):  
    print(i)
```

while Loops

. While Loop:

- A "while" loop repeats a block of code as long as the specified condition is true.

- It's useful when you don't know in advance how many times you need to iterate.

- Example:

```
count = 0
```

```
while count < 5:
```

```
    print(count)
```

```
    count += 1
```

while True

We use the while True to keep the whole program in a loop even if the conditions have been satisfied

while True:

```
    name = str(input("What's your name ")).capitalize()
```

```
    age = int(input("What's your age "))
```

```
    gender = str(input("What's your gender ")).lower()
```

```
    if gender == "male":
```

```
        print("Hello sir")
```

```
    elif gender == "female":
```

```
        print("Hello ma")
```

```
    else:
```

```
        print("Invalid gender")
```

Also we use **break** to end loops(i.e. a block of loop in python)

while True

We use the while True to keep the whole program in a loop even if the conditions have been satisfied while True:

```
        name = str(input("What's your name
")).capitalize()
        age = int(input("What's your age "))
        gender = str(input("What's your gender
")).lower()
        if gender == "male":
            print("Hello sir")
        elif gender == "female":
            print("Hello ma")
        else:
            print("Invalid gender")
break
```

nested Loops

Nested Loop:

- A nested loop is a loop inside another loop.
- Used for iterating over items in multi-dimensional data structures like lists of lists.
- Example:

```
for i in range(3):  
    for j in range(2):  
        print(i, j)
```

Which is also applicable to while loops

DATA STRUCTURES IN PYTHON

- List
- Dictionary
- Tuple
- Set

List

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

Lists are one of 4 built-in structures in Python used to store collections of data, the other 3 are Dictionary, Tuple and Set, all with different qualities and usage.

Lists are created using square brackets:

Create a List:

```
thislist = ["apple", True, 12, 2422.433]
```

```
print(thislist)
```

Attribute of a list

List Items

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.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates

Since lists are indexed, lists can have items with the same value:

Example

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
```

```
print(thislist)
```

List items can be of any data type:

Example

```
list1 = ["apple", "banana", "cherry"]
```

```
list2 = [1, 5, 7, 9, 3]
```

```
list3 = [True, False, False]
```

Note that “()” is a constructor.

type()

What is the data type of a list?

```
mylist = ["apple", "banana", "cherry"]
```

```
print(type(mylist))
```

The list() Constructor

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

Example

Using the `list()` constructor to make a List:

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

```
print(thislist)
```

LIST CONSTRUCTOR

We have 11 list operators in python, which are:

Pop

Append

Clear

Count

Extended

Index

Insert

Remove

Reverse

Sort

copying

Dictionary

Python Dictionary is a powerful and versatile data structure used to store Key value pairs, they are an unordered, mutable and indexed collection. They are used to represent mappings between unique keys and values.

Example of a Dictionary

```
my_dict = {"apple": 2, "banana": 3, "orange": 1}
```

Note that in list named my dict “apple”, “banana”, and “orange” is the key while 2, 3 and 1 are their value each

Accessing Values in a Dictionary:

We can access the values of a dictionary associated with a specific key using square bracket [] e.g

```
print(my_dict[banana])
```

Adding and Modifying Entries

To add a new key-value pair or modify an existing one, simply assign a value to the key:

```
my_dict["grape"] = 4 # Adding a new entry
```

```
my_dict["banana"] = 5 # Modifying an existing entry
```

Removing Entries

You can remove entries from a dictionary using the ``del`` keyword or the ``pop()`` method:

```
del my_dict["apple"] # Removes the entry with key  
"apple"
```

```
my_dict.pop("orange") # Removes the entry with key  
"orange"
```

Removing Entries

You can remove entries from a dictionary using the ``del`` keyword or the ``pop()`` method:

```
del my_dict["apple"] # Removes the entry with key  
"apple"
```

```
my_dict.pop("orange") # Removes the entry with key  
"orange"
```

Dictionary Methods

Python dictionaries offer various built-in methods for performing common operations in dictionary. Some useful methods include:

- ``keys()``: Returns a view object that displays a list of all the keys in the dictionary.
- ``values()``: Returns a view object that displays a list of all the values in the dictionary.

- ``items()``: Returns a view object that displays a list of tuples containing key-value pairs.
- ``get()``: Returns the value associated with a specified key. If the key does not exist, it returns a default value (if provided) or ``None``.
- ``update()``: Updates the dictionary with the key-value pairs from another dictionary or an iterable of key-value pairs.
- ``clear()``: Removes all the key-value pairs from the dictionary.

You can iterate through a dictionary using loops. For example, to iterate through keys:

```
for key in my_dict:
```

```
    print(key, my_dict[key])
```

To iterate through key-value pairs:

```
for key, value in my_dict.items():
```



```
print(key, value)
```

Dictionary Comprehensions

Similar to list comprehensions, you can also create dictionaries using dictionary comprehensions. For example:

```
squares = {x: x*x for x in range(1, 6)}
```

Nested Dictionaries:

```
nested_dict = {  
    "person1": {"name": "John", "age": 30},  
    "person2": {"name": "Alice", "age": 25}  
}
```

Tuple

What is a Tuple?

A tuple in Python is a data structure that is used to store an ordered sequence of elements. It is similar to a list, but with the key difference being that tuples are immutable, meaning they cannot be modified once created. Tuples are created by enclosing comma-separated values within parentheses `()`.

Creating Tuples

Here's how you create a tuple in Python:

```
my_tuple = (1, 2, 3, 4, 5)
```

Tuples can also contain elements of different data types:

```
mixed_tuple = ('apple', 3.14, True, 42)
```

Accessing Elements

You can access elements of a tuple using indexing. Indexing in Python starts from 0.

```
my_tuple = (1, 2, 3, 4, 5)
```

```
print(my_tuple[0]) # Output: 1
```

```
print(my_tuple[2]) # Output: 3
```

Tuple Operations

1. Concatenation:

You can concatenate two tuples using the `+` operator.

```
tuple1 = (1, 2, 3)
```

```
tuple2 = (4, 5, 6)
```

```
concatenated_tuple = tuple1 + tuple2
```

```
print(concatenated_tuple) # Output: (1, 2, 3, 4, 5, 6)
```


2. Multiplication:

You can repeat a tuple using the `*` operator.

```
my_tuple = ('a', 'b')
```

```
repeated_tuple = my_tuple * 3
```

```
print(repeated_tuple) # Output: ('a', 'b', 'a', 'b', 'a', 'b')
```

Tuple Methods

1. count():

Returns the number of occurrences of a specified value in the tuple.

```
my_tuple = (1, 2, 2, 3, 2, 4)
```

```
print(my_tuple.count(2)) # Output: 3
```

2. index():

Returns the index of the first occurrence of a specified value.

```
my_tuple = (1, 2, 3, 4, 5)
```

```
print(my_tuple.index(3)) # Output: 2
```

Tuple Packing and Unpacking

Packing:

Packing multiple values into a tuple.

```
my_tuple = 1, 2, 'hello'
```

```
print(my_tuple) # Output: (1, 2, 'hello')
```

Unpacking:

Assigning individual values of a tuple to variables.

```
my_tuple = (1, 2, 'hello')
```

```
a, b, c = my_tuple
```

```
print(a) # Output: 1
```

```
print(b) # Output: 2
```

```
print(c) # Output: 'hello'
```

When to Use Tuples

Use tuples when you have a collection of items that should not be changed, such as days of the week, coordinates, or database records

Sets

A set in Python is an unordered collection of unique elements. It is similar to the mathematical concept of a set. Sets are mutable, meaning you can add or remove elements from them, but they cannot contain duplicate elements. You can create a set in Python by enclosing comma-separated values within curly braces `{}`.

```
my_set = {1, 2, 3, 4, 5}
```

If you have a list and want to convert it to a set to remove duplicates, you can use the `set()` function.

```
my_list = [1, 2, 2, 3, 4, 4, 5]
```

```
unique_set = set(my_list)
```

```
print(unique_set)
```

Since sets are unordered collections, you cannot access elements by index. You can, however, check for membership using the `in` keyword.


```
my_set = {1, 2, 3, 4, 5}
```

```
print(3 in my_set)
```

```
print(6 in my_set)
```

You can add elements to a set using the `add()` method.

```
my_set = {1, 2, 3}
```

```
my_set.add(4)
```

```
print(my_set)
```

Sets support various operations like union, intersection, difference, and symmetric difference.

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
```

```
print(set1 | set2)
```

```
print(set1 & set2)
```

```
print(set1 - set2)
```

```
print(set1 ^ set2)
```

Sets have several useful methods such as ``union()`,
`intersection()`, `difference()`, `symmetric_difference()`,
and `clear()`.`

```
print(set1.union(set2))
```

```
print(set1.intersection(set2))
```

```
print(set1.difference(set2))
```

```
print(set1.symmetric_difference(set2))
```

```
set1.clear()
```

```
print(set1)
```

Sets are a powerful tool in Python for working with collections of unique elements. They offer efficient methods for set operations like union, intersection, and difference. Understanding sets is essential for any Python programmer.

Function

Function are block of code that gives room for re-useability, modularity and cleaner code. In python we use function to declare lines of code that we want to make reusable.

In python we declare function by the **def** followed by the name(i.e the desired name of your function) **()** parenthesis and column.

E.g. `def greetings():`

Then whatever comes under this line must be indented. Also it is also important that we close our function by using the **return** statement

E.g.

```
def greetings():  
    word = "Hello Boss"  
    return word
```

The above is a proper illustration of a function, aside from just greetings we can perform several logic within the function block and return the output.

The now call a function we move outside the function and type the function name followed by the parenthesis

E.g.

```
def greetings():  
    word = "Hello Boss"  
    return word  
greetings()
```

We can also put a function within the print syntax

```
print(greetings())
```


In python function we can set parameters in a function, which makes our function dynamic and flexible. For example

E.g.

```
def greetings(a):  
    word = f"Hello {a} "  
    return word  
  
greetings("Badmus")
```

The function above has been fixed with one parameter defined as **a** to take one argument which we declared as **"Badmus"**.

E.g.

```
def add(a, b):  
    c = a + b  
    return c  
  
add(2, 4)
```

The output of this code is 6 because the function parameters adds the two arguments and return the output

Lambda

In Python, a lambda function is a compact and anonymous function that can take any number of arguments but can only have one expression. It's often used for short, simple tasks where defining a full function is unnecessary. Let me illustrate how to create and use lambda functions:

1. Basic Syntax:

- A lambda function is defined using the keyword ``lambda``, followed by the arguments and the expression.
- The expression is executed when the lambda function is called, and the result is returned.

```
x = lambda a: a + 10
```

```
print(x(5)) # Output: 15
```

2. ****Multiple Arguments****:

- Lambda functions can take multiple arguments.

- For instance, multiplying `a` with `b`:

```
x = lambda a, b: a * b
```

```
print(x(5, 6)) # Output: 30
```

3. Use Cases:

- Lambda functions are particularly useful when used as anonymous functions inside other functions.
- Suppose you have a function that takes an argument ``n`` and multiplies it with an unknown number:

```
def myfunc(n):
```

```
    return lambda a: a * n
```

```
mydoubler = myfunc(2)
```



```
print(mydoubler(11)) # Output: 22
```

```
mytripler = myfunc(3)
```

```
print(mytripler(11)) # Output: 33
```

4. When to Use Lambda Functions:

- Use lambda functions when you need an anonymous function for a short period of time, especially within other functions.

Remember, lambda functions are concise and handy for specific scenarios