PYTHON

Instructor : Nirajan Bekoju

ML Engineer, Fusemachines Inc.

Imagine a tool that's powerful enough to drive Artificial Intelligence, easy enough to Create Websites, flexible enough to Analyze Data, and handy for Scientific Research, all while being Beginner-Friendly!

That's Python...

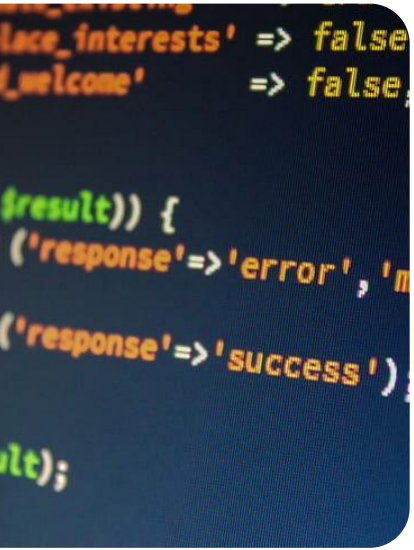
Training - Outline

- Introduction to Python Programming
- Data Types, Variables and Strings
- Basic Data Structures
- Conditionals, Loops, and Functions
- Object-oriented Programming (OOP)
- Decorators and Generators
- Error Handling, Logging and Debugging
- File Handling
- Introduction to Modules and Libraries
- Data Science and Visualization Basics
- Introduction to Machine Learning
- Coding Exams and Interview Questions

5 Reasons Why You Should Learn Python

- Beginner-Friendly
- Versatility Across Fields
- Career Opportunities
- Extensive Libraries & Community Support
- Cross-Platform & Future-Proof

print(" Welcome to Python ! Your journey into code begins here.")



Introduction

- ❏ Programming Language
- ❏ Overview of Python and its applications
- ❏ Setting up the environment (Virtualenv., Jupyter Notebooks, VSCode)
- ❏ Python Syntax, Variables, and Data Types

Programming Language

- Programming Language is a formal language that is used to write instructions that a computer can understand and execute.

Feature	High-Level Languages	Low-Level Languages
Abstraction	Closer to human language	Closer to machine language
Ease of Use	Easier to read, write, and maintain	Harder to read and write
Examples	Python, Java, C++	Assembly, Machine Code
Portability	Platform-independent	Platform-specific
Execution Speed	Slower (needs translation)	Faster (executed directly)

Hello world !

Hello World Program in 3 Different Language.

C

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Python

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

Java

```java
public class HelloWorld {
    public static void main(String[]
        args)
    {
        System.out.println("Hello,
            World!");
    }
}
```

Overview of Python and its applications

- Python is a high-level, interpreted programming language created by Guido van Rossum and first released in 1991.
- Known for its simplicity and readability.
- Open-source and cross-platform.

Van Rossum was inspired by the British comedy series Monty Python's Flying Circus and wanted to create a language that was fun and approachable. He also wanted to create a language that was easy to understand and efficient to write, with clear syntax.

Built Python from scratch in three months using a C compiler on his home Macintosh.

Top Companies who uses Python

Netflix uses python in personalized recommendations, Data analytics, Text automation. etc.

Google uses python in web services, in data processing and in Machine Learning and AI etc.

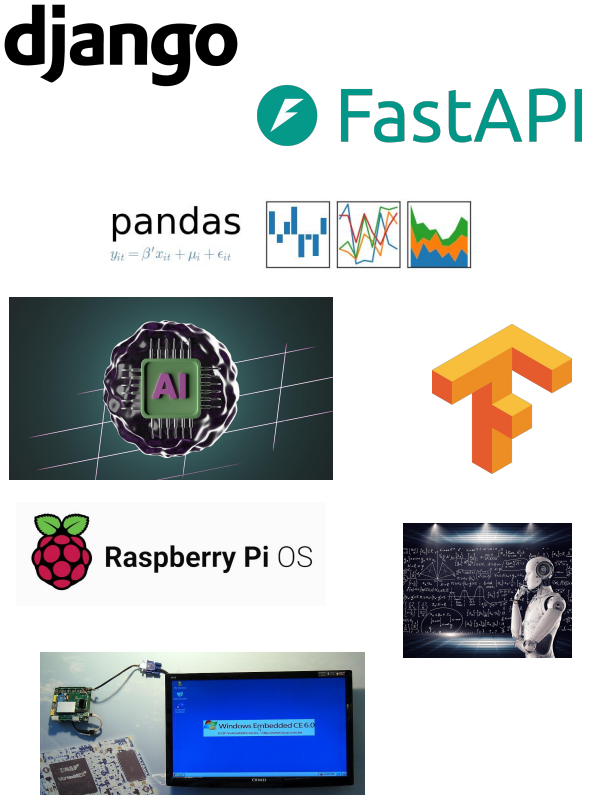
Spotify uses python for data processing and data analysis. etc.

Instagram (Meta) uses python for performance optimization and in web framework (django). etc.

Reddit uses python to handle massive traffic, web services and at the core. etc.

Applications of Python Programming

- Web Development
- Machine Learning & Artificial Intelligence
- Data Science and Analytics
- Automation/Scripting
- IoT and Embedded Systems
- Game Development, Cybersecurity etc.



SET UP

- Install Python
- Install Vs Code (IDE)
- Download Packages: Virtualenv , ipykernel.. etc.

VS Code

VS Code is a lightweight, customizable IDE with built-in Python support, debugging, and Git integration, making it perfect for both beginners and professionals.

VS Code offers excellent support for Jupyter Notebooks through its extensions

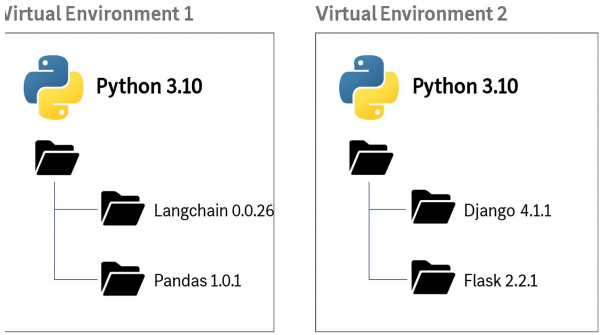
Virtualenv

Virtualenv is a tool for creating isolated Python environments, allowing developers to manage dependencies for different projects without conflicts. It's lightweight, easy to use, and ensures consistent environments across development and deployment.

Coding.....

Virtual Environment

- A virtual environment in Python is an isolated environment on your computer, where you can run and test your Python projects.
- Think of a virtual environment as a separate container for each Python project. Each container:
  - Has its own Python interpreter
  - Has its own set of installed packages
  - Is isolated from other virtual environments
  - Can have different versions of the same package



Virtualenv

Installation

python -m venv /path/to/new/virtual/environment

Creating Environment in Windows

python -m venv /path/to/new/virtual/environment

Note: On Microsoft Windows, it may be required to enable the `Activate.ps1` script by setting the execution policy for the user. You can do this by issuing the following PowerShell command:

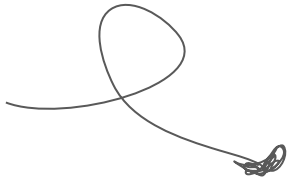
```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

See About Execution Policies for more information.

Activate

For Linux:
source .venv/bin/activate

For Windows:
.venv\Scripts\activate



<venv>\Scripts\Activate.ps1

Deactivate

deactivate

Day 2

- Data Types
- Variables
- Strings
- Escape Sequences
- Input/Output
- Operator Precedence

Syntax

- Syntax refers to the guidelines that determine the structure of a language.
- If the syntax of a language is not followed, the code will not be understood by a compiler or interpreter.

Key Features of Python Syntax:

- **Comments**: Single line comment (#) , Multiline comment (""" """).
- **Case Sensitivity**: Python is case-sensitive.
- **Statements**: Python executes the statement in line by line manner. I.e one line at a time.
- **Input/Output** : input() is used to take user input. And print() is used to show the output.
- **Indentation**: Python uses indentation to define code blocks instead of braces { }.

Variables

Variables

- Variables are containers for storing data values.

Rules for creating the variable name:

- Must start with a letter or the underscore character
- Cannot start with a number
- Can only contain alphanumeric characters and underscores (A-z, 0-9, and _)
- Case-sensitive (mobile, Mobile and MOBILE are three different variables)
- Cannot be any of the Python keywords.

Note: Variables do not require a specific type declaration and can change their type even after being assigned a value.

Constant

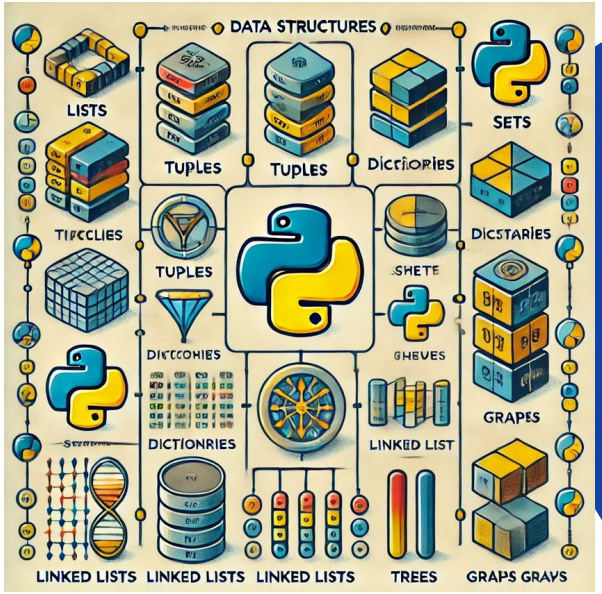
- Python doesn't have inbuilt method to declare Constant. But the convention is to use UpperCase Letter.

Data Type

Data type refers to the classification or category of data that determines the kind of value a variable can hold and the operations that can be performed on it.

Data Types	Classes	Description
Numeric	int, float, complex	holds numeric values
String	str	holds sequence of characters
Sequence	list, tuple, range	holds collection of items
Mapping	dict	holds data in key-value pair form
Boolean	bool	holds either `True` or `False`
Set	set, frozenset	hold collection of unique items

Python is a dynamically typed language, meaning variables do not need explicit type declarations.



Data Structures

- Lists, Tuples, Sets, Dictionaries
- List and dictionary comprehensions
- Basic operations and methods for each data structure
- Common use cases in data manipulation

Data Structures

A data structure is a method for organizing, storing, and managing data in a way that allows for easy access and modification. It specifies how data is arranged in memory and outlines how various operations—like searching, sorting, inserting, and deleting—can be carried out efficiently.

Common examples of data structures include lists, tuples, sets, dictionaries, arrays, stacks, queues, linked lists, trees, graphs, and hash tables.

Lists: Mutable, ordered, indexing, slicing, methods (append(), remove(), sort(), reverse(), etc.)
Tuples: Immutable, ordered, packing & unpacking, count(), index()
Sets: Unordered, unique elements, set operations (union, intersection, difference)
Dictionaries: Key-value pairs, accessing/modifying elements, dictionary methods (keys(), values(), items(), update(), etc.)

Advance Data Structure:

- Stack (LIFO), Queues, Priority Queues, LInked LIsts
- Hash Tables
- BInary Trees, BST
- Graph Data Structure

Data Structures

List and Dictionary Comprehensions:

In Python, List and dictionary comprehensions provide a concise way to create lists and dictionaries in Python using a single line of code.

List Comprehension

Syntax: [expression for item in iterable if condition]

Dictionary Comprehension

Syntax: {key: value for item in iterable if condition}

Why ? - Concise and readable code

- Faster than traditional Loops
- Useful while filtering and transforming

Basic Operations and Methods:

List : Accessing Elements, Slicing, Adding Elements (append, insert, extend, remove, pop, del, sort, sorted, reverse, iterate)

When it is used ?

- Sorting and processing sequential data (eg, list of numbers, user data)
- Sorting and searching
- Stacks (append, pop) and Queues (collection.deque)

Data Structures

Dictionary: Accessing Values, adding and updating values, removing keys, iterating(checking membership, merging dictionaries)

When it is used ?

- For Lookups using key-value pairs
- Storing structuring data (JSON-like objects)
- Caching results (Memoization)

Tuple: Accessing Values, Slicing, Concatenation, Repetition, Checking membership, Coughing occurrence.

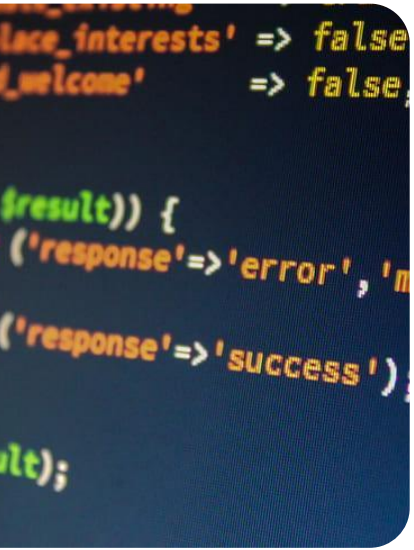
When it is used ?

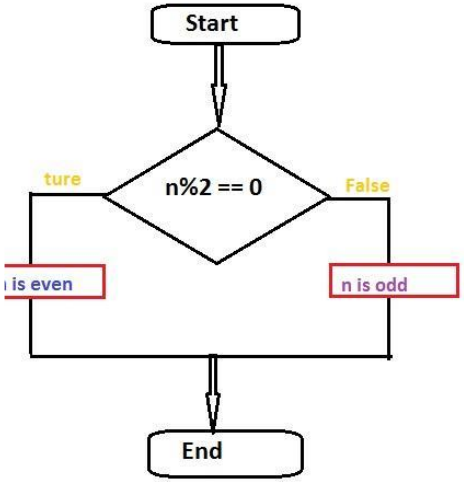
- Storing fixed collection of data (coordinates, db records)
- Returning multiple values from functions
- Hashable feature.

Set: Adding elements, Removing elements, Checking membership, Set operations(union, intersection, difference, symmetric difference)

When it is used ?

- Removing duplicate value from list
- Checking fast membership
- Mathematical set operations (union, different etc)





Control Structures and Functions

- Conditional statements: if, else, elif
- Loops: for, while, and control keywords (break, continue)
- Functions: Defining, calling, return values, lambda functions
- Scope and Lifetime of Variables

Control Structures and Functions

Conditional statements:

Conditional statements in programming are control flow structures that allow a program to make decisions and execute specific blocks of code based on whether a certain condition is True or False. These conditions are typically based on logical or comparison expressions.

Types of Conditional Statements

- **if**: Executes a block of code if the condition is true.
- **else**: Executes a block of code if the condition is false.
- **elif**: Checks additional conditions if the initial if condition is false

Why Are Conditional Statements Required?

- Decision-Making
- Control Flow
- Dynamic Behavior
- Error Handling
- Efficient Resource Usage

Control Structures and Functions

Loops:

Loops are a fundamental concept in programming that allow you to execute a block of code repeatedly as long as a specified condition is true. Python provides two main types of loops: **for** and **while**, along with control keywords like **break**, **continue**, and **pass** to manage loop behavior.

For Loop

Used to iterate over a sequence (like a list, tuple, string, or range). Often used when the number of iterations is known beforehand. Combined with the enumerate() function for index-value pairs:

While Loop

The while loop runs as long as a specified condition evaluates to True. Used when the number of iterations is not known beforehand and depends on a condition.

Control College

Control keywords allow you to manage the flow of loops by breaking out, skipping iterations, or using placeholders.

Break: Terminates the loop prematurely, regardless of the loop condition.

Continue: Skips the current iteration and moves to the next iteration of the loop.

Pass: Does nothing. It serves as a placeholder when a statement is syntactically required but no action is desired.

Control Structures and Functions

Function:

Functions are "self contained" modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. Once a function is written, it can be used over and over and over again. Functions can be "called" from the inside of other functions.

- **Defining Function:**
  A function is a block of reusable code that performs a specific task. In Python, functions are defined using the def keyword.
- **Calling Function**:
  To execute a function, you call it by its name and pass the required arguments (if any).
- **Return Values:**
  Functions can return a value using the return statement. The return keyword ends the function execution and sends a value back to the caller.

Lambda Functions (Anonymous Functions)

A small, one-line, anonymous function defined using the lambda keyword. It is useful for short operations or functions that are used as arguments to other functions. Lambda functions are often used with higher-order functions like map, filter, or reduce.

Control Structures and Functions

Scope:

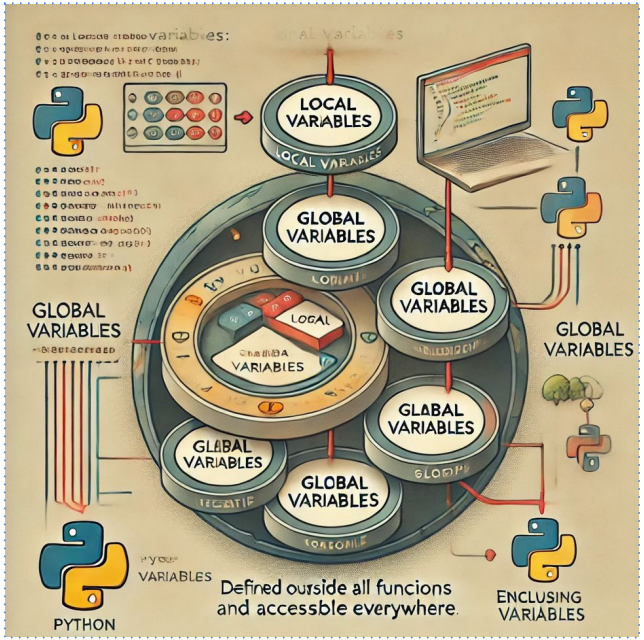
In Python, the scope of a variable refers to the area of a program where that variable can be accessed or modified. The lifetime of a variable refers to how long the variable exists in memory during the program's execution.

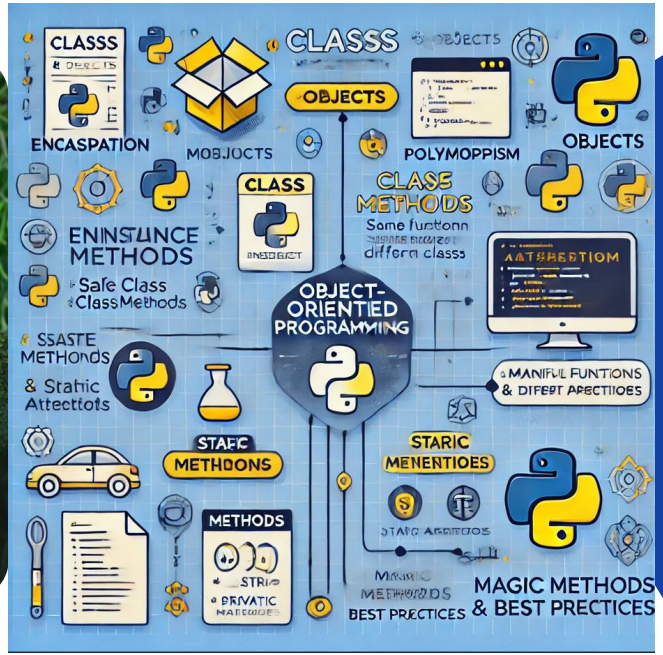
Scope of variables:

- Local
- Enclosing
- Global
- Built-in

Lifetime of Variables

- Local Variables
- Global Variables
- Enclosing Variables





Object-oriented Programming (OOP)

- ❑ What is OOP? Classes and objects.
- ❑ __init__: Constructor method.
- ❑ Four pillars of OOP: Encapsulation, abstraction, inheritance, polymorphism.
- ❑ super(): Accessing parent class methods.
- ❑ Multiple inheritance: Combining multiple parent classes.
- ❑ Dunder methods: __str__, __repr__, __add__, etc.

Object-oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects.

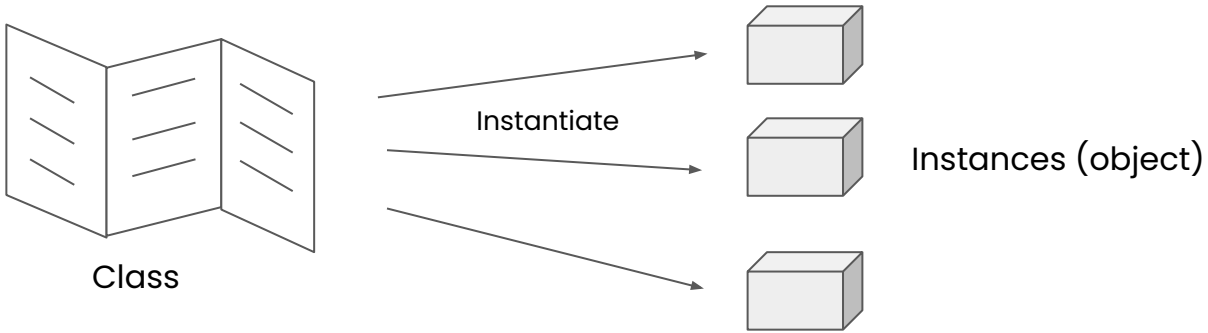
- data (attributes)
- behavior (methods)

Example:

- It helps structure code using real-world entities.
- A car (object) has attributes (color, model) and behaviors (drive, stop).

Classes and Objects

- Classes are blueprints for creating objects, and objects are instances of classes.
- Objects store attributes (variables) and methods (functions) that define their behavior.



Object-oriented Programming (OOP)

Creating class

```
class PlayerCharacter: # CamelCase for classes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def run(self):
        print(f"Player {self.name} of age {self.age} is running.")
```

Constructor

The __init__ Method (Constructor) :

__init__ is a special method (dunder method) that initializes object attributes when an instance is created. It is automatically called when a new object is instantiated.

Self represents the instance of the class and allows access to its attributes and methods. It must be the first parameter in instance methods but is not a keyword (can be renamed, though not recommended)

Object-oriented Programming (OOP)

Class instance

```
player1 = PlayerCharacter('Nirajan', 24)
player2 = PlayerCharacter('John', 20)

player2.attack = 50  # Creating attribute

print(player1.age) # 24
print(player2.name) # John
player1.run() # Player Nirajan of age 24 is running.
print(player2.attack) # 50
```

Object-oriented Programming (OOP)

Class attribute

Class attributes are static unlike attributes used in __init__. They are same for all objects and are accessed by using class name directly.

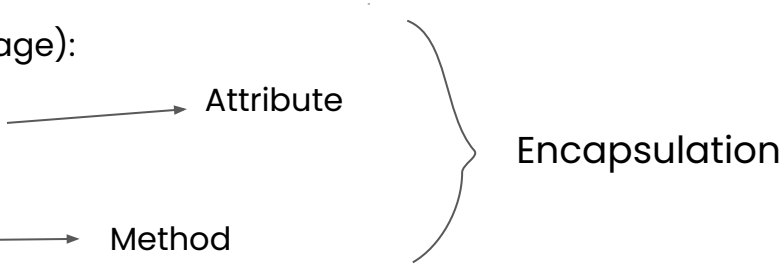
```
class PlayerCharacter:
        membership = True
        def __init__(self, name,):
                self.name = name

        def shout(self):
                if PlayerCharacter.membership:
                        return self.name
                        # return PlayerCharacter.name # This gives error because name is class attribute

obj1 = PlayerCharacter("Nirajan")
print(obj1.shout()) # Nirajan
```

Four Pillars of OOP (Encapsulation)

Encapsulation:

Binding data (attribute) and functions (methods) that manipulate the data.

```
class PlayerCharacter:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def run(self):
        print("Run")

player1 = PlayerCharacter('Nirajan', 24)
```



Also in Python code, when we create a string, because of encapsulation we have different functions and methods available that we can access.
For example: upper(), count(), reverse(), etc.

Four Pillars of OOP (Abstraction)

Abstraction:

Hiding of information or abstracting away information and giving access to only what's necessary.

```
class PlayerCharacter:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def run(self):
        print("Run")

player1 = PlayerCharacter('Nirajan', 24)
player1.run()
```

Here, when we call run, we don't really care how run is implemented. All we know is that player1 has access to run method and we can use it.

Four Pillars of OOP (Abstraction)

Abstraction:

There is a concept of protected and private in abstraction.

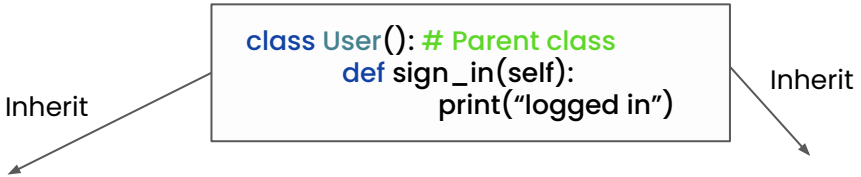
- Using the **single underscore** makes **protected** members or methods, and protected variables are accessible in classes and subclasses only.
- A **double underscore** is a bit tricker. These are refers as **private** members or methods, but they aren't really private either since we can still access it.

```
class PlayerCharacter:
        def __init__(self, name, age):
                self._name = name               # Protected member
                self.__age = age                # Private member

        def _run(self):                         # Protected method
                print("Run")


player1 = PlayerCharacter('Nirajan', 24)
print(player1.__age)                            # This will give an error
print(player1._PlayerCharacter__age)            # But this will work
```

Four Pillars of OOP (Inheritance)

Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.




```
class Wizard(User): # Child class
        def __init__(self, name, power):
                self.name = name
                self.power = power

        def attack(self):
                print(f"attacking with power of {self.power}")

wizard1 = Wizard("Merlin", 50)
wizard1.attack() # attacking with power of 50
```

```
class Archer(User): # Child class
        def __init__(self, name, num_arrows):
                self.name = name
                self.power = num_arrows

        def attack(self):
                print("attacking with arrows")

archer1 = Archer("Robin", 100)
archer1.attack() # attacking with arrows
```

Four Pillars of OOP (Inheritance)

Super()

In inheritance, when we want to use attribute from main class to sub class, we can do as:

```
class User(): # Parent class
        def __init__(self, email, address):
                self.email = email
                self.address = address

        def attack(self):
                print("do nothing")

class Wizard(User): # Child class
        def __init__(self, name, power, email, address):
                super().__init__(email, address) # can also be done by User.__init__(self, email, address)
                self.name = name
                self.power = power

        def attack(self):
                print(f"attacking with power of {self.power}")

wizard1 = Wizard("Nirajan", 50, "nirajan@gmail.com", "Hattiban")
print(wizard1.email) # nirajan@gmail.com
```

Using super to inherit attribute

Four Pillars of OOP (Inheritance)

Multiple Inheritance

Allows a class to inherit from more than one parent class.

```
class A:
    def greet(self):
        print("Hello from A")

class B:
    def greet(self):
        print("Hello from B")

class C(A, B):
    pass

obj = C()
obj.greet() # Hello from A
```

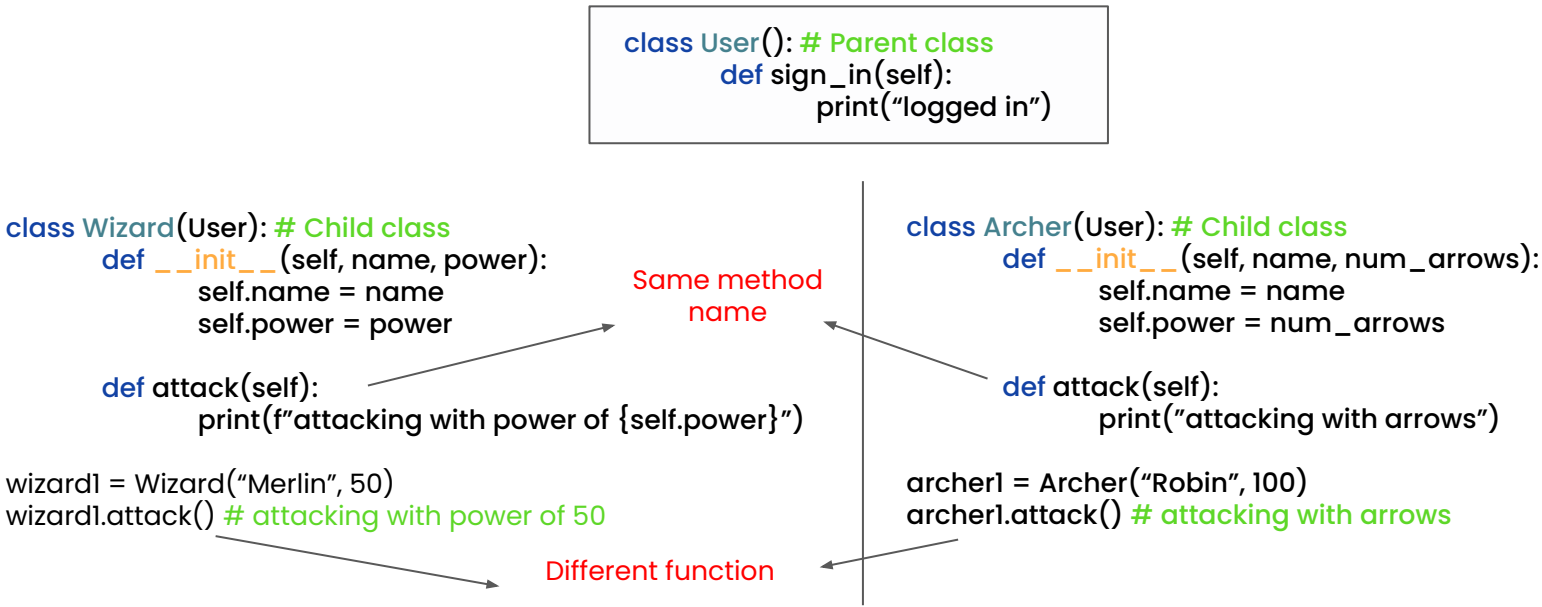
The reason C.greet() calls the method from A is because of Python's Method Resolution Order (MRO), which determines the order in which classes are searched when executing a method. MRO is left-to-right, depth-first (C3 linearization). No need to know this algorithm.

```
print(C.__mro__)
# OR
print(C.mro()) # (<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

Four Pillars of OOP (Polymorphism)

Polymorphism

- Poly means many and morphism means form so polymorphism means having many form.
- In python, polymorphism refers to the way in which object classes can share the same method name. But these method names can act differently based on what object calls them.



Object-oriented Programming (OOP)

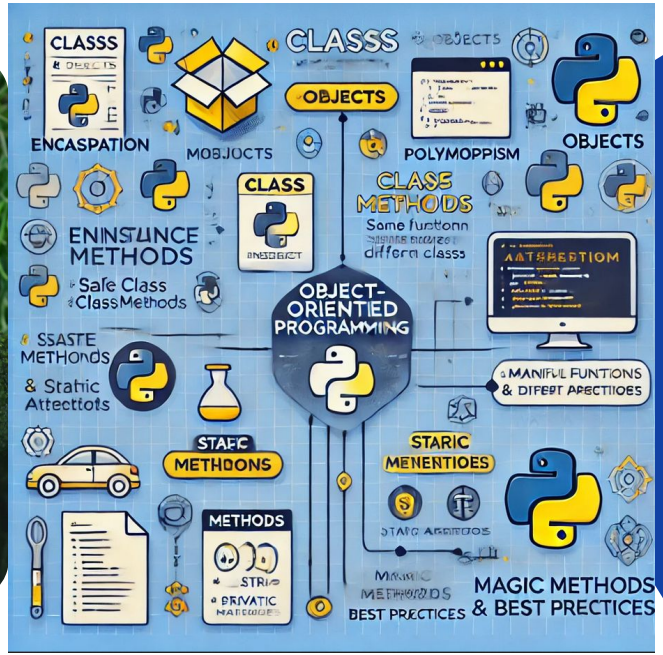
Dunder Methods (Magic method)

- Special methods that starts and end with double underscores.
- When we add two numbers using the + operator, internally the __add__() method will be called.
- We can modify dunder methods for specific class. We usually don't modify dunder method but there are some cases when we have to.

```
print(action_figure.__str__())
        is same as
print(str(action_figure))
```

```
class Toy():
        def __init__(self, color):
                self.color =  color

        def __len__(self):
                return 10

action_figure = Toy('red')
print(len(action_figure))     # Output: 10
print(len('hello'))           # Output: 5;  Not class object so not applicable
```

Coding.....



Decorators

- Prerequisites for learning decorators
  - Nested Function
  - Pass Function as Argument
  - Return a Function as a Value
- Python Decorators
- @Symbol with Decorators
- Decorating Functions with Parameters

Prerequisites for learning decorators

- Nested Function
- Pass Function as an Argument
- Return a Function as a value

Function Call vs Function Object Assignment

Function Call

```
def add(x, y):
    return x + y

addition_result = add(10, 20)
print(addition_result) # pring 30
```

Function Object

```
def add(x, y):
    return x + y

f = add  # f is now a reference to the function 'add'
print(f(3, 4))  # prints 7
```

Pass Function as an Argument

- Just like we can pass integers or other data type as an argument in functions, we can also pass one function as an argument in another function.

```
## Argument and Parameter in Normal Function
def add(x, y):  ## x, y are the parameters
of this function
    return x + y

addition_result = add(10, 20) ## 10, 20 are
the arguments
print(addition_result) # pring 30
```

```
## Example of PASS FUNCTION AS AN ARGUMENT
#### Pass add(...) function as an argument in
calculate() function
def add(x, y):
    return x + y

def calculate(func, x, y):
    return func(x, y)

result = calculate(add, 4, 6)
print(result)  # prints 10
```

Nested Function

- **Nested Function :** Function defined inside a function.

```
## Nested for loop example
for i in range(1, 4):  # Outer loop
    for j in range(1, 4):  # Inner loop
        print(f"i={i}, j={j}")
```

```
## An Example of Nested Function
def greeting(name):  ## Outer Function
    def hello(): ## Inner Function
        return "Hello, " + name + "!"
    return hello

greet = greeting("Atlantis")
print(greet())  # prints "Hello, Atlantis!"
```

Return a Function as a Value

- Return Inner Function as a value from Outer Function.

```
### Nested Function Example
def outer(x):    ## Outer Function
    def inner(y):   ## Inner Function
        return x + y
    return inner

add_five = outer(5)
result = add_five(6)
print(result)   # prints 11
```

Python Decorators

- Python Decorators is a function that takes a function as an argument and returns it by adding some functionality.

```
## This is the Decorator Function
def make_pretty(func):
    def inner():
        print("I got decorated") ## Added Functionality
        func()
    return inner


## we will decorate this function use make_pretty decorator
def ordinary():
    print("I am ordinary")

decorated_func = make_pretty(ordinary)
decorated_func()

Output:
I got decorated
I am ordinary
```

make_pretty is a decorator function

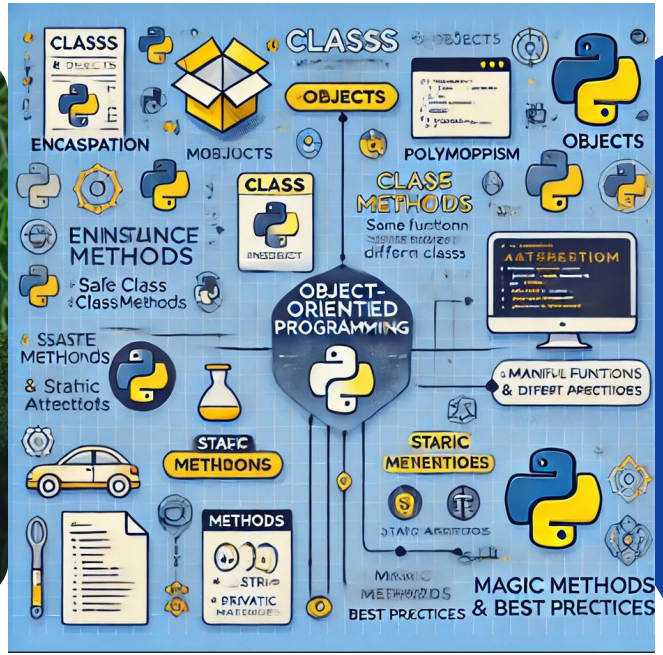
- It takes a function as an argument.
- It adds a functionality in inner function
- It returns the inner function object

@Symbol with a decorator

- Instead of assigning the function call to a variable, Python provides a much more elegant way to achieve this functionality using the @ symbol.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def ordinary():
    print("I am ordinary")

decorated_func = make_pretty(ordinary)
decorated_func()

Output:
I got decorated
I am ordinary
```

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

@make_pretty
def ordinary():
    print("I am ordinary")

ordinary()
```



Generators

- ❑ Basic Generators using for loop
- ❑ Generator using Function
- ❑ Generator using Class
- ❑ Memory Efficiency of Generators.
- ❑ Limitation of Generator

Basic Generator

- Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.
- Syntax : `(expression for item in iterable)`

List Comprehension

```
l1 = [1, 2, 5, 7, 2, 3]
output_l1 = [x ** 2 for x in l1]

print(type(output_l1)) ## <class 'list'>
print(output_l1) ## [1, 4, 25, 49, 4, 9]
```

Generator

```
l1 = [1, 2, 5, 7, 2, 3]
generator_example = (x ** 2 for x in l1)

print(type(generator_example)) ## <class
'generator'>
```

Generator using Function

- If there is **yield** keyword instead of **return** in a function, then the function is a generator

```
def my_generator(n):
    value = 0
    while value < n:
        yield value
        value += 1

k = 5
gen = my_generator(k)

for num in gen:
    print(num)
```

Generator using Class

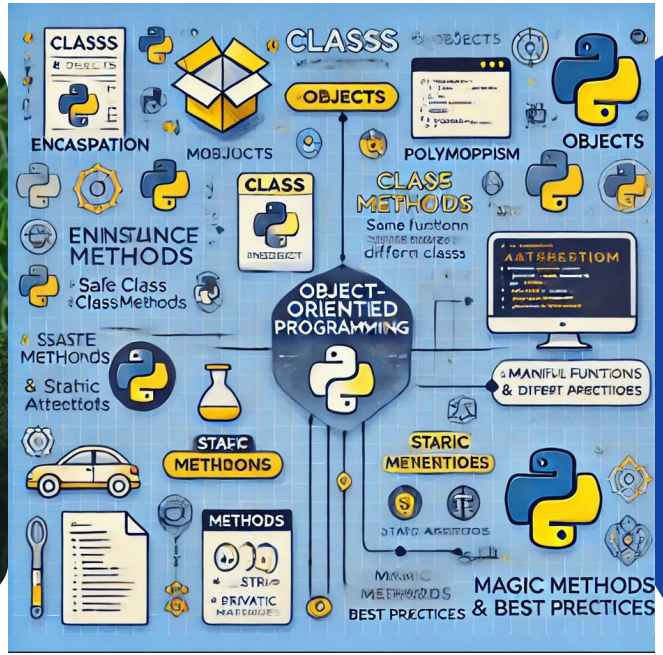
- Dunder method : __iter__ and __next__ are used to create a generator.

```
class Counter:
    def __init__(self, max):
        self.max = max
        self.current = 0

    def __iter__(self): ## This dunder method makes the Class iterable
        return self

    def __next__(self): ## This generates the next value of the Generator
        if self.current < self.max:
            num = self.current
            self.current += 1
            return num
        else:
            raise StopIteration   # Signals the end of the iteration

# Using the generator class
counter = CountUpTo(5)
for num in counter:
    print(num)
```

Generator

Characteristics of Generator

- Lazy evaluation in a generator means that values are produced only when requested, not all at once upfront.
- **No Random Access :** You have to go sequentially to access an element in generator.
- **No backtracking :** Once you move past a value, you can’t go back unless you recreate the generator.



Error Handling and Debugging

- ❑ Exception Handling
- ❑ Built-in Exceptions
- ❑ Debugging

Error Handling and Debugging

- The **try** block lets you test a block of code for errors.
- The **except** block lets you handle the error.
- The **else** block lets you execute code when there is no error.
- The **finally** block lets you execute code, regardless of the result of the try- and except blocks.

```
try:
   print("Hello")
except:
   print("Something went wrong")
else:
   print("Nothing went wrong")
finally:
   print("Everything is completed regardless of error or no error")
```

```
Hello
Nothing went wrong
Everything is completed regardless of error or no error
```

```
try:
   print(x) # This will cause an error since x is not defined
except:
   print("Something went wrong")
else:
   print("Nothing went wrong")
finally:
   print("Everything is completed regardless of error or no error")
```

```
Something went wrong
Everything is completed regardless of error or no error
```

Error Handling Examples

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
    else:
        print(f"Result: {result}")
    finally:
        print("Operation attempted.")



divide(10, 0)
```

```
Error: Cannot divide by zero.

Operation attempted.
```

```
def square_number(x):
    try:
        return int(x) ** 2
    except ValueError:
        print("Please enter a valid integer.")


square_number(2) # output: 4
square_number('hello') # output : Please enter
a valid integer.
```

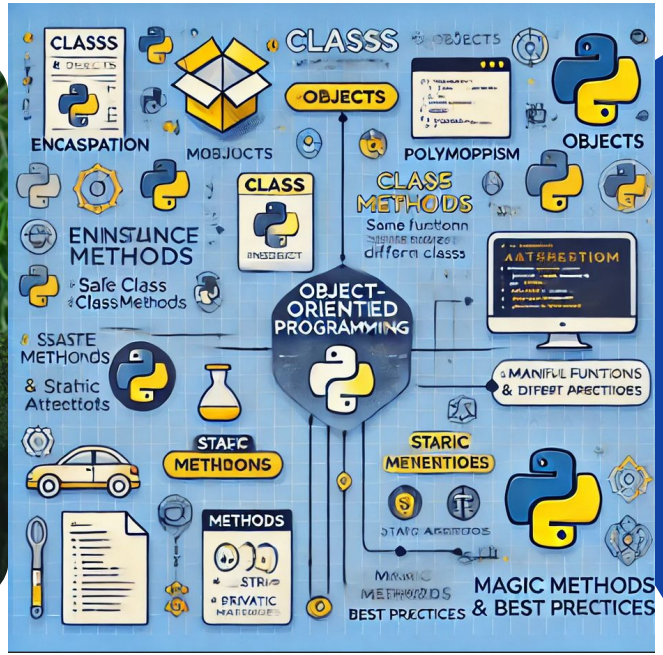
Custom Exception

- Custom Exception inherits from Base Class Exception

```
class NegativeNumberError(Exception):
    """Exception raised when a negative number is provided."""
    pass


def square_root(x):
    if x < 0:
        raise NegativeNumberError("Cannot take square root of a negative number.")
    return x ** 0.5


try:
    result = square_root(-9)
except NegativeNumberError as e:
    print(f"Custom Error: {e}")
```

When to use Exception Handling?

- Handle Runtime Errors like ValueError,
- Working with external resources like File Handling (Reading CSV Files)
- Prevent Crashes in Large Applications (Especially when there is used of APIs)



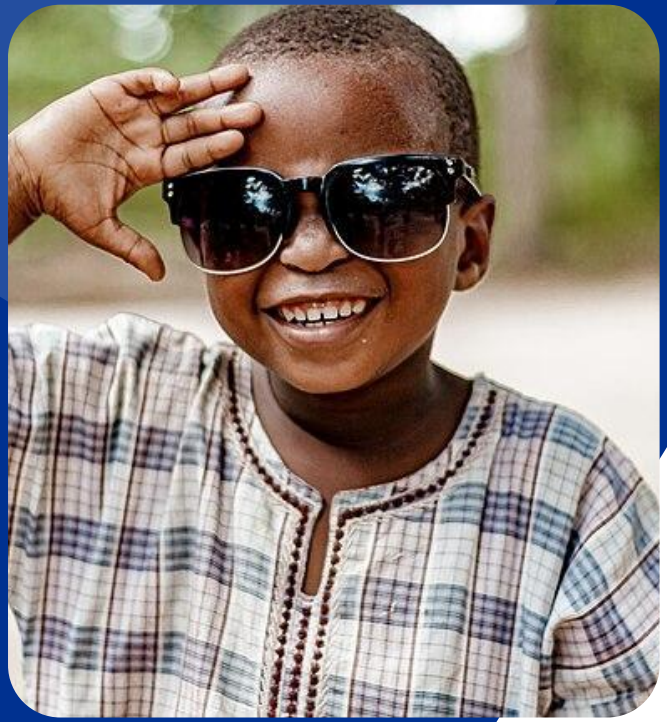
Logging

- 5 Levels of Logging

Logging

- Logging in Python is a way to record messages that describe what your program is doing.
- These messages can help you debug, monitor behavior, or just understand what's going on under the hood.

```
import logging

# Basic configuration
logging.basicConfig(level=logging.DEBUG)

logging.debug("This is a debug message")
logging.info("This is an info message")
logging.warning("This is a warning message")
logging.error("This is an error message")
logging.critical("This is a critical message")

DEBUG:root:This is a debug message
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```



Thank you for your time today.