Python Programming – Day 1

Focus: Foundations that matter

Goal: Write your first clean Python programs

What is python?

- Python is a **high-level programming language**
- Created by **Guido van Rossum**
- Designed for **readability & simplicity**
- Used by beginners and professionals alike

Guido Van Rossum

Why Python?

Python is popular because it is:

- Easy to learn
- Powerful
- Widely used

Use Cases:

- Backend (FastAPI, Django)
- Automation & scripting
- Data analysis & ML
- Cybersecurity & tooling

Python vs C

Python

- High-level language
- Easy to read & write
- Uses **indentation** instead of braces
- Dynamically typed
- Garbage collected
- Slower execution (but faster development)

C

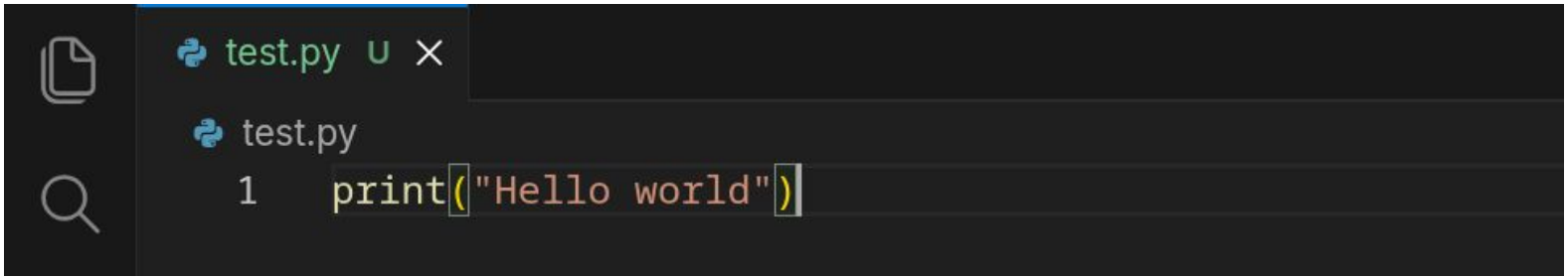
- Low-level language
- Complex syntax
- Uses {} and ;
- Statically typed
- Manual memory management
- Very fast execution

Ways to Use Python

1. **System Python**

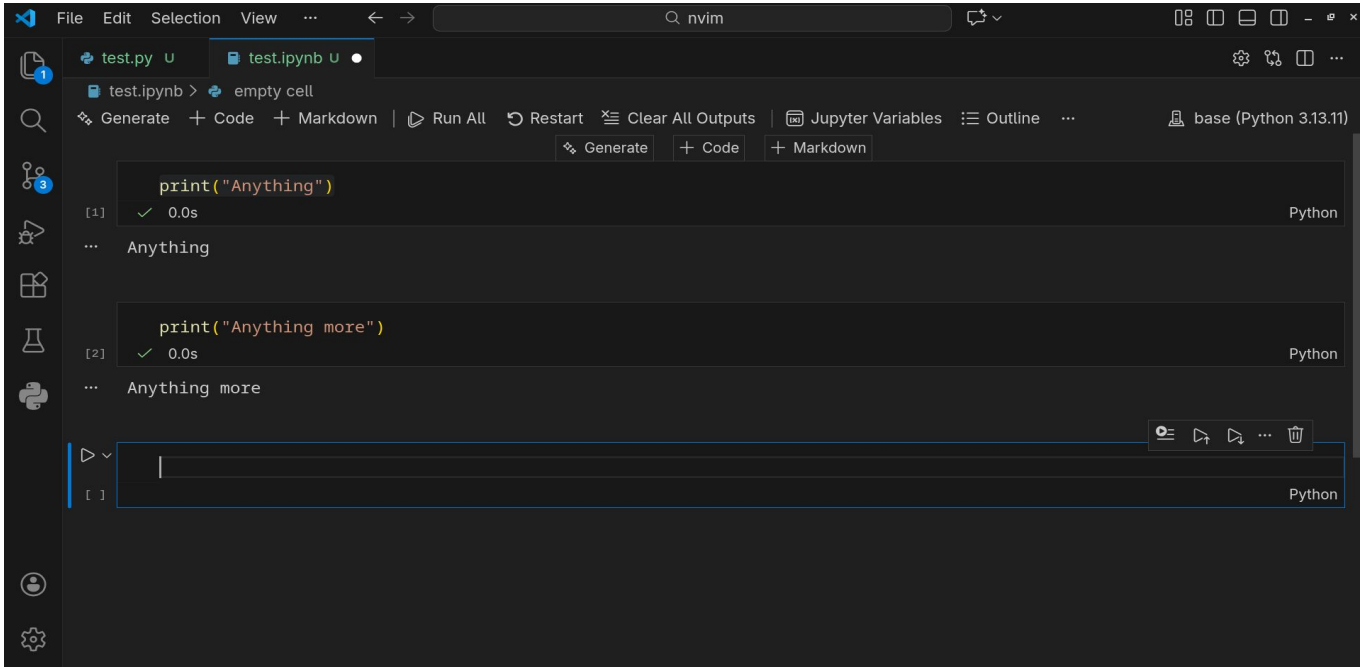
```
(base) ➜  nvim git:(master) python
Python 3.13.11 | packaged by Anaconda, Inc. | (main, Dec 10 2025, 21:28:48) [GCC 14.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2. **IDE**



Ways to Use Python

3. Notebook



What is a Virtual Environment?

A **virtual environment** is:

- An isolated Python workspace
- Keeps dependencies separate per project

Why it matters:

- Avoids version conflicts
- Clean & professional workflow
- Industry standard

PIP vs UV

Pip (Most used)

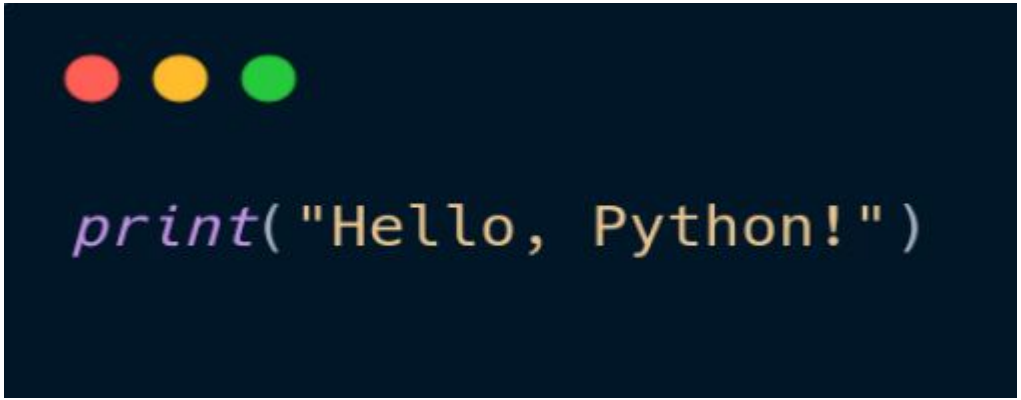
- Traditional Python package manager
- Widely used
- Slower dependency resolution

Uv (less in used)

- Modern & fast
- Written in Rust
- Better dependency handling

Your First Python Code

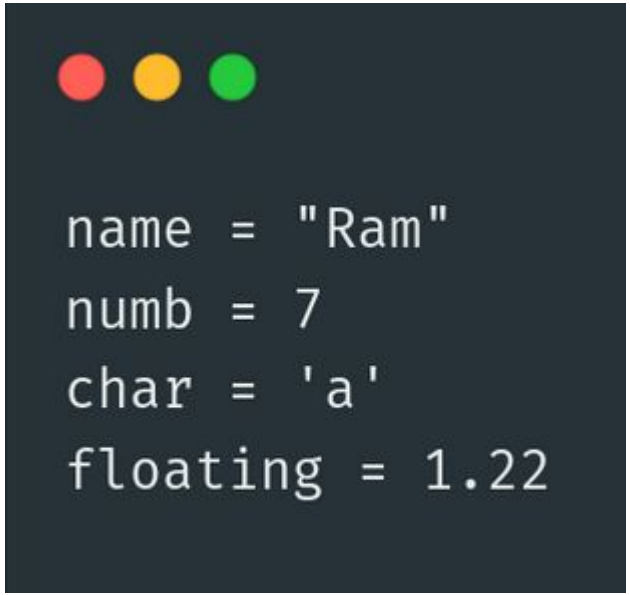
- `print()` outputs text to the screen
- This is your first Python program 🎉



Variables && Data Types

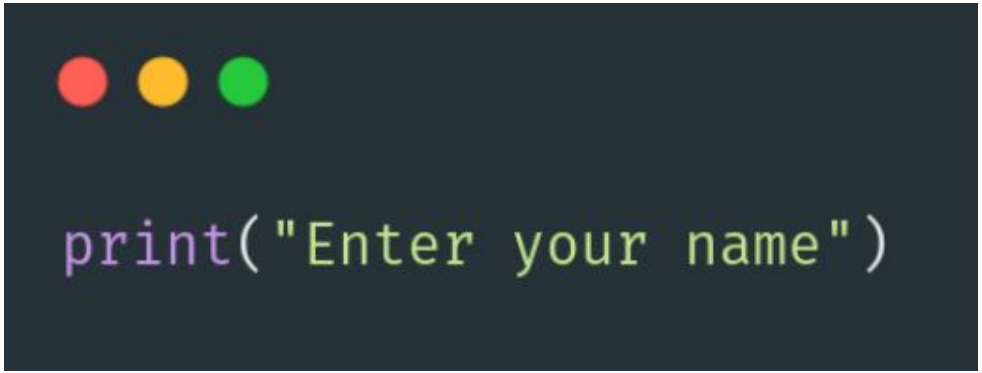
- Variables store data.

- Common Python data types:
  - `int` → 10, -5
  - `float` → 3.14, 0.5
  - `str` → "hello"
  - `bool` → True / False



Input & Output

- Output:

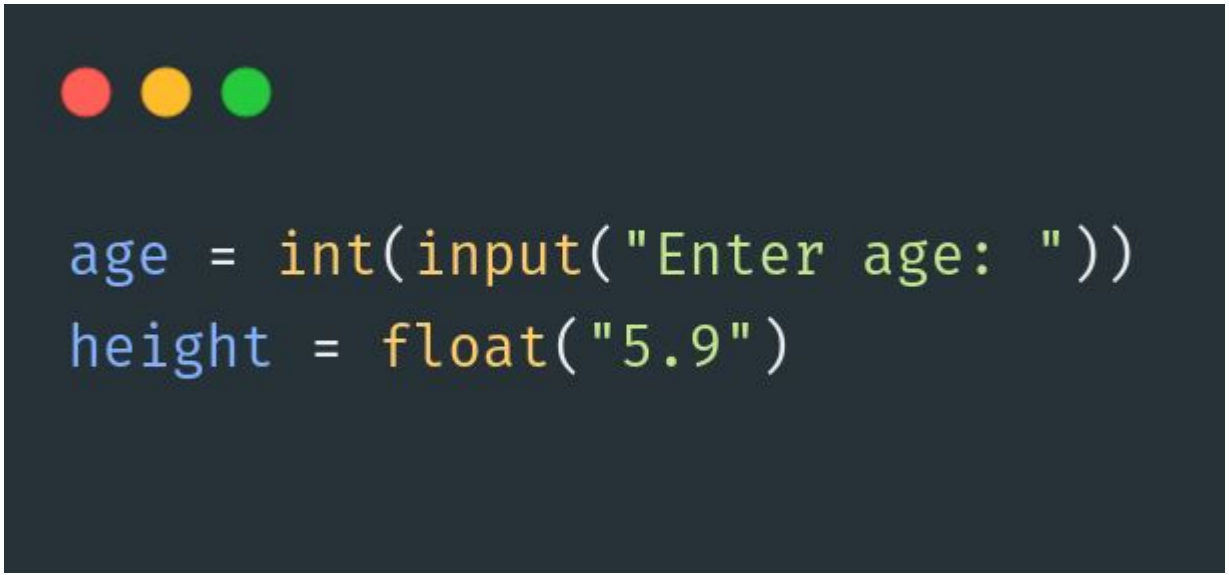


Input & Output

- Input:

Type Casting

- Convert one type to another



Control flow

- Control flow is the way to **control the order in which code executes** in a program.

Key Components:

- Conditional statements: `if`, `elif`, `else`
- Loops: `for`, `while`
- Loop controls: `break`, `continue`, `pass`

Control flow (if-elif-else)

Logical Operators

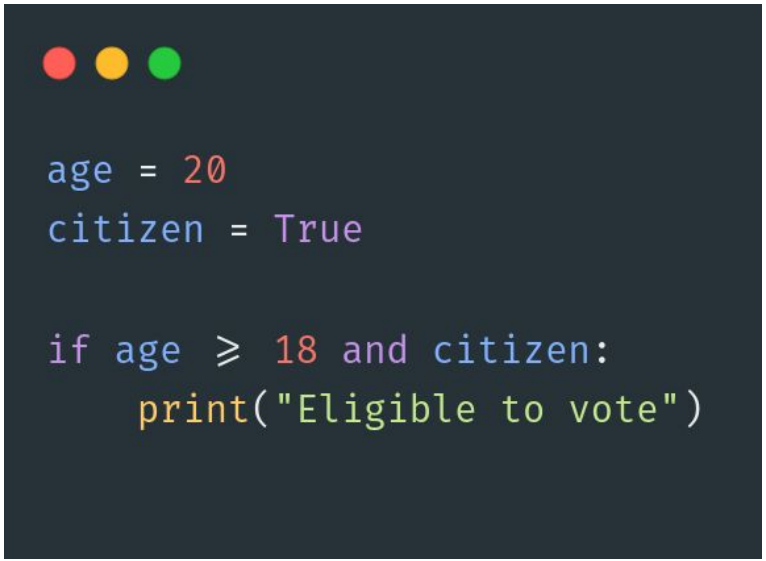
What are Logical Operators?

- Logical operators are used to **combine or modify conditions** in Python.

Common Operators:

- `and` → True if **both** conditions are True
- `or` → True if **at least one** condition is True
- `not` → Reverses a condition

Logical Operators



Practice Task

- **Even / Odd Checker**
  - 1. Prompt user to enter a number
  - 2. Read the number and store in variable NUM
  -
  - 3. IF NUM modulo 2 equals 0 THEN
  - PRINT "Number is Even"
  - ELSE
  - PRINT "Number is Odd"

HomeWork Question

Write a Python program that:

1. Takes three numbers as input from the user
2. Determines and prints:
    - The **largest number**
    - Whether it is **even or odd**

-

Day 2

By Pujan Neupane

Loops, Data Structures & Built-ins

By Pujan Neupane

What Are Loops?

Loops let us **repeat actions** without writing the same code again.

Why loops matter:

- Reduce code repetition
- Handle large data easily
- Automate repetitive tasks

For Loop

Used to iterate over a sequence (list, string, range, etc.)

Key idea:

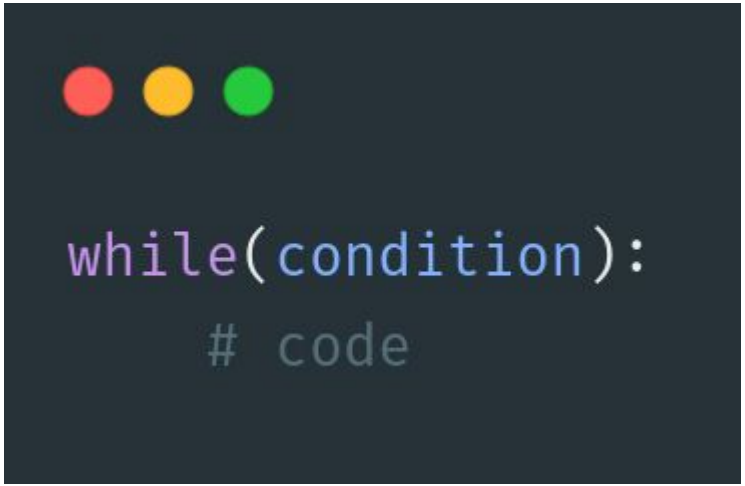
- Runs **for each item** in a sequence

while Loop

Runs **while a condition is true**

Key idea:

- Condition-based repetition



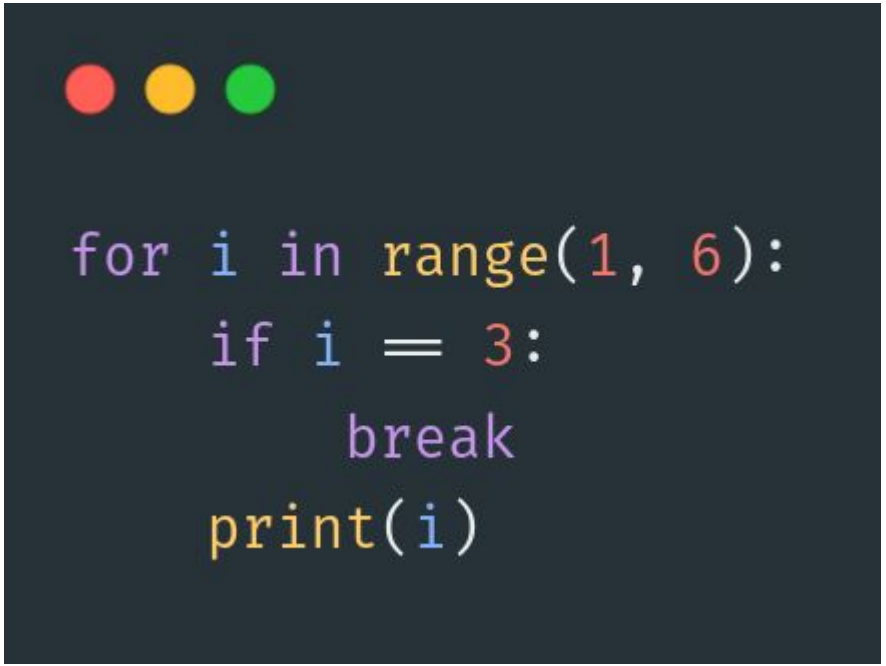
Loop Control Keywords

Break: Stops the loop immediately

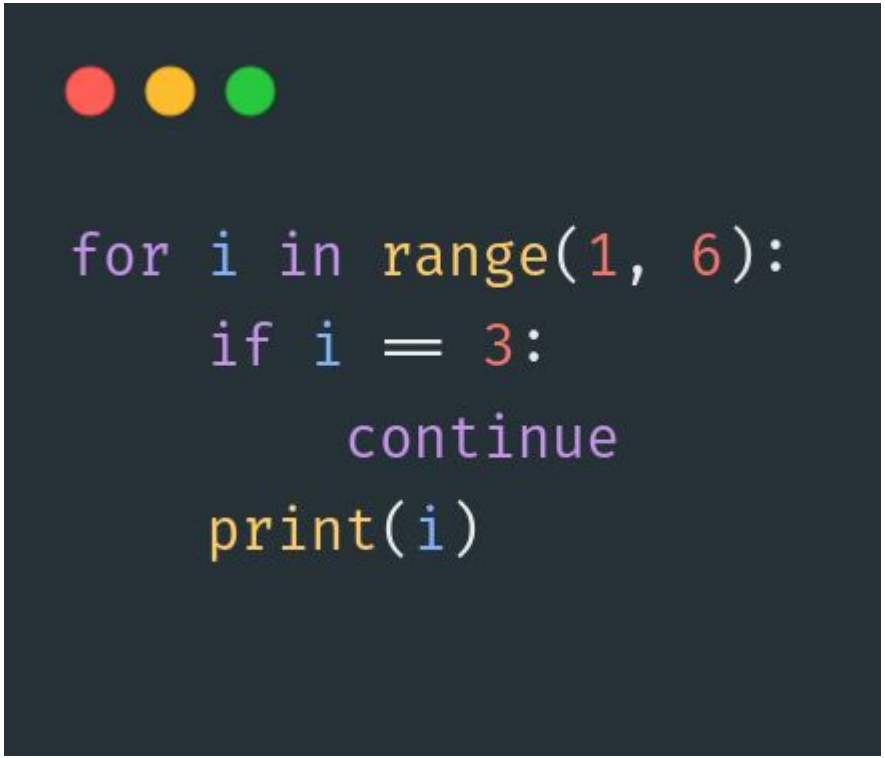
Continue : Skips current iteration, moves to next

Pass : Does Nothing

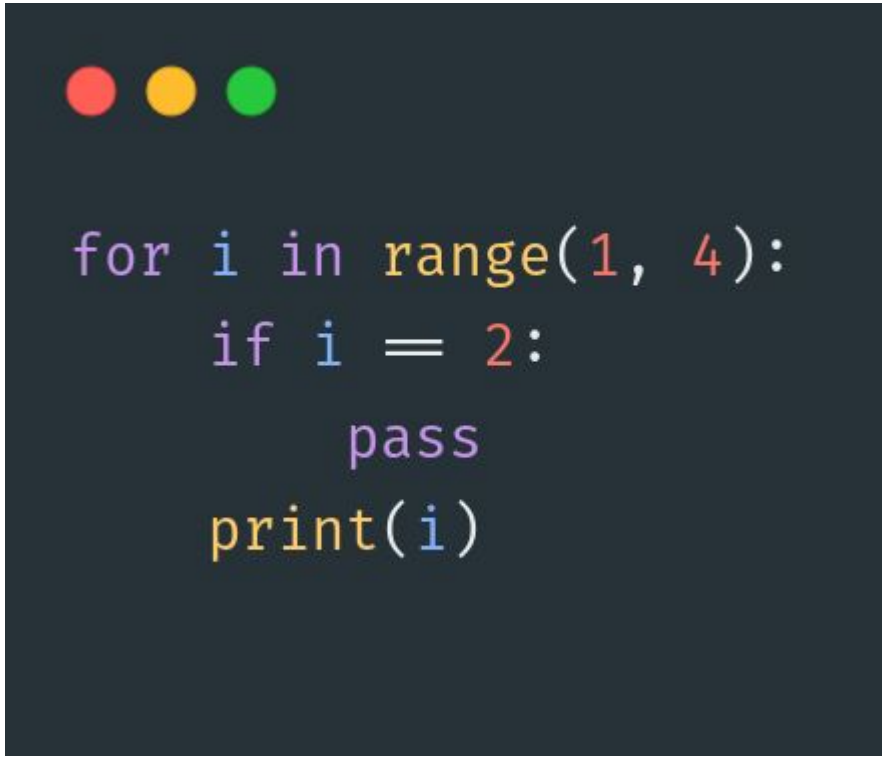
Break: Control Keywords



Continue : Control Keywords



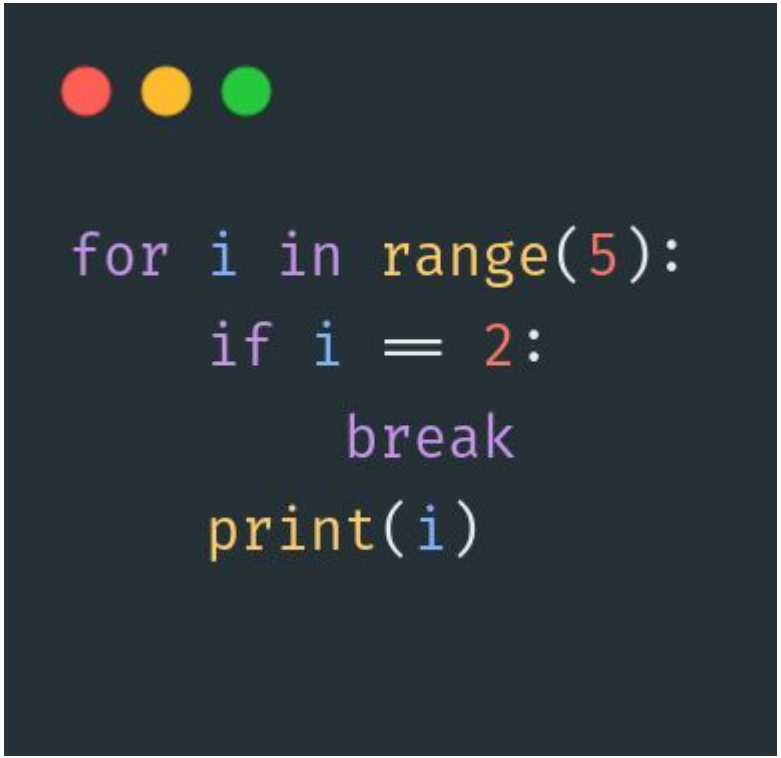
pass : Control Keywords



```
for i in range(1, 4):
    if i == 2:
        pass
    print(i)
```

Question: What will be printed by this code?

a. 0 1 2
b. 0 1
c. 1 2 3
d. 0 1 2 3 4



```
for i in range(5):
    if i == 2:
        break
    print(i)
```

Task

- Write a program to print **all numbers from 1 to 10 except 5**, and stop **if the number is 8**.

Data Structures in Python

Used to **store and organize data**

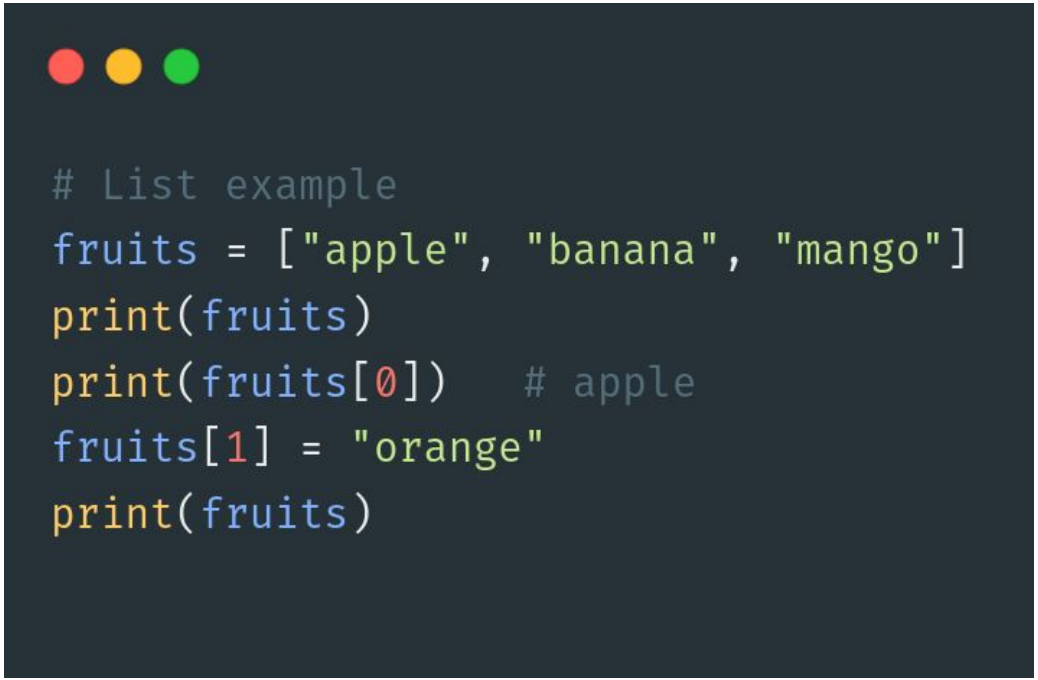
Main types:

- List
- Tuple
- Set
- Dictionary

List

- Ordered collection
- Mutable (can be changed)

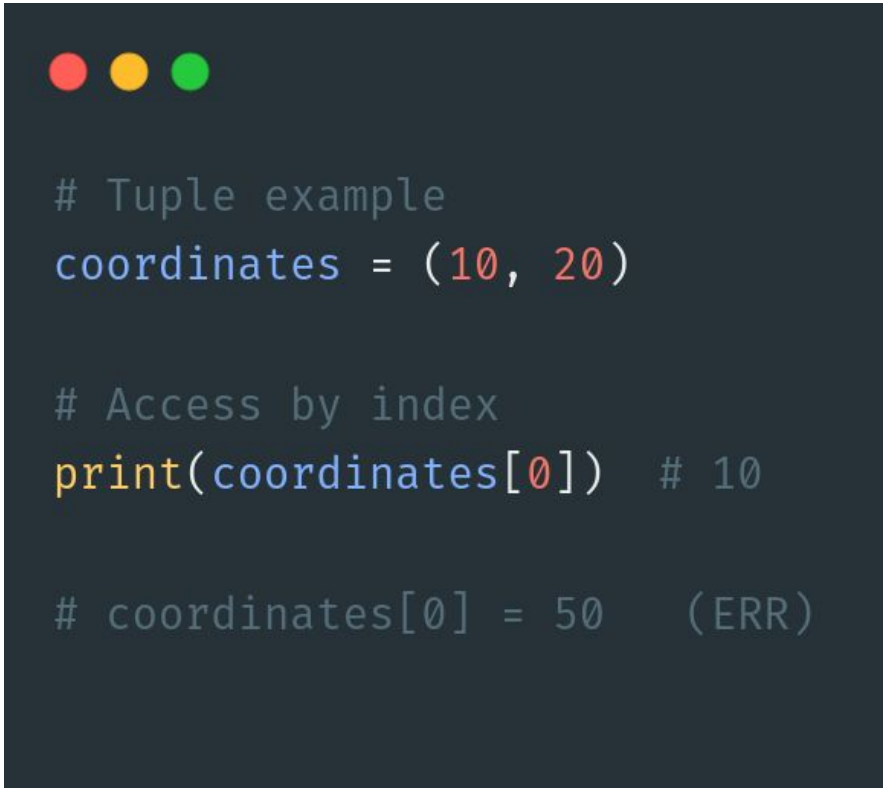
List: Example



```
# List example
fruits = ["apple", "banana", "mango"]
print(fruits)
print(fruits[0])   # apple
fruits[1] = "orange"
print(fruits)
```

Tuple

- Ordered collection
- Immutable (cannot be changed)



```python
# Tuple example
coordinates = (10, 20)

# Access by index
print(coordinates[0])  # 10

# coordinates[0] = 50   (ERR)
```

Set

- Unordered collection
- No Duplicates
- mutable

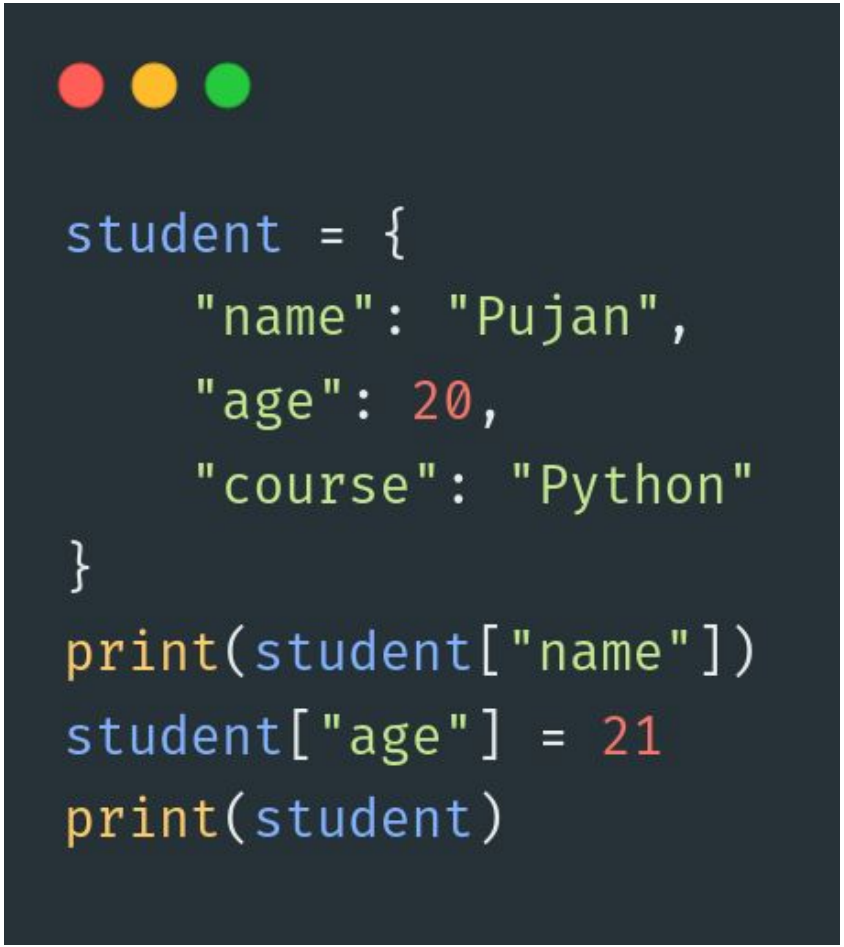
```python
# Set example
numbers = {1, 2, 3, 3, 4}

print(numbers)  # duplicates removed automatically

# Membership testing
print(3 in numbers)   # True
print(5 in numbers)   # False
```

Dictionary

- Key–Value Pairs
- Mutable

Used for:

- Fast lookups
- Mapping relationships



Indexing & Slicing

Indexing

- Access single element

Slicing

- Access a range of elements

Works with:

- Lists
- Tuples
- Strings

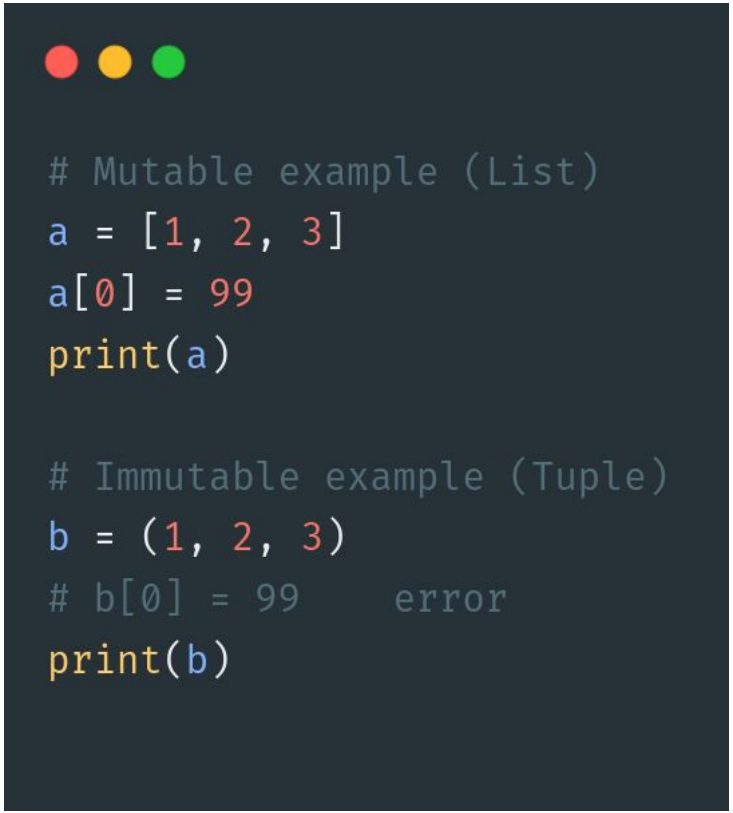
Mutability vs Immutability

Mutable

- Can change after creation
- Example: List, Dictionary

Immutable

- Cannot change after creation
- Example: Tuple, String



```
# Mutable example (List)
a = [1, 2, 3]
a[0] = 99
print(a)

# Immutable example (Tuple)
b = (1, 2, 3)
# b[0] = 99    error
print(b)
```

Task: CODE

Question: Create a **list of fruits**, a **tuple of colors**, and a **set of numbers**. Then:

- Add a new fruit to the list
- Try changing a tuple element (observe error)
- Add a new number to the set

Built-in Functions

`len()` – count items (already done)

`type()` – check data type (already done)

`range()` – generate numbers (already done)

`sorted()` – sort data

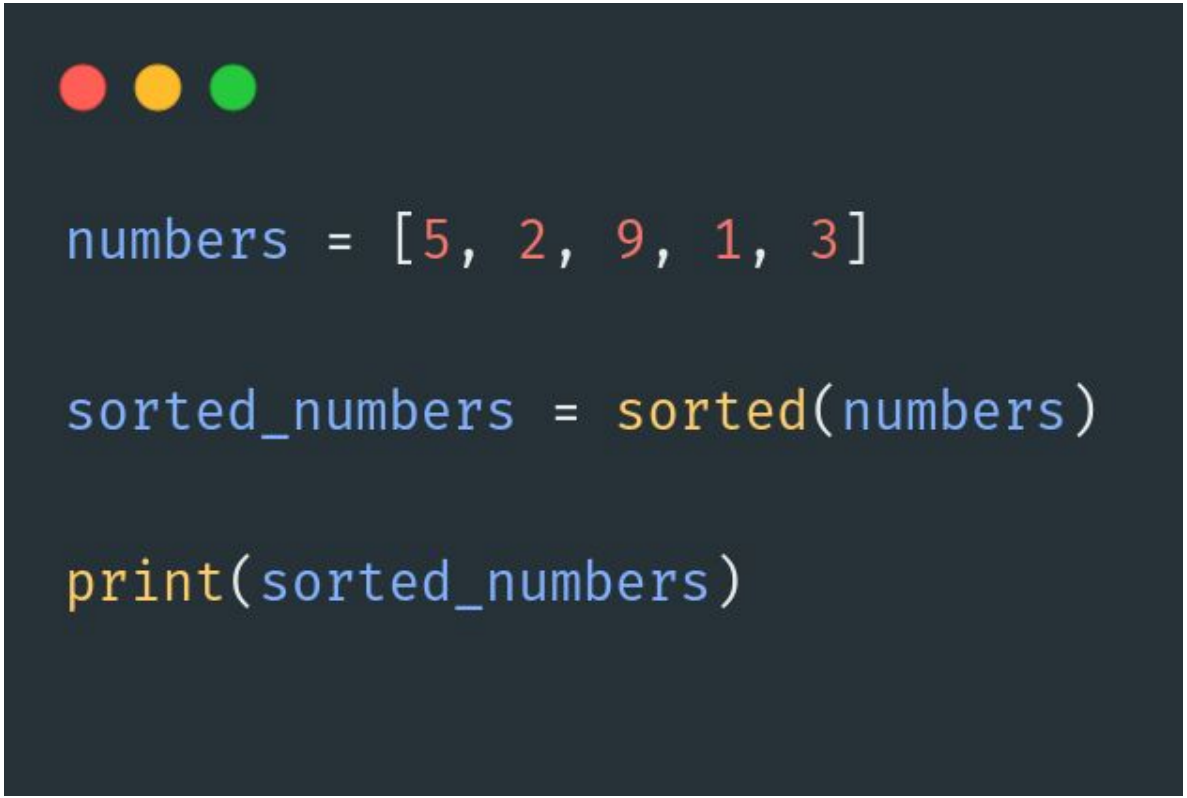
`enumerate()` – index + value

`sum()` – total

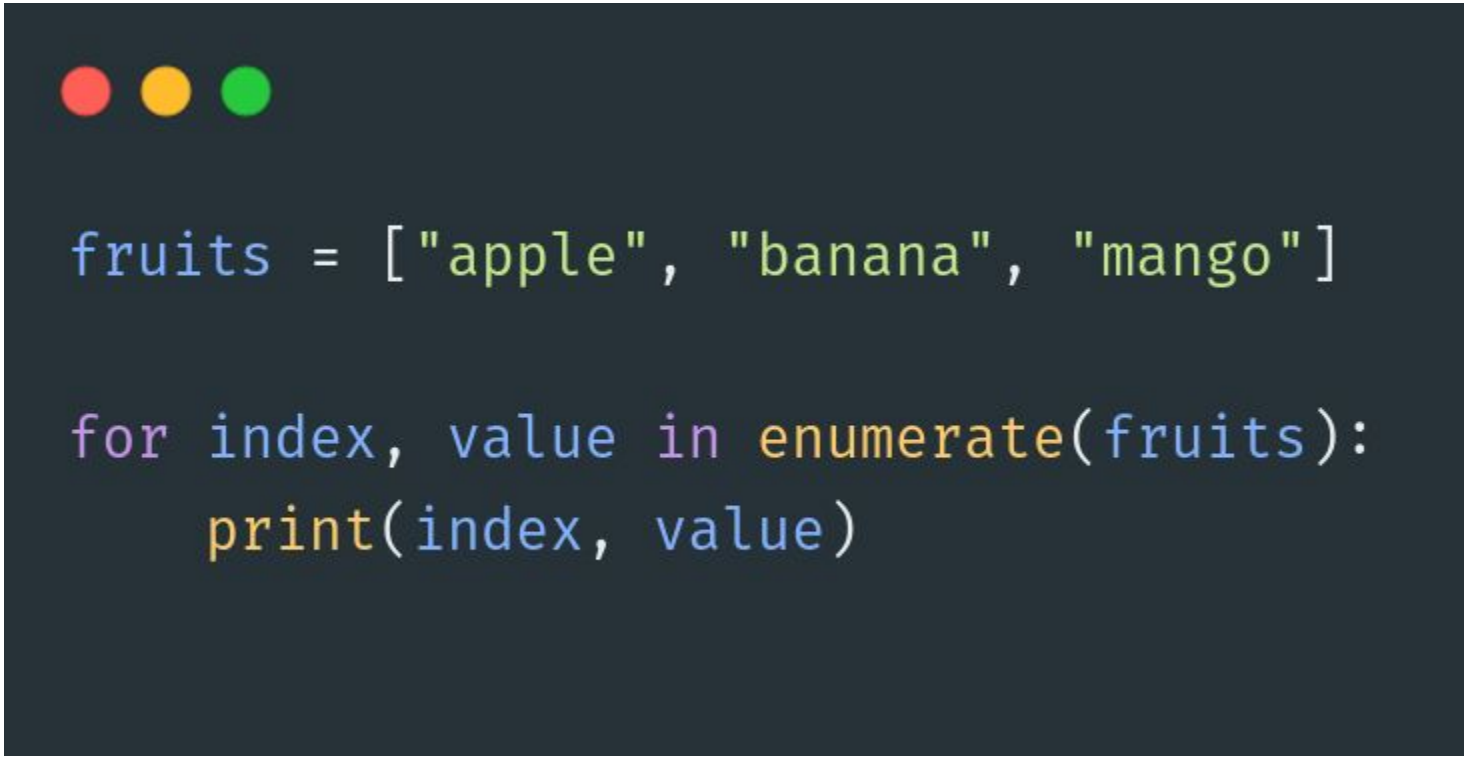
`min()` – smallest value

`max()` – largest value

`sorted()` – sort data

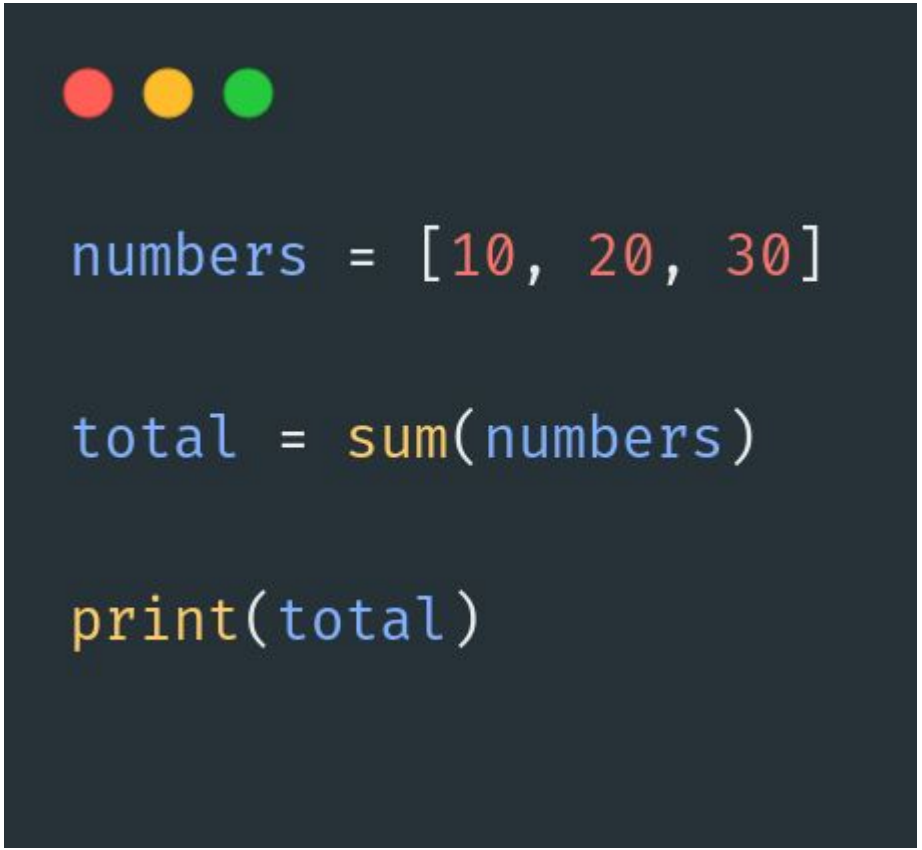


`enumerate()` – Index + Value

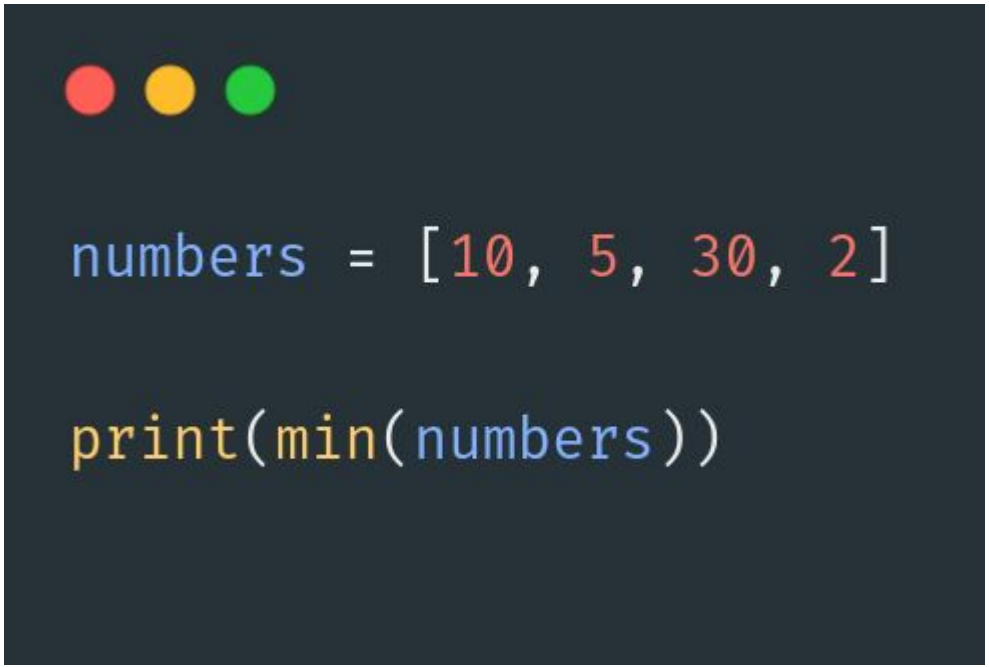


```
fruits = ["apple", "banana", "mango"]

for index, value in enumerate(fruits):
    print(index, value)
```

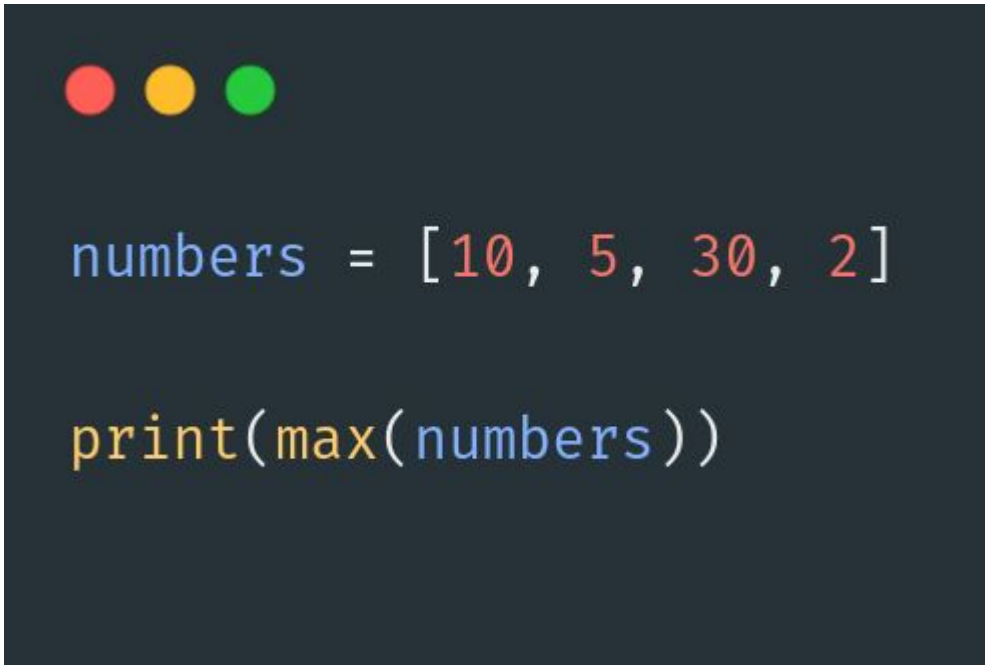
sum() – Total



`min()` – Smallest Value from the list



`max()` – Largest Value from the list



Task: CODE

Question: Given a list of marks `[40, 55, 70, 85]`, write a program to print:

- Total marks
- Highest mark
- Lowest mark
- Marks sorted in ascending order

Day 3

By Pujan Neupane

Function && Clean Code

By Pujan Neupane

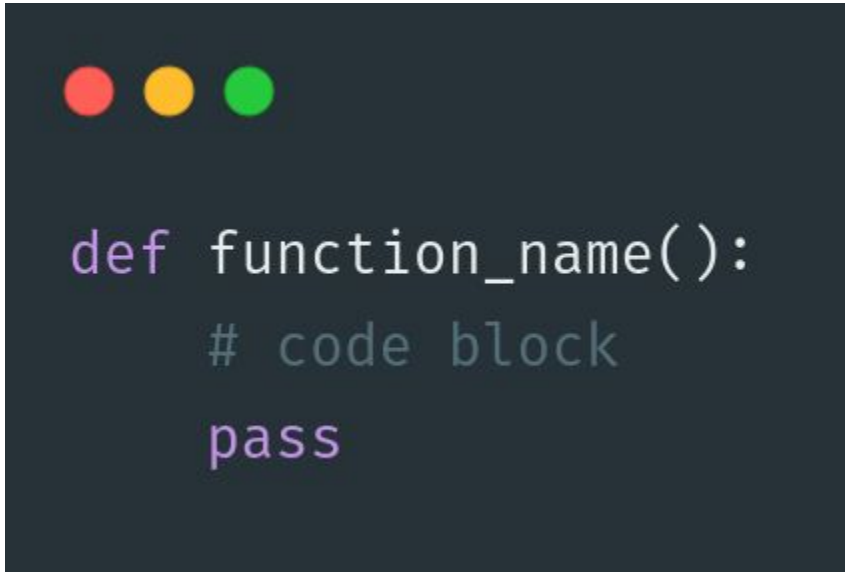
What is a Functions?

A **function** is a reusable block of code that performs a specific task.

Why functions matter:

- Reduce code repetition
- Improve readability
- Easier debugging and maintenance

Defining a Function



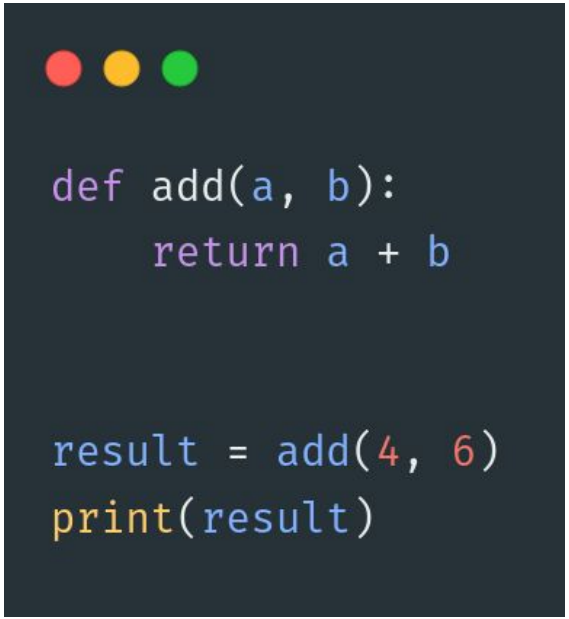
Example of a Function

Parameters & Return Values

- Parameters allow data to be passed into a function.

Key points:

- `return` sends data back to the caller
- A function can return multiple values



Default Arguments

- Default arguments provide a fallback value.

- Why use defaults?
  - Prevent errors
  - Make functions flexible

Keyword Arguments

- Arguments passed using parameter names.

*args (Variable Arguments)

- Used to pass multiple positional arguments.

**kwargs (Keyword Arguments)

- Used to pass multiple keyword arguments.

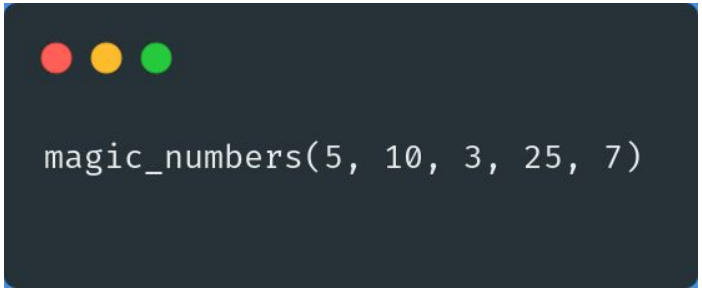
```
def profile(**kwargs):
  for key, value in kwargs.items():
    print(key, value)


profile(name="Pujan", role="Student")
```

Question: Use args and kwargs

- **Magic Number Machine**

  Write a function that:

  - Accepts **any number of integers** using `*args`
  - Prints:
    - the **total count** of numbers
    - the **sum**
    - the **largest number**



Question: Use kwargs

- **Character Profile Builder**

  Write a function that:

  - Accepts **character details** using `**kwargs`
  - Prints a **game-style profile**

```
character(name="Pujan", level=5, power="Python", hp=100)
```

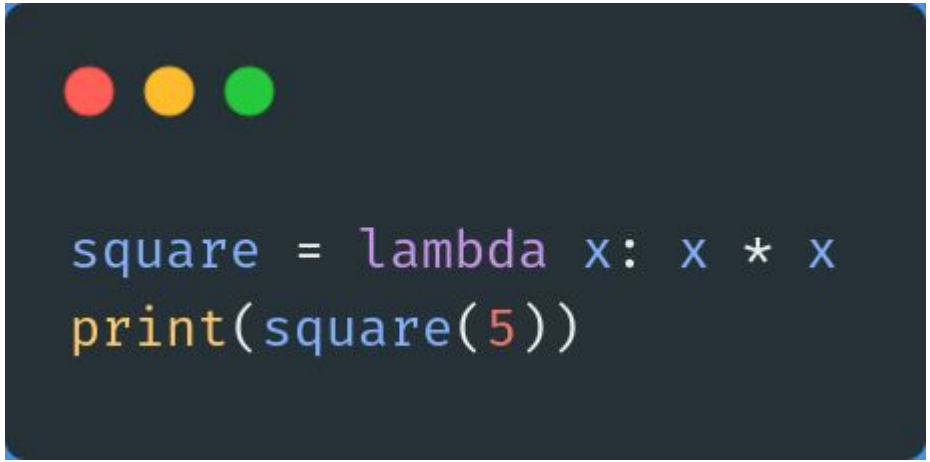
```
Character Profile
Name   : Pujan
Level  : 5
Power  : Python
HP     : 100
```

Lambda Functions

- Anonymous one-line functions.

Lambda Functions example:

- Anonymous one-line functions.



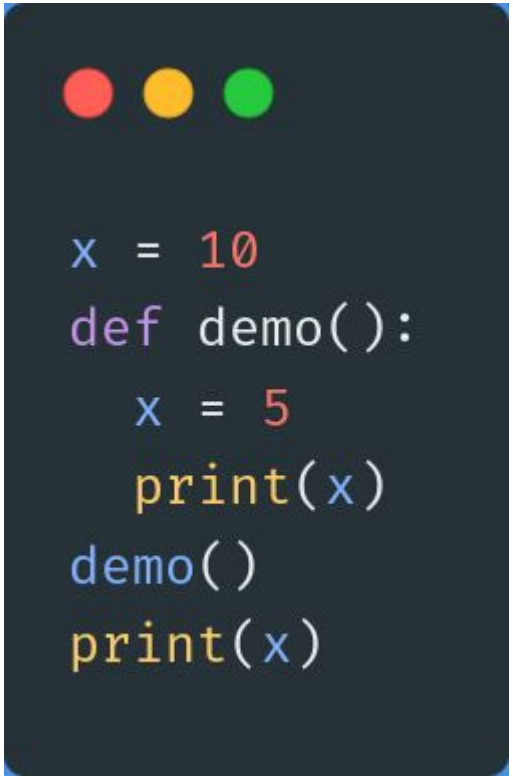
Scope (Local vs Global)

Local variables:

- Defined inside a function
- Accessible only within that function

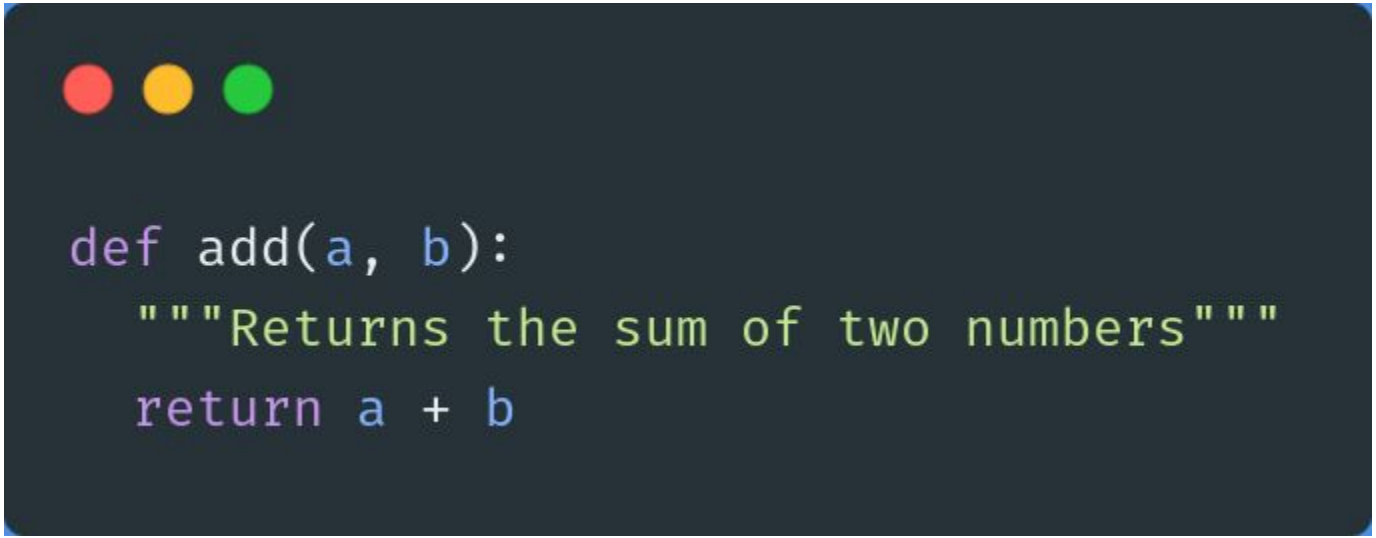
Global variables:

- Defined outside functions
- Accessible everywhere



Docstrings

- Docstrings describe what a function does.



Function Naming

Good naming rules:

- Use lowercase
- Use underscores
- Be descriptive

Examples:

- `calculate_total()`
- `get_user_data()`

Avoid:

- `func1()`
- `x()`

Reusability & Best Practices

Best practices:

- Keep functions small
- Do one task per function
- Avoid hardcoding values

Reusable functions:

- Save time
- Improve scalability

OOP

- By Pujan Neupane

What is OOP?

Object-Oriented Programming (OOP) is a way of structuring programs using **classes** and **objects**.

Why OOP?

- Organizes code better
- Improves reusability
- Models real-world entities

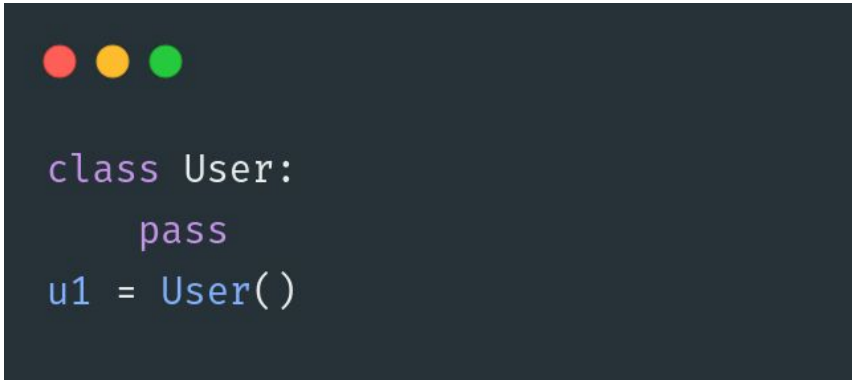
Class & Object

Class

A blueprint for creating objects.

Object

An instance of a class.



`__init__` Method

- Special method (constructor)
- Runs automatically when an object is created

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

Instance vs Class Variables

- **Instance variable** → unique per object
- **Class variable** → shared among all objects

Theory Question:

- Why do we use the `__init__` method?
- What is the difference between a class and an object?

Coding Question:

- Create a `Car` class with attributes: `brand`, `model`, `year`.
  - Create 2 objects and print their details.
  - Add a method `age()` to the `Car` class to calculate the age of the car (current year - year of the car).

soln:

```python
# Car Class Example
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def display(self):
        print(f"{self.brand} {self.model} ({self.year})")

    def age(self, current_year):
        return current_year - self.year


# Objects
car1 = Car("Toyota", "Corolla", 2018)
car2 = Car("Honda", "Civic", 2020)

car1.display()
car2.display()
current_year = 2025
print(f"Car1 Age: {car1.age(current_year)} years")
print(f"Car2 Age: {car2.age(current_year)} years")
```

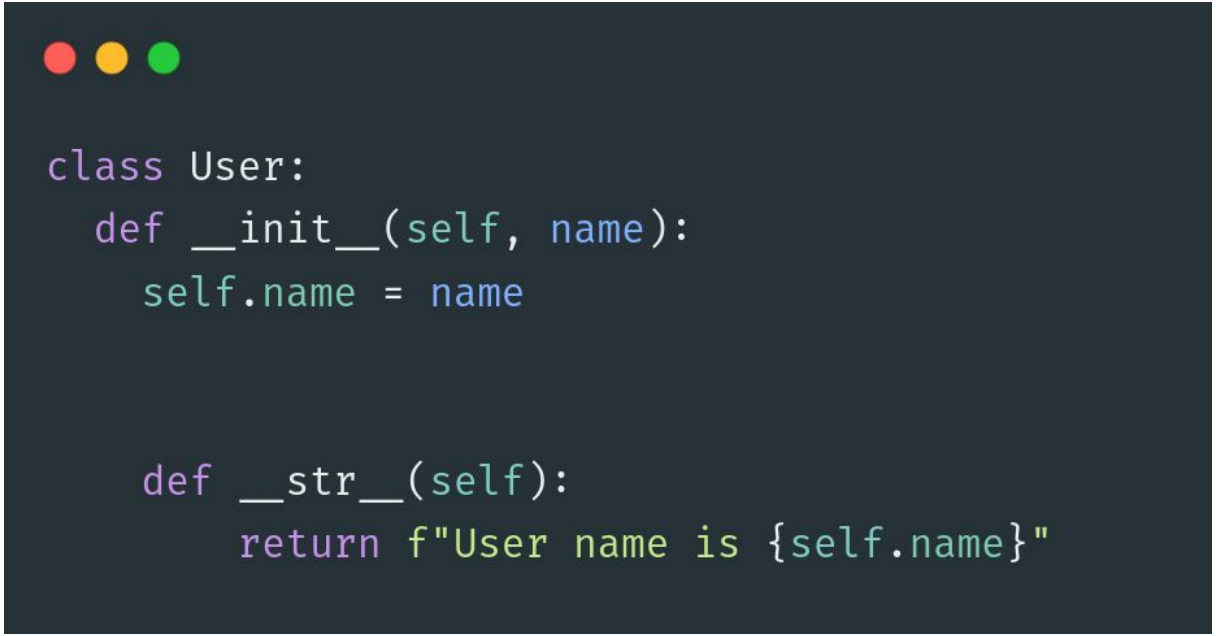
Magic Methods

- Magic methods start and end with double underscores.

- Examples:
  - `__init__`
  - `__str__`
  - `__repr__`

`__str__` Method

- Defines **user-friendly** string output



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


    def __str__(self):
        return f"User name is {self.name}"
```

`__repr__` Method

- Used for **developer/debugging** representation

```
class User:
  def __repr__(self):
    return f"User(name={self.name})"
```

Question: code

- Create a `Book` class with attributes: `title`, `author`.
    - Use `__str__` to print: `"Book: <title> by <author>"`
    - Use `__repr__` to print developer-friendly format: `"Book(title='<title>', author='<author>')"`

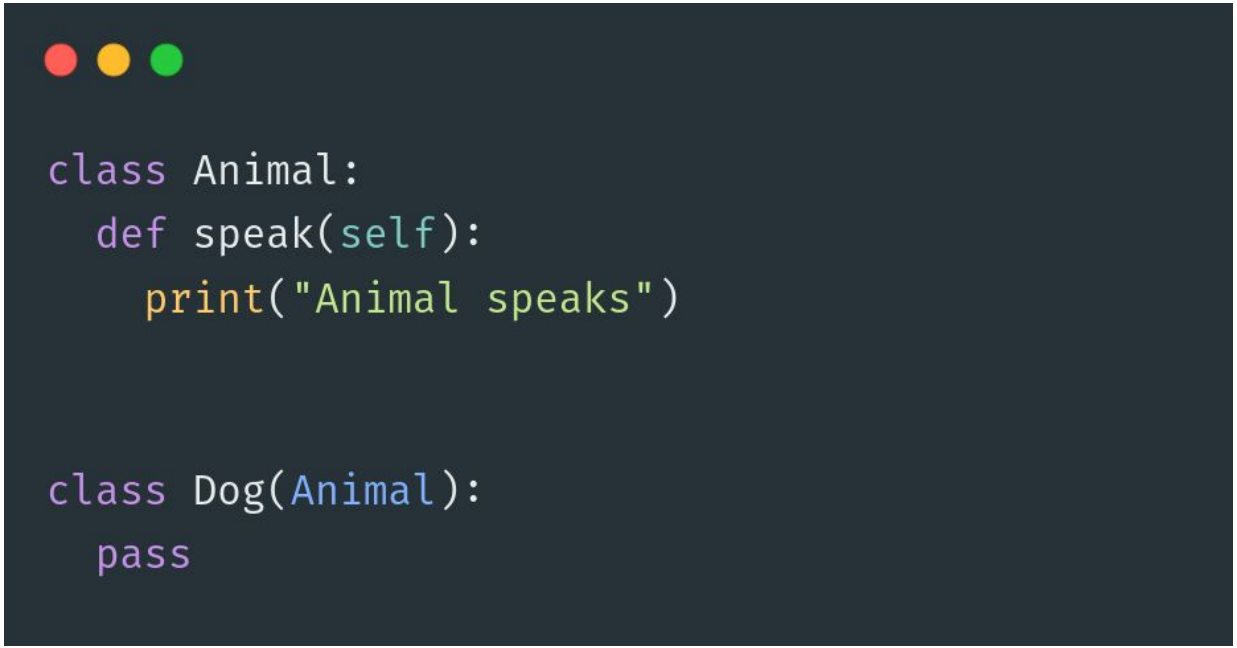
What happens if you print the object without defining `__str__`?

soln:

```python
# Book Class Example
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"Book: {self.title} by {self.author}"

    def __repr__(self):
        return f"Book(title='{self.title}',
author='{self.author}')"


b1 = Book("1984", "George Orwell")

print(b1)       # calls __str__
print(repr(b1)) # calls __repr__
```

Inheritance

- Inheritance allows a class to acquire properties of another class.



```python
class Animal:
  def speak(self):
    print("Animal speaks")


class Dog(Animal):
  pass
```

Polymorphism

- Same method name, different behavior.

```python
class Dog:
  def speak(self):
    print("Bark")


class Cat:
  def speak(self):
    print("Meow")
```

Method Overriding

- Child class modifies parent method.

Operator Overloading

- Redefining operators using magic methods.

Examples:

- `+` → `__add__`
- `==` → `__eq__`

Operator Overloading : Example

```python
class Number:
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        return Number(self.value + other.value)

    def __str__(self):
        return str(self.value)
n1 = Number(10)
n2 = Number(20)
print(n1 + n2)
```

Question:

Create a `Distance` class:

- Attributes: `feet`, `inches`
- Overload the `+` operator to **add two distances**.
- 12 inches = 1 foot (handle overflow properly)
- Add a `__str__` method to display in `"X ft Y in"` format

```
d1 = Distance(5, 8)
d2 = Distance(3, 9)
print(d1 + d2)
```

Files, Errors & Standard Library

- By Pujan Neupane

File Handling – Basics

Reading a Text File:

File Handling – Basics

Write a Text File:

File Handling – Basics

File Modes:

- `r` → Read
- `w` → Write (overwrite)
- `a` → Append
- `rb / wb` → Binary modes

File Handling – Basics

Write in JSON Files

```python
import json


data = {
  "name": "Pujan",
  "role": "Cybersecurity"
}


with open("user.json", "w") as f:
  json.dump(data, f, indent=4)
```

File Handling – Basics

Reading JSON Files

Question

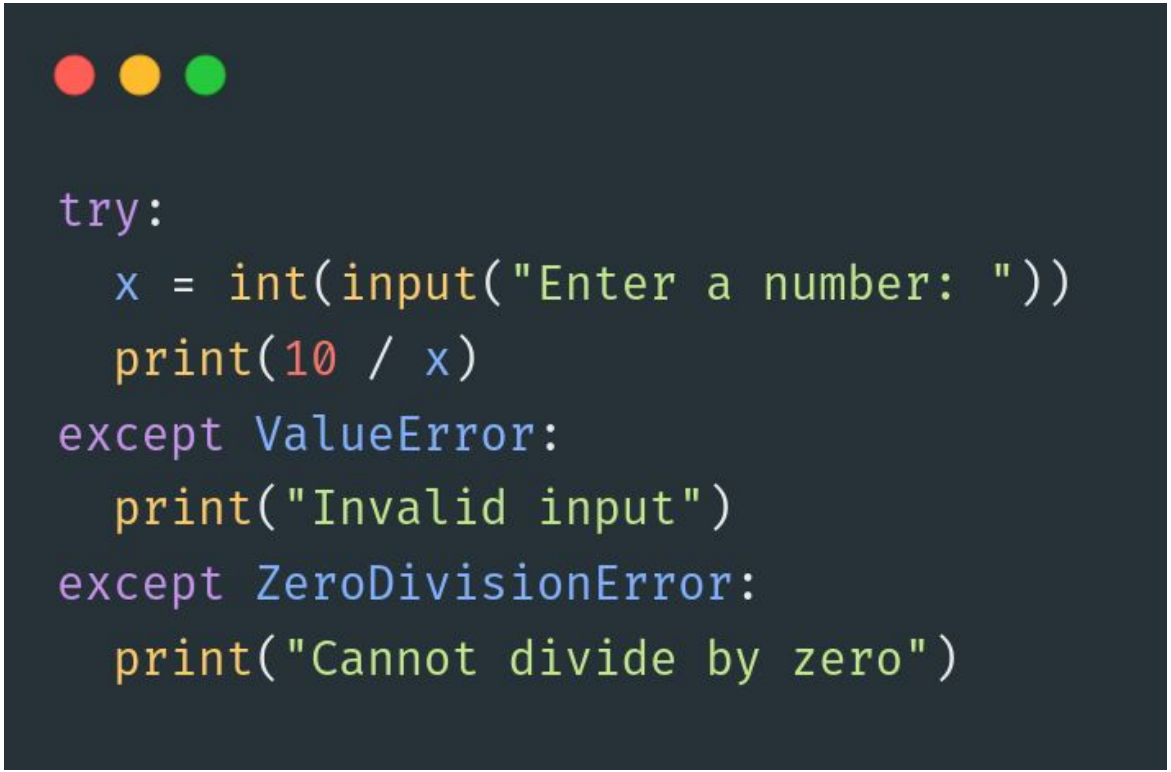
- JSON is used to save the player’s game progress.
- Player data is stored as key–value pairs.
- The data is written to a file so it can be loaded later.

Question

Solution:

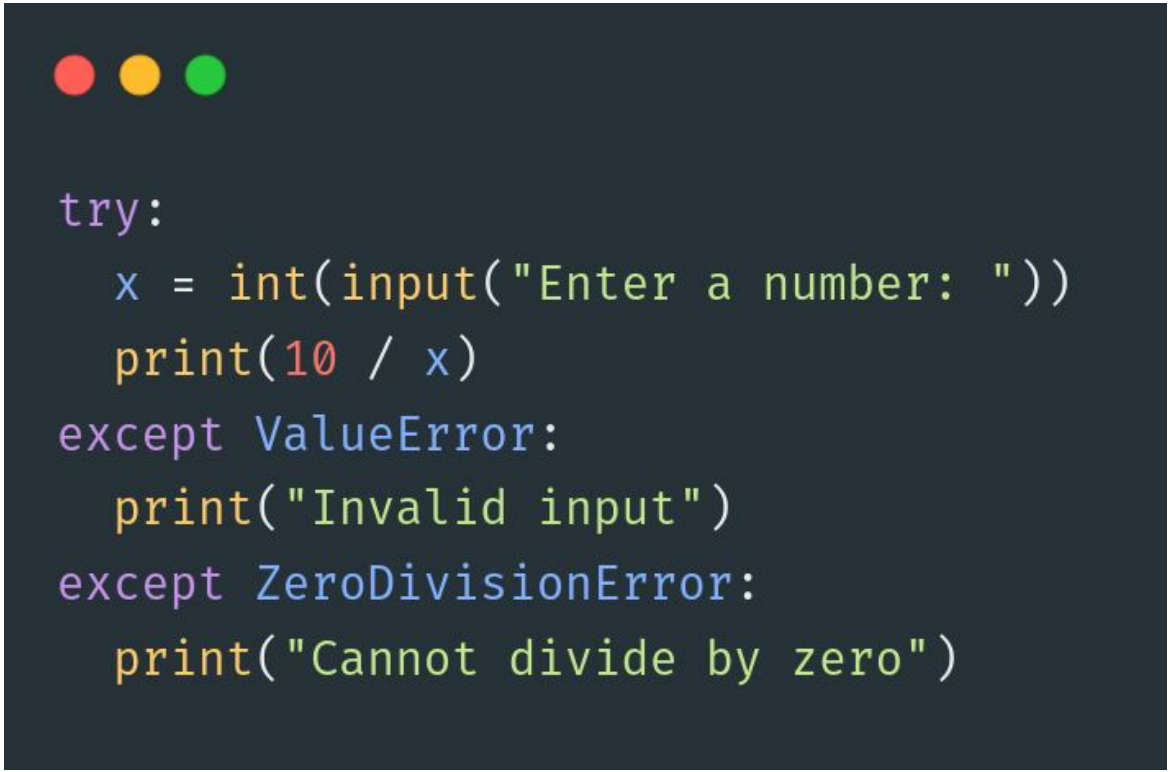
```python
import json

# create player data
player = {
    "name": "Hero",
    "level": 1,
    "score": 0
}

# write to json file
with open("savegame.json", "w") as file:
    json.dump(player, file, indent=4)

print("Game saved successfully!")
```

Exception Handling – Basics



```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ValueError:
    print("Invalid input")
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Exception Handling – Basics



```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ValueError:
    print("Invalid input")
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Exception Handling – Basics

Try

Except

Else

finally

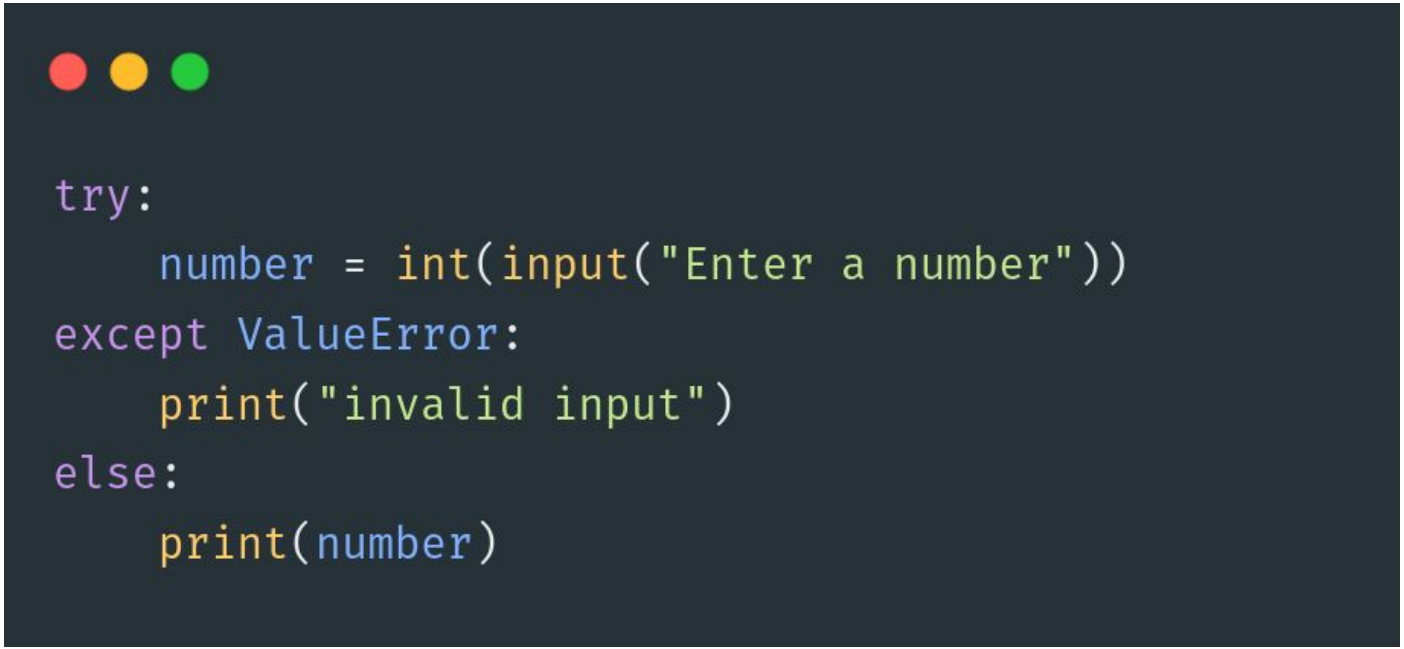
```
try:
  f = open("data.txt")
except FileNotFoundError:
  print("File not found")
else:
  print(f.read())
finally:
  print("Execution finished")
```

Exception Handling – Task

Write a Python program that:

- Takes a number as input
- Converts it to an integer
- Handles **invalid input** using `ValueError`

Exception Handling – Task



```
try:
    number = int(input("Enter a number"))
except ValueError:
    print("invalid input")
else:
    print(number)
```

MODULES:

datetime Module :

Why `datetime`?

- Work with current date & time
- Create timestamps for logs
- Format date/time for humans or systems

Datetime

```python
from datetime import datetime
now = datetime.now()
print(now)
print(now.strftime("%Y-%m-%d %H:%M:%S"))
```

`os` Module (Operating System Interaction)

Why **`os`**?

- Interact with file system
- Manage folders & paths
- Environment-level operations

os Module (Operating System Interaction)

```python
# Get Current Working Directory
import os
print(os.getcwd())
#Create a Directory
os.mkdir("logs")
#Check If File or Folder Exists
print(os.path.exists("logs"))
```

`sys` Module (System-Level Access)

Why `sys`?

- Access Python runtime info
- Read command-line arguments
- Control program execution

`sys` Module (System-Level Access)

```python
import sys
print(sys.argv)
```

Practice – Log Generator

```python
from datetime import datetime
import uuid


log_id = uuid.uuid4()
time = datetime.now()


with open("app.log", "a") as f:
  f.write(f"[{time}] {log_id} - User logged in")
```