Introduction To Big Data Analytics INSY 8413

ADVENTIST UNIVERSITY OF CENTRAL AFRICA

Instructor:

- Eric Maniraguha | eric.maniraguha@auca.ac.rw | LinkedIn Profile

6h00 pm – 8h50 pm

- Monday A-G104
- Tuesday E-G108
- Wednesday A-G104
- Thursday E-G108

2h30 pm – 8h50 pm

- Sunday B- G205

June 2025

Reference reading

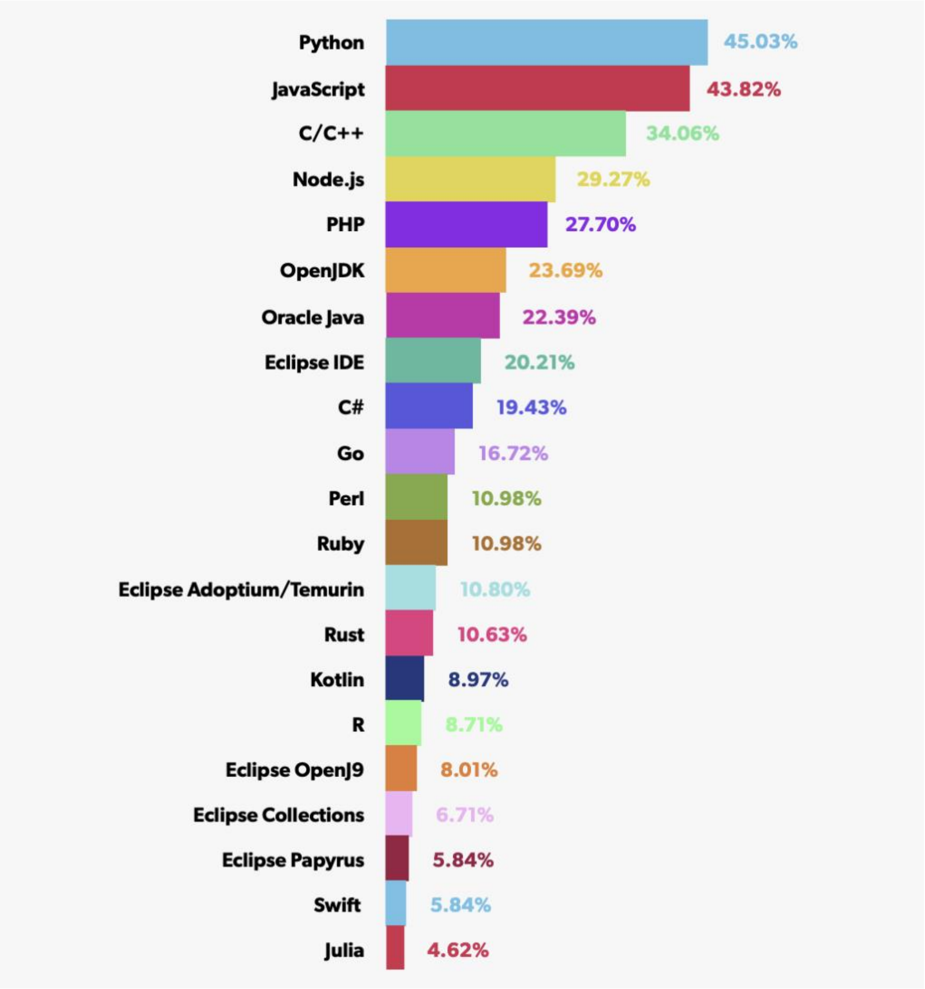
- Python for Data Science - Getting Started
- Python for Beginners – Full Course [Programming Tutorial]
- Installing conda

Learning Objectives

Learning Objectives
By the end of this lecture, students will be able to:

1. **Justify the use of Python for data analytics**
   Understand Python’s popularity in data science due to its readability, rich ecosystem (e.g., Pandas, NumPy, Scikit-learn), strong community support, and cross-platform compatibility.
2. **Set up a Python development environment**
   Successfully install Python and essential data analytics packages using tools like Anaconda or pip.
   Understand the role of environments in package management.
3. **Use Jupyter Notebooks for interactive coding**
   Launch, navigate, and write code in Jupyter Notebooks for combining code, output, and documentation in a single interface ideal for data analysis and storytelling.
4. **Apply core Python programming constructs**
   Write functional Python code using variables, built-in data types (int, float, str, bool), operators, and control structures (if, for, while).
5. **Utilize essential Python data structures**
   Work with lists, tuples, dictionaries, and sets to store, access, and manipulate data effectively.
6. **Create reusable code using functions**
   Define and invoke functions with parameters and return values to support modular, organized programming.
7. **Import and use key Python libraries**
   Use standard modules (math) and essential data science libraries (pandas, numpy, matplotlib) to perform data operations and visualizations.
8. **Perform basic data loading and exploration tasks**
   Read datasets using Pandas, explore data with summary statistics and structure inspection (.head(), .info(), .describe()), and create simple visualizations using Matplotlib.

Why python ?



Source Image: https://www.openlogic.com/blog/top-open-source-programming-languages-runtimes

Python is not just a programming language — it's a comprehensive **platform for data analytics** that combines **ease of use**, a **powerful ecosystem**, and **broad community and industry adoption**. Whether you're cleaning raw data, visualizing trends, or building predictive models, **Python offers everything you need** to move from data to insights efficiently.

Miniconda and the full Anaconda distribution

Feature	Miniconda	Anaconda
Installer size	~50 MB	~3–4 GB
Included packages	Minimal (just conda, Python)	250+ packages pre-installed
Flexibility	High	Medium
Ideal for	Advanced users	Beginners or all-in-one setups
Customization	Full control	Somewhat limited initially

Recommendation:

- Use **Miniconda** if:
  - You’re comfortable managing environments and want to save space
  - You want to avoid package bloat and dependency conflicts
- Use **Anaconda** if:
  - You’re new to Python/data science
  - You want a ready-to-use toolkit without manual installation
- Let me know if you want a step-by-step comparison or installation guide.

Installing Python and Jupyter Notebook – Jupyter Notebook

To begin coding in Python for data analytics, you'll need a proper development environment. The recommended setup includes **Python**, essential libraries, and **Jupyter Notebook** for interactive development.

Option 2: Install Python and Jupyter via pip (For Advanced Users)
If you prefer to install Python and Jupyter manually or want a lightweight setup:

Step 1: Install Python

1. Go to: https://www.python.org/downloads/
2. Download and install Python 3.x for your OS.
3. Ensure you **check the box** that says **"Add Python to PATH"** during installation.

Step 2: Install pip (if not installed)

- Pip usually comes with Python. You can verify using:

bash

```
pip --version
```

Step 3: Install Jupyter Notebook and Libraries
Open your terminal or command prompt and run:

bash

```
pip install notebook pandas numpy matplotlib
```

Step 4: Launch Jupyter Notebook

bash

```
jupyter notebook
```

It will open Jupyter in your default browser.

Why Use Jupyter Notebooks?

- Interactive, cell-based interface for writing and running Python code.
- Ideal for data exploration, visualization, and documentation in one place.
- Supports Markdown and LaTeX for creating well-documented analysis reports.

YouTube Tutorial: How to Install JUPYTER NOTEBOOK in Windows 11

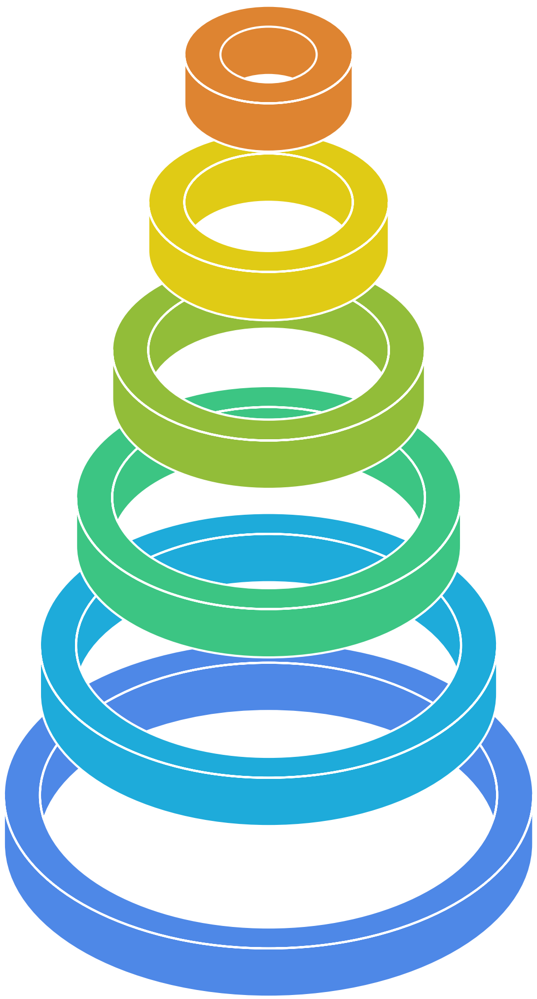
Summary

Tool	Purpose	Installation Method
Python	Programming Language	Python.org or Anaconda
pip	Python Package Installer	Included with Python
Jupyter	Interactive Coding Environment	Comes with Anaconda or via pip
Libraries	Data analysis and visualization support	Installed via pip or included in Anaconda

NLTK
Spark
theano
Numpy
Keras
Pytorch
TensorFlow
mxnet
learn
Scikit-learn
Pandas

Fundamental building blocks of Python programming

Welcome to the world of Python!



Functions

Reusable blocks of code

Input/Output

Mechanisms for user interaction

Loops

Structures that repeat code blocks

Conditionals

Statements that execute based on conditions

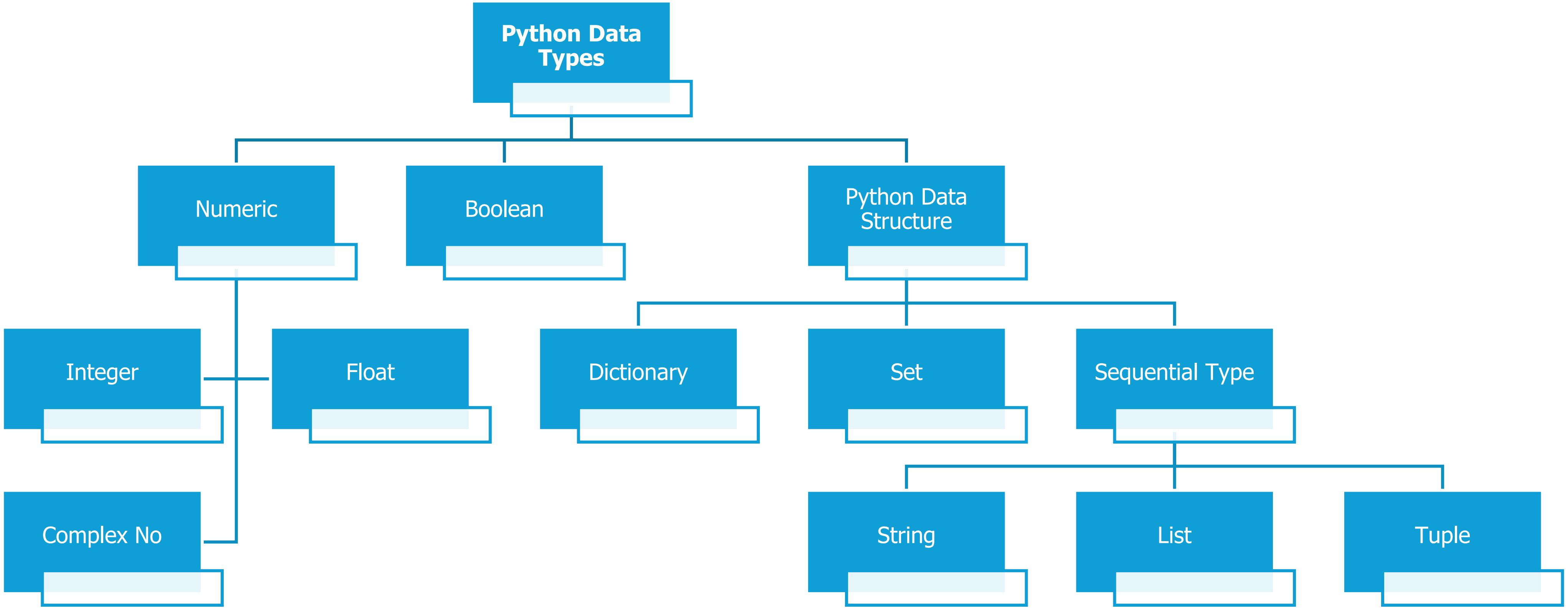
Data Types

Classifications of data values

Variables

Storage units for data values

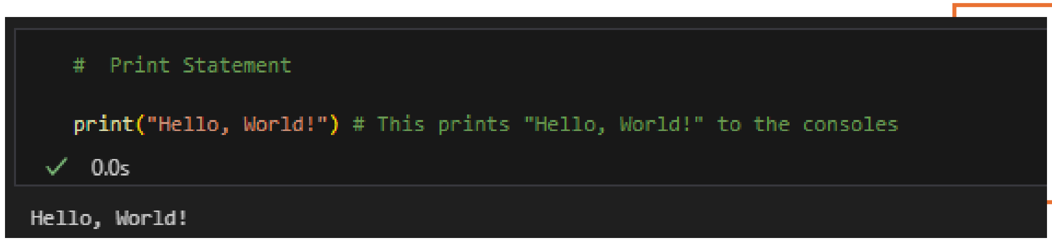
Python Data Type



Comment & Variables

What Are Comments?

- Lines ignored by the Python interpreter.
- Used to explain code, aid debugging, or temporarily disable execution.

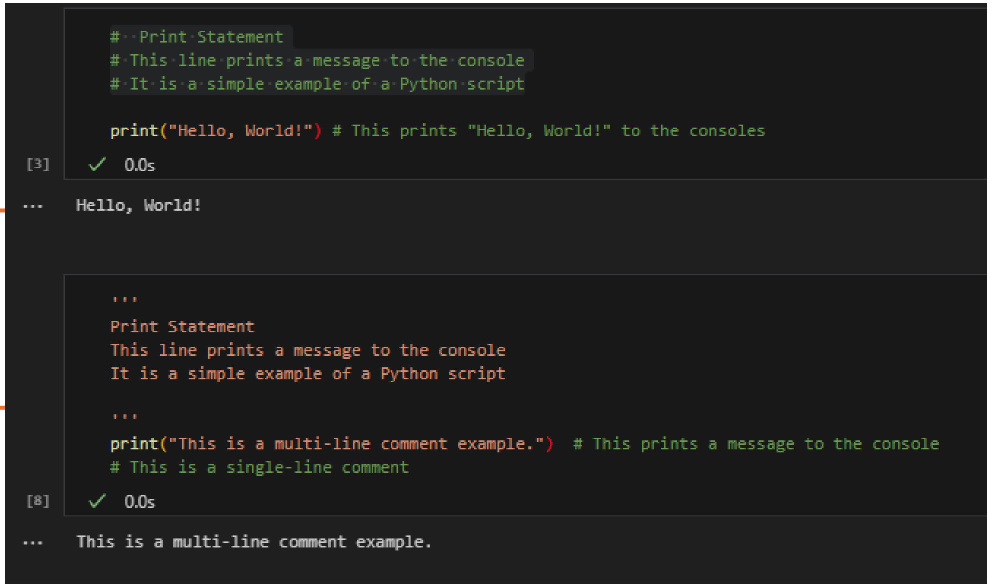


```
#  Print Statement

print("Hello, World!") # This prints "Hello, World!" to the consoles
✓ 0.0s

Hello, World!
```

Types of Comments

1. Single-line Comments

- Start with #



```
    # Print Statement
    # This line prints a message to the console
    # It is a simple example of a Python script

    print("Hello, World!") # This prints "Hello, World!" to the consoles
[3] ✓ 0.0s

... Hello, World!

    '''
    Print Statement
    This line prints a message to the console
    It is a simple example of a Python script

    '''
    print("This is a multi-line comment example.")  # This prints a message to the console
    # This is a single-line comment
[8] ✓ 0.0s

... This is a multi-line comment example.
```

Multi-line Comments

- Option 1: **Multiple # lines**
- Option 2: **Triple quotes (''' or """)**

Python has several built-in data types that are essential for programming:

Integers:

- □ Whole numbers,
- □ e.g., 5, -3.

Floats:

- □ Decimal numbers,
- □ e.g., *3.14*, *-0.001*.

Strings:

- □ Text data, enclosed in quotes,
- □ e.g., *"Hello, World!"*.

Booleans:

- □ Represents
- □ *True* or *False*.

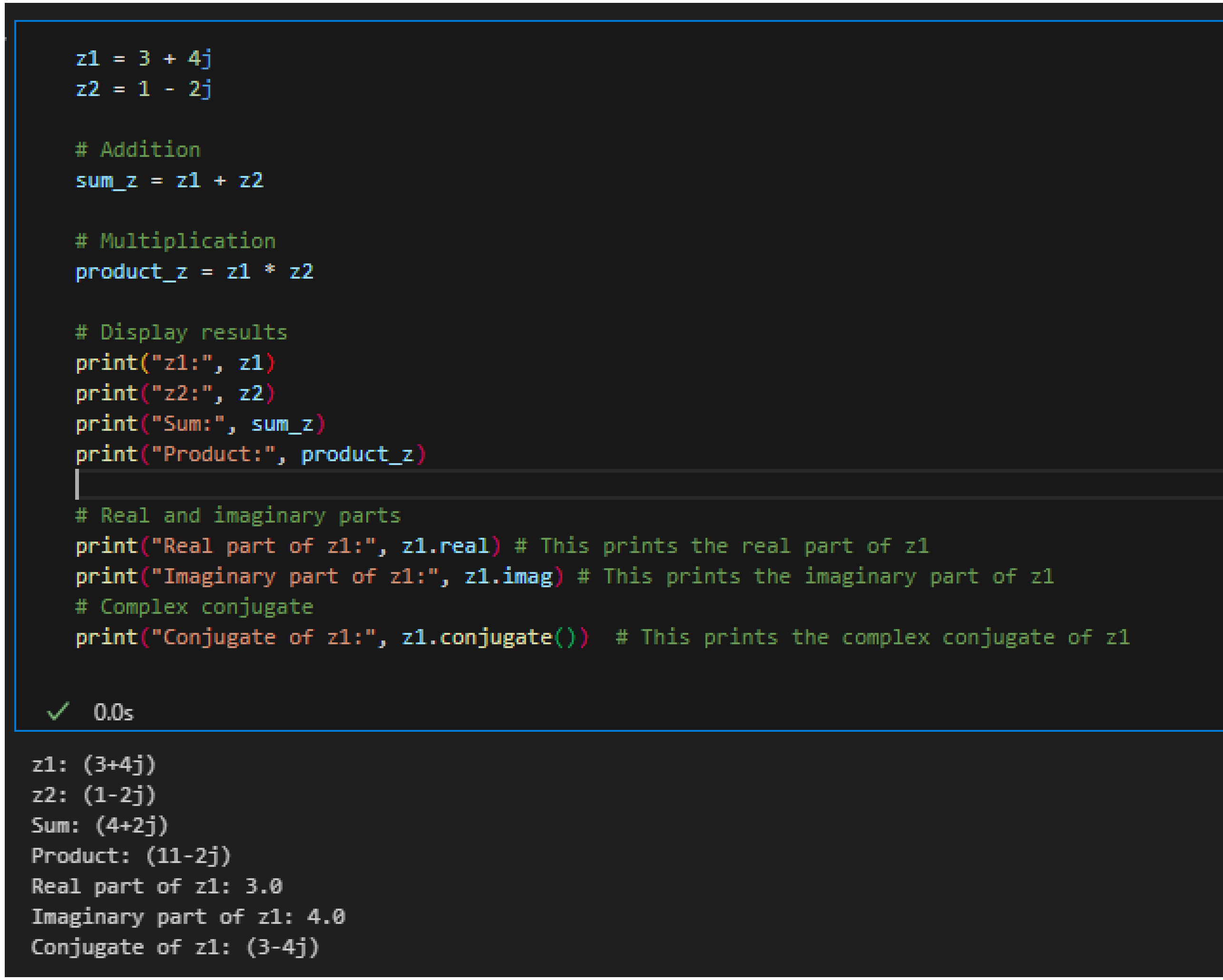
Example:

```
age = 25  # Integer
height = 5.9  # Float
name = "Alice"  # String
is_student = True  # Boolean
```

Complex Number

Complex Numbers
Complex numbers consist of a real part and an imaginary part, represented as *a + bj*, where a is the real part and b is the imaginary part.

z = 3 + 4j # where 3 is real, 4 is imaginary



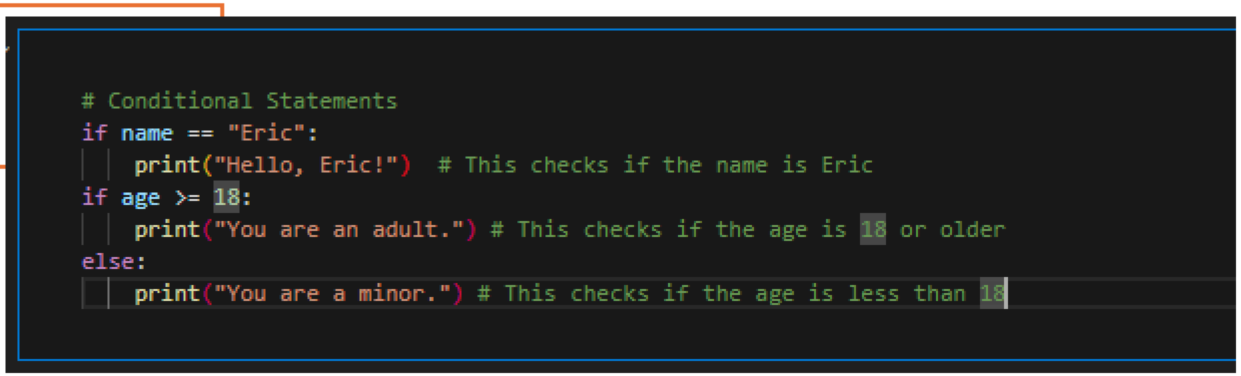
```
z1 = 3 + 4j
z2 = 1 - 2j

# Addition
sum_z = z1 + z2

# Multiplication
product_z = z1 * z2

# Display results
print("z1:", z1)
print("z2:", z2)
print("Sum:", sum_z)
print("Product:", product_z)

# Real and imaginary parts
print("Real part of z1:", z1.real) # This prints the real part of z1
print("Imaginary part of z1:", z1.imag) # This prints the imaginary part of z1
# Complex conjugate
print("Conjugate of z1:", z1.conjugate())  # This prints the complex conjugate of z1
```

```
z1: (3+4j)
z2: (1-2j)
Sum: (4+2j)
Product: (11-2j)
Real part of z1: 3.0
Imaginary part of z1: 4.0
Conjugate of z1: (3-4j)
```

Control Structures: Condition Statement & For Loop

1. Conditional Statements

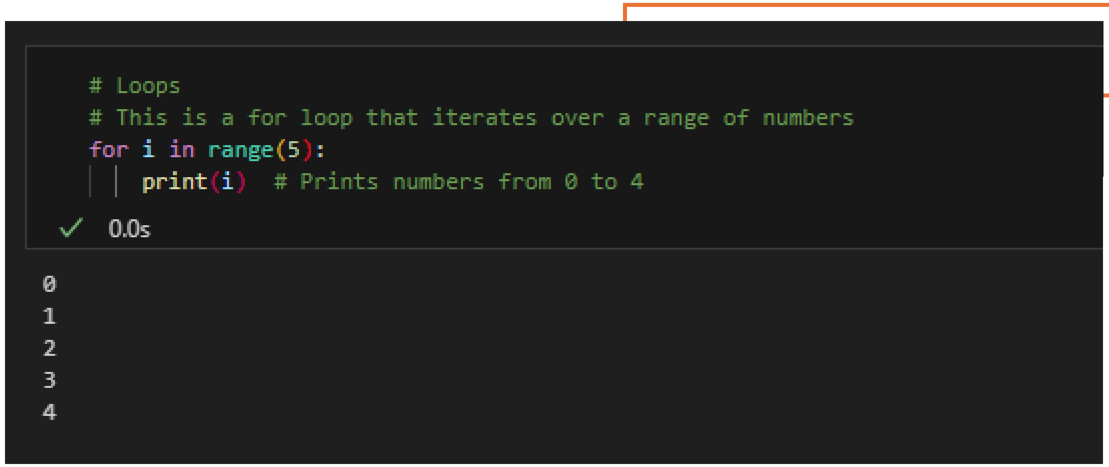
Conditional statements let you execute code based on certain conditions.



```
# Conditional Statements
if name == "Eric":
    print("Hello, Eric!")  # This checks if the name is Eric
if age >= 18:
    print("You are an adult.") # This checks if the age is 18 or older
else:
    print("You are a minor.") # This checks if the age is less than 18
```

2. Loops

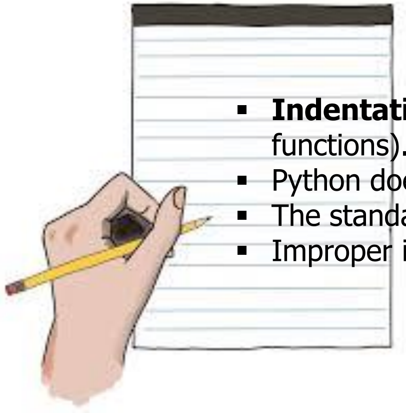
Loops enable you to repeat a block of code multiple times.



```
# Loops
# This is a for loop that iterates over a range of numbers
for i in range(5):
    print(i)  # Prints numbers from 0 to 4
```

✓ 0.0s

```
0
1
2
3
4
```

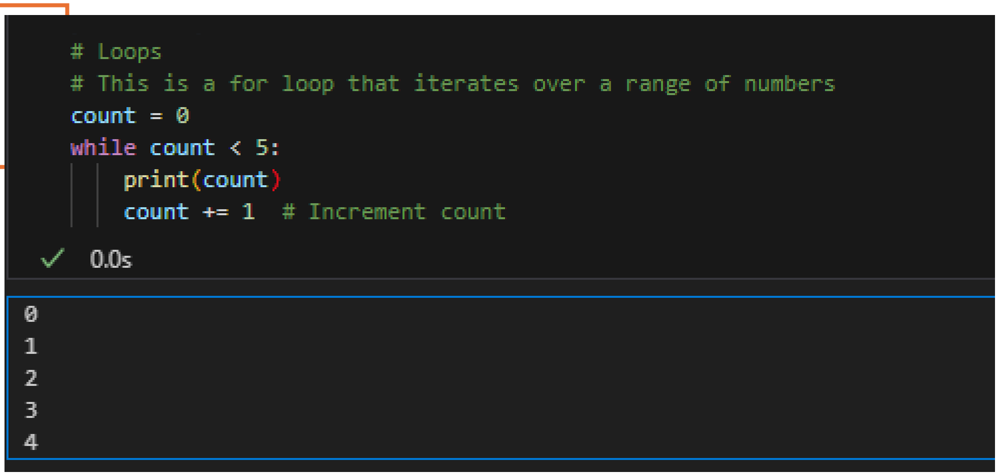


- **Indentation defines code blocks** in Python (e.g., inside if, for, while, and functions).
- Python does **not** use braces {} like other languages.
- The standard is **4 spaces** per indent level (avoid using tabs and spaces together).
- Improper indentation leads to errors.

Control Structures: While Loop

While Loop

Conditional statements let you execute code based on certain conditions.



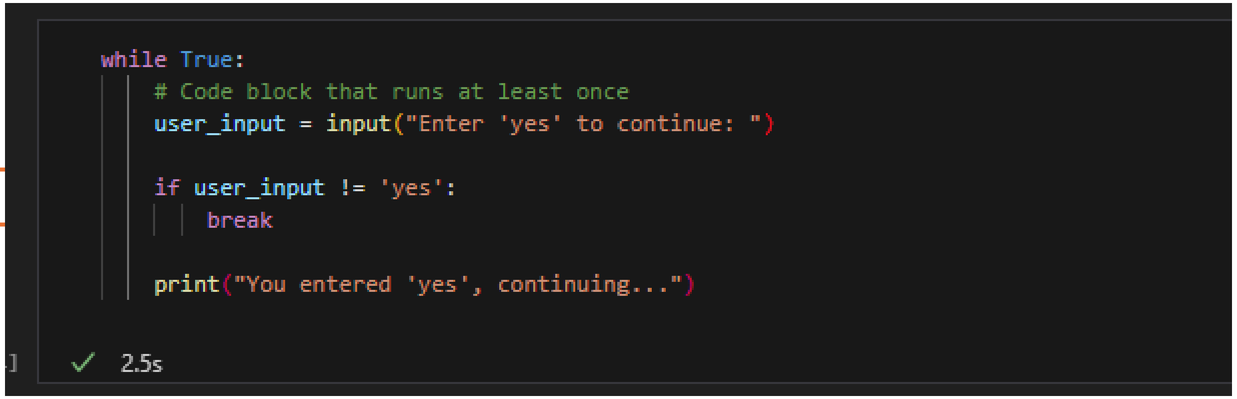
```
# Loops
# This is a for loop that iterates over a range of numbers
count = 0
while count < 5:
    print(count)
    count += 1  # Increment count
```

✓ 0.0s

```
0
1
2
3
4
```

No, Python does not have a built-in do...while loop like some other languages (e.g., C, Java).
However, you can **mimic** a do...while loop using a while True loop with a break condition. This ensures the loop runs **at least once**, similar to how do...while works.

Python Equivalent of do...while

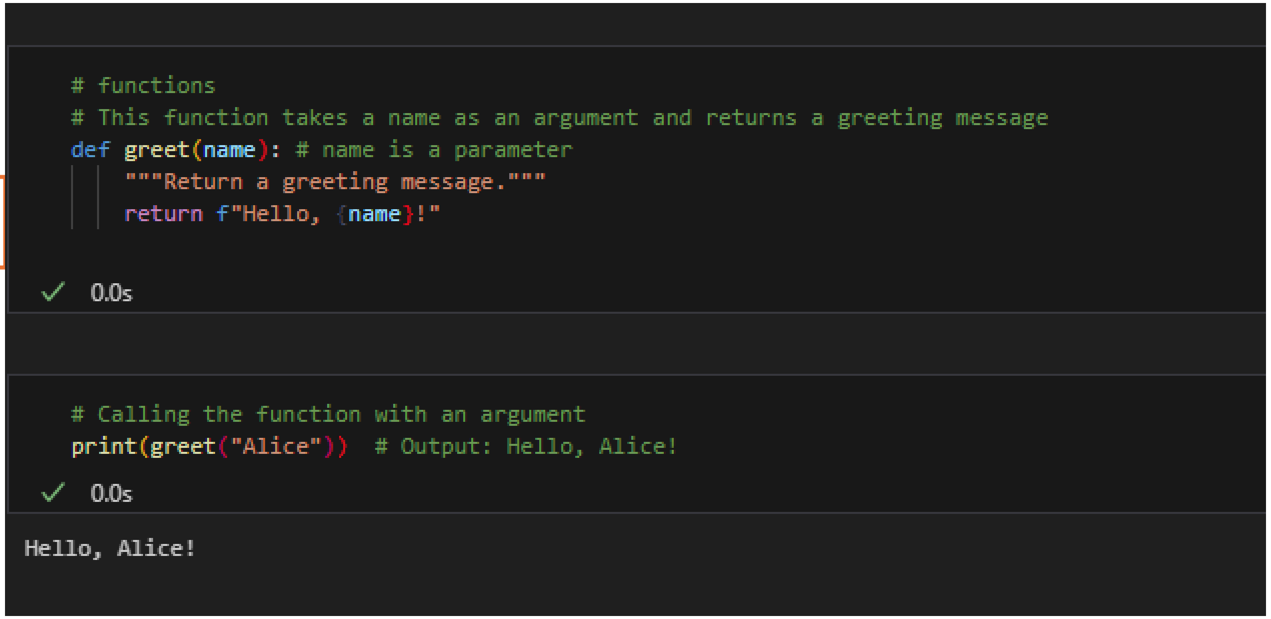


```
while True:
    # Code block that runs at least once
    user_input = input("Enter 'yes' to continue: ")

    if user_input != 'yes':
        break

    print("You entered 'yes', continuing...")
```

✓ 2.5s

Function & Input and Output

Functions are reusable blocks of code that perform a specific task. You can define a function using the def keyword.



```
# functions
# This function takes a name as an argument and returns a greeting message
def greet(name): # name is a parameter
    """Return a greeting message."""
    return f"Hello, {name}!"
```

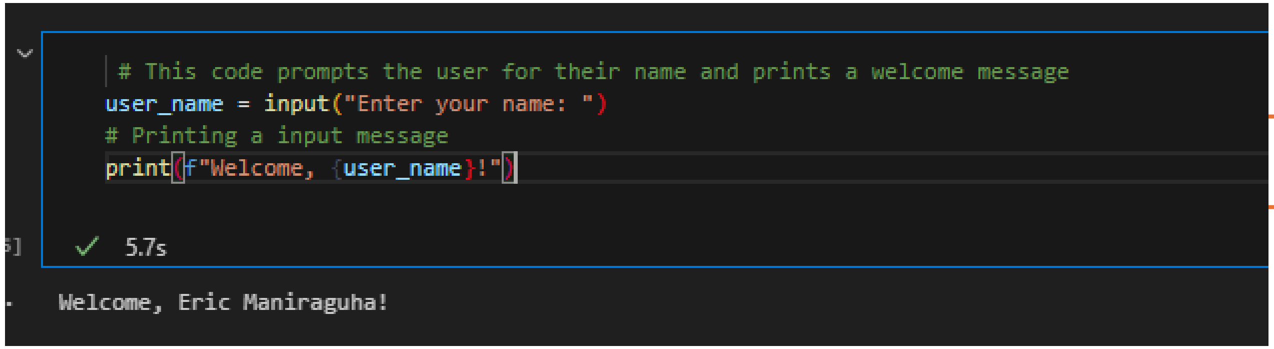
✓ 0.0s

```
# Calling the function with an argument
print(greet("Alice"))  # Output: Hello, Alice!
```

✓ 0.0s

```
Hello, Alice!
```

Python provides built-in functions for input and output operations.



```
# This code prompts the user for their name and prints a welcome message
user_name = input("Enter your name: ")
# Printing a input message
print(f"Welcome, {user_name}!")
```

✓ 5.7s

```
Welcome, Eric Maniraguha!
```

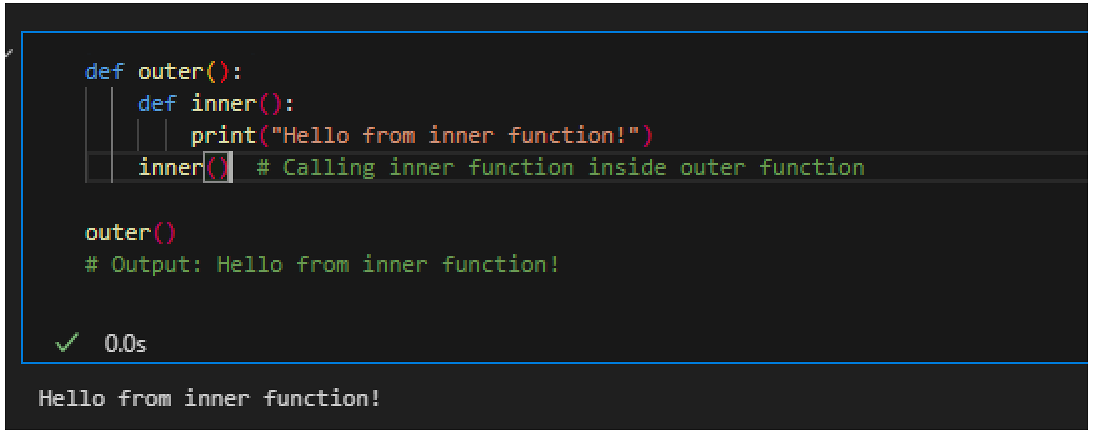
You can take user input using the input() function.

Inner Function

An **inner function** (also called a **nested function**) is a function defined *inside* another function. It exists only within the scope of the outer function and is usually used to help organize code or encapsulate some logic that should not be accessible outside the outer function.

Key points about inner functions:

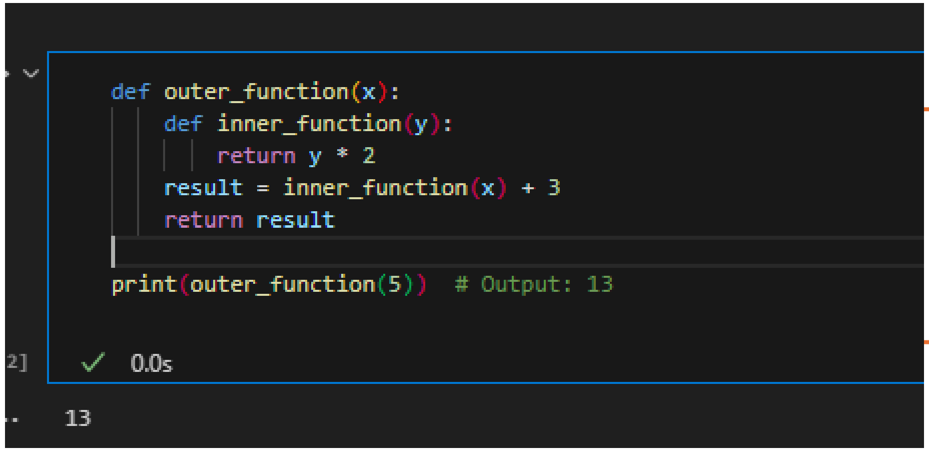
- Defined inside another function.
- Can access variables from the outer function.
- Helps keep code modular and clean.
- Cannot be called from outside the outer function.



```
def outer():
    def inner():
        print("Hello from inner function!")
    inner()  # Calling inner function inside outer function

outer()
# Output: Hello from inner function!
```

✓ 0.0s

```
Hello from inner function!
```



```
def outer_function(x):
    def inner_function(y):
        return y * 2
    result = inner_function(x) + 3
    return result

print(outer_function(5))  # Output: 13
```

✓ 0.0s

```
13
```

Explanation:

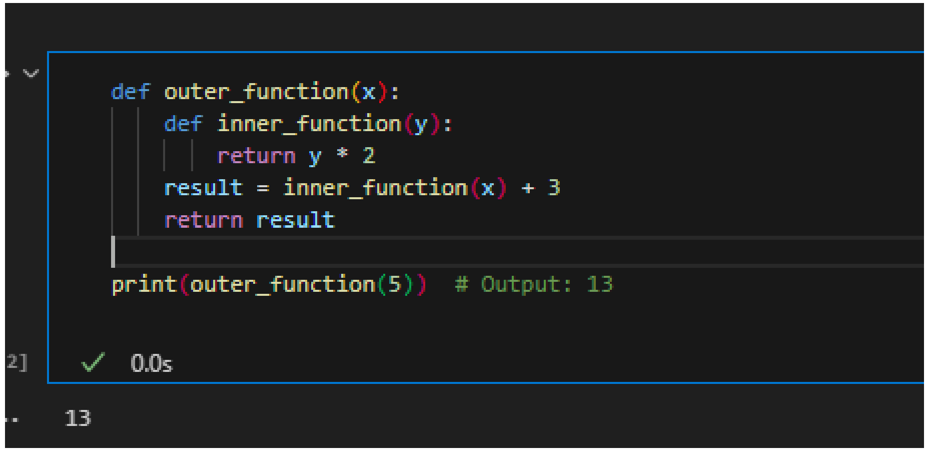
- inner_function is defined inside outer_function.
- outer_function calls inner_function with the argument x.
- inner_function doubles the input y.
- outer_function adds 3 to the result from inner_function and returns it.

Class Structure

In Python, a **class** is a blueprint for creating **objects**. Objects represent **real-world entities** with **attributes (data)** and **methods (functions)** that operate on the data.

```
class ClassName:
    def __init__(self, parameters):
        # initialize attributes
        self.attribute = value

    def method_name(self):
        # method logic
        pass
```

Basic Structure of a Class



```
def outer_function(x):
    def inner_function(y):
        return y * 2
    result = inner_function(x) + 3
    return result

print(outer_function(5))  # Output: 13
```

```
✓ 0.0s

13
```

Class & Object

```
class Person:
    def __init__(self, name, age):  # Constructor method
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

A **class** is like a blueprint for creating objects. It defines properties (attributes) and behaviors (methods).

An **object** is an instance of a class. You use the class to create (instantiate) an object.

```
# Creating objects
person1 = Person(“Didier", 30)
person2 = Person(“Aline", 25)

# Calling methods
person1.greet()   # Output: Hello, my name is Didier and I am 30 years old.
person2.greet()   # Output: Hello, my name is Aline and I am 25 years old.
```

Key Concepts

Concept	Explanation
__init__	A special method called a constructor; used to initialize object attributes
self	Refers to the current instance of the class
Attributes	Variables that belong to the object (like name and age)
Methods	Functions defined inside a class

Class with or without __init__

If you **don't add the __init__ method** in a Python class, **Python will still create the object**, but it won't automatically initialize any attributes when the object is created.

Case 1: Class *without* __init__

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

# Create object
a = Animal()
a.speak()  # Output: Animal speaks
```

This works fine because you're not trying to pass any data when creating the object.

Case 2: Trying to pass arguments *without* __init__

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

# This will cause an error
a = Animal("Dog")
```

⊗ *TypeError: Animal() takes no arguments*

You get this error because without __init__, the class doesn't expect any arguments.

So when should you use __init__?

- Use it when your object needs **initial values** (attributes) upon creation.
- Without it, you'd need to set attributes manually after object creation.

Example with manual attribute setting:

```
class Animal:
    pass

a = Animal()
a.name = "Dog"
print(a.name)  # Output: Dog
```

This works, but it's not clean or safe, especially in larger applications.

What does pass mean in Python?

In Python, pass is a **placeholder** statement. It does **nothing** when executed — it's just used to **define an empty block of code** that won't cause an error

Summary:

With __init__	Without __init__
Initializes object attributes	No automatic initialization
Accepts arguments during instantiation	Must set attributes manually
More structured and safer	More error-prone and harder to manage

What is a Constructor in Python?

A **constructor** is a special method that is **automatically called** when you create a new object from a class. Its main job is to **initialize the object's attributes** (i.e., set initial values for the object).

In Python, the constructor is named __init__

```
class Person:
    def __init__(self, name, age):  # constructor
        self.name = name
        self.age = age
```

- __init__ is short for "**initialize**"
- It gets called automatically when you create an object, like this:

```
p = Person("Eric", 30)  # __init__ is automatically called here
```

How it works step by step:

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

# Creating an object
p = Person("Alice")

# Accessing the attribute
print(p.name)  # Output: Alice
```

When Person("Alice") runs:

- Python creates a new Person object
- It automatically calls __init__(self, "Alice")
- self.name = "Alice" assigns the name to the object

Default constructor (if you don't define one)

If you don't create your own __init__, Python adds a default one that takes no arguments (except self). But then you must set attributes manually:

```
class Empty:
    pass

e = Empty()
e.name = "Eric"  # manual
```

Future Work / Advanced Topics in OOP

As students become more comfortable with OOP basics, the following topics are recommended for deeper understanding and practical application:

- **Encapsulation**: Protecting object data using public, private, and protected attributes.
- **Abstraction**: Hiding implementation details using abstract base classes and interfaces.
- **Inheritance**: Reusing and extending existing classes.
- **Polymorphism**: Writing flexible code that works with objects of different classes.
- **Real-world Modeling**: Using classes to simulate real systems like libraries, hospitals, or banks.
- **Exception Handling in OOP**: Making classes more robust by handling errors gracefully.
- **Design Patterns (Intro)**: Introduction to patterns like Singleton, Factory, and Observer.

Python Programming Exercises for Students

Section A: Input, Output, and Variables

Question 1:
Write a Python program that asks the user to input their name and age, then prints a message saying: "Hello [name], you are [age] years old."

Question 2:
Ask the user to input two numbers and print their sum, difference, product, and quotient.

Section B: Conditionals and Loops

Question 3:
Write a program that asks the user to enter a number. If the number is even, print "Even number", otherwise print "Odd number".

Question 4:
Write a Python program that prints all numbers from 1 to 20 using a for loop.

Question 5:
Write a program that asks the user for a number and prints its multiplication table up to 10.

Section C: Strings and Lists

Question 6:
Ask the user to enter a word and print how many vowels (a, e, i, o, u) it contains.

Question 7:
Given the list numbers = [4, 7, 2, 9, 5], write code to:

- Print the list in reverse order.
- Find and print the largest number.

Question 8:
Write a program that takes a sentence from the user and counts how many words it contains.

Section D: Functions

Question 9:
Write a function called square(number) that returns the square of a number. Call the function with the number 5 and print the result.

Question 10:
Write a function that takes a list of numbers as an argument and returns the sum of all elements in the list.

Thank you!

Stay Connected!