Subject of the notebook: C++ vs Python (Full Notes)

Authourized by : *NOOR UL NISA*

Contact by: Github / Linkdein

Table of Contents

🔷 *1. What is C++?*

Definition: C++ is a general-purpose, compiled, statically typed programming language created by Bjarne Stroustrup. It is an extension of the C language and supports procedural, object-oriented, and generic programming.

✅ Key Features of C++

- **Compiled Language**: Requires a compiler (like `g++`) to convert code into machine code.
- **Statically Typed**: Data types must be declared explicitly.
- **High Performance**: Great for system-level programming.
- **Supports OOP**: Classes, objects, inheritance, polymorphism, etc.

✅ Sample Program (C++)

```
#include<iostream>
using namespace std;

int main() {
    cout << "Hello, C++!";
    return 0;
}
```

🔷 2. What is Python?

Definition: Python is an interpreted, dynamically typed, high-level programming language created by Guido van Rossum. It emphasizes code readability and simplicity.

✅ Key Features of Python

- **Interpreted Language**: Runs line-by-line using Python interpreter.
- **Dynamically Typed**: No need to declare variable types.
- **Beginner-Friendly Syntax**: Simple and clean.
- **Extensive Libraries**: Great for web, AI, data science, scripting.

✅ Sample Program

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

🔷 3. Basic Syntax Differences

Feature	C++	Python
File Extension	`.cpp`	`.py`
Hello World	`cout << "Hello";`	`print("Hello")`
Main Function	`int main() { ... }`	Optional: `if __name__ == '__main__':`
Variable Decl.	`int x = 5;`	`x = 5`
Semicolon	Required (`;`)	Not required
Block Structure	Braces `{}` + indentation	Indentation only
Comments	`//`, `/* */`	`#`, `''' '''`
String Output	`cout << "Hello";`	`print("Hello")`
String Input	`cin >> x;`	`x = input()`
Type Conversion	`stoi(s)`, `to_string(n)`	`int(s)`, `str(n)`
Libraries	`#include <iostream>`	Built-in or `import`
Code Execution	Compile & Run (`g++ file.cpp`)	Direct Run (`python file.py`)

🔷 4. Variables and Data Types

📌 What is a Variable?

A **variable** is a container used to store data values. It can change during execution.

📌 What is a Data Type?

A **data type** defines what kind of data a variable can hold.

✅ Variable Declaration

Feature	C++	Python
Declaration	Must specify type	No need to specify type
Initialization	Done at declaration or later	Same

C++ Example:

```cpp
int age = 20;
float pi = 3.14;
char grade = 'A';
bool is_passed = true;
string name = "Alice";
```

Python Example:

```python
age = 20
pi = 3.14
grade = 'A'
is_passed = True
name = "Alice"
```

✅ Common Data Types

Type	C++	Python	Example (C++)	Example (Python)
Integer	`int`, `long`, `short`	`int`	`int x = 10;`	`x = 10`
Float	`float`, `double`	`float`	`float y = 3.14;`	`y = 3.14`
Character	`char`	`str` (1 char)	`char c = 'A';`	`c = 'A'`
String	`string`	`str`	`string s = "Hi";`	`s = "Hi"`
Boolean	`bool`	`bool`	`bool b = true;`	`b = True`
Null/None	`nullptr`, `NULL`	`None`	`int* p = nullptr;`	`x = None`

🔷 *5. Strings*

📌 What is a String?

A **string** is a sequence of characters used to store and manipulate text.

✅ Declaration and Initialization

Feature	C++	Python
String Decl	`string s = "Hello";`	`s = "Hello"`

C++ Example:

```cpp
#include<iostream>
#include<string>
using namespace std;

int main() {
    string name = "Alice";
    cout << name << endl;
    return 0;
}
```

Python Example:

```python
name = "Alice"
print(name)
```

🔷 6. String Indexing

📌 What is String Indexing?

String indexing allows *access to individual characters in a string using positions* (indexes).

- Index starts from 0 in both languages.
- Python supports negative indexing (e.g., `-1` is last character), C++ does not.

✅ Indexing Table

Operation	C++	Python
First character	`s[0]`	`s[0]`
Last character	`s[s.length()-1]`	`s[-1]`
Substring	`s.substr(1, 3)`	`s[1:4]`
Length	`s.length()` or `s.size()`	`len(s)`

C++ Example:

```cpp
string s = "Python";
cout << s[0] << endl;           // P
cout << s.substr(1, 3) << endl; // yth
cout << s[s.length() - 1];      // n
```

Python Example:

```
s = "Python"
print(s[0])       # P
print(s[1:4])     # yth
print(s[-1])      # n
```

🔷 7. Loops

📌 What is a Loop?

A **loop** allows you to execute a block of code repeatedly as long as a certain condition is true. It is useful for tasks like iteration, repetition, and automation.

✅ Types of Loops

1. `for` loop
2. `while` loop
3. `do-while` loop *(only in C++)*

🔷 *For Loop*

Definition: Used when the number of iterations is known.

Feature	C++	Python
Syntax/Initialization	`for(int i=0; i<5; i++)`	`for i in range(5):`
Range Function	Use counter manually	Use `range()` function

> ***Use: Iterating over known range.***

C++ Example:

```
for(int i = 0; i < 5; i++) {
    cout << i << " ";
}
```

Python Example:

```
for i in range(5):
    print(i, end=" ")
```

◈ While Loop

Definition: Repeats as long as a condition is true.

Use: When the end condition isn't predefined.

Feature	C++	Python
Syntax	`while(condition)`	`while condition:`

C++ Example:

```cpp
int i = 0;
while(i < 5) {
    cout << i << " ";
    i++;
}
```

Python Example:

```python
i = 0
while i < 5:
    print(i, end=" ")
    i += 1
```

◈ Do-While Loop (Only in C++)

Definition: Runs at least once before checking the condition.

Use: Menu-driven programs, input-first loops.

- Executes the loop **at least once** before checking the condition.
- Not available in Python.

C++ Example:

```cpp
int i = 0;
do {
    cout << i << " ";
    i++;
} while(i < 5);
```

☑ Loop Control Statements

Statement	Purpose	C++ Syntax	Python Syntax

Statement	Purpose	C++ Syntax	Python Syntax
`break`	Exit the loop immediately	`break;`	`break`
`continue`	Skip the current iteration	`continue;`	`continue`

Example (Python):

```python
for i in range(5):
    if i == 3:
        continue
    print(i)
```

Example (C++):

```cpp
for(int i = 0; i < 5; i++) {
    if(i == 3)
        continue;
    cout << i << endl;
}
```

🔁 *Loop Comparison*

Type	Condition Location	Runs at Least Once	C++	Python
For Loop	Beginning	❌	✅	✅
While Loop	Beginning	❌	✅	✅
Do-While	End	✅	✅	❌

🔷 *8. Conditional Statements*

📌 *What are Conditional Statements?*

Conditional statements are used to *perform different actions based on different conditions*. They allow the program to make decisions and change the flow of execution.

✅ Common Types of Conditional Statements

1. `if` statement
2. `if-else` statement
3. `else-if` ladder (C++) / `elif` ladder (Python)
4. `switch` statement (C++ only)
5. `match-case` (Python 3.10+ only)

🔷 1. If Statement

Definition: Executes a block of code if a specified condition is true.

Language	Syntax Example
C++	`if (condition) { ... }`
Python	`if condition:`

C++ Example:

```cpp
int age = 18;
if (age >= 18) {
    cout << "Adult";
}
```

Python Example:

```python
age = 18
if age >= 18:
    print("Adult")
```

🔷 2. If-Else Statement

Definition: Executes one block if condition is true, else another block.

C++ Example:

```cpp
int num = 7;
if (num % 2 == 0) {
    cout << "Even";
} else {
    cout << "Odd";
}
```

Python Example:

```python
num = 7
if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

🔷 3. Else-If Ladder / Elif Ladder

Definition: Used when multiple conditions need to be checked in sequence.

Language	Keyword
C++	`else if`
Python	`elif`

C++ Example:

```cpp
int marks = 75;
if (marks >= 90) {
    cout << "A+";
} else if (marks >= 80) {
    cout << "A";
} else if (marks >= 70) {
    cout << "B";
} else {
    cout << "Fail";
}
```

Python Example:

```python
marks = 75
if marks >= 90:
    print("A+")
elif marks >= 80:
    print("A")
elif marks >= 70:
    print("B")
else:
    print("Fail")
```

🔷 4. Switch Statement (Only in C++)

Definition: A multi-branch conditional that tests a variable for equality against multiple values.

C++ Example:

```cpp
int day = 2;
switch(day) {
    case 1:
        cout << "Monday";
        break;
    case 2:
```

```
        cout << "Tuesday";
        break;
    default:
        cout << "Other day";
}
```

🔷 5. Match-Case (Python 3.10+)

Python's alternative to `switch` in newer versions.

Python Example:

```
day = 2
match day:
    case 1:
        print("Monday")
    case 2:
        print("Tuesday")
    case _:
        print("Other day")
```

🔄 Summary Table

Condition Type	C++	Python
Simple If	✅ `if`	✅ `if`
If-Else	✅ `if-else`	✅ `if-else`
Else-If/Elif	✅ `else if`	✅ `elif`
Switch/Match	✅ `switch`	✅ `match-case` (3.10+)

🔷 9. Functions

📌 *What is a Function?*

A **function** is a *reusable block* of code that performs a specific task. Functions help organize code, reduce repetition, and improve readability.

✅ Types of Functions

1. Built-in Functions
2. User-defined Functions
3. Parameterized Functions
4. Functions with Return Values

◈ Function Syntax Comparison

Feature	C++	Python
Define	`returnType functionName()`	`def function_name():`
Call	`functionName();`	`function_name()`
Parameters	`void greet(string name)`	`def greet(name):`
Return	`return value;`	`return value`

◈ 1. Built-in Functions

C++ Examples: `cout`, `cin`, `sqrt()`, `strlen()` **Python Examples:** `print()`, `len()`, `type()`, `input()`

◈ 2. User-defined Function

C++ Example:

```cpp
#include<iostream>
using namespace std;

void greet() {
    cout << "Hello from C++";
}

int main() {
    greet();
    return 0;
}
```

Python Example:

```python
def greet():
    print("Hello from Python")

greet()
```

◈ 3. Function with Parameters

C++ Example:

```cpp
void greet(string name) {
    cout << "Hello " << name;
}
```

Python Example:

```python
def greet(name):
    print("Hello", name)
```

🔹 *4. Function with Return Value*

C++ Example:

```cpp
int add(int a, int b) {
    return a + b;
}
```

Python Example:

```python
def add(a, b):
    return a + b
```

🔄 Difference Table

Feature	C++	Python
Main Function	`int main()` required	Optional `main()`
Return Type Required	Yes (e.g., `int`, `void`)	No need to specify
Function Overloading	Supported	Not supported
Default Parameters	Supported	Supported

🔹 *10. Recursion*

📌 *What is Recursion?*

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.

✅ When to Use Recursion

- When a problem can be broken down into smaller subproblems.
- Suitable for tasks like factorial, Fibonacci, tree traversal, etc.

Basic Structure

Language	Syntax
C++	Function calls itself
Python	Function calls itself

What is Factorial?

The **factorial** of a non-negative integer n is the product of all positive integers less than or equal to n.

Mathematical Formula:

```
n! = n × (n-1) × (n-2) × ... × 1
```

Example:

```
5! = 5 × 4 × 3 × 2 × 1 = 120
```

Why Use It?

- Common in mathematics, permutations, combinations.
- Used to demonstrate recursion and iterative techniques.

Example: Factorial using Recursion

C++ Example:

```cpp
int factorial(int n) {
    if(n == 0 || n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

Python Example:

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

What is Fibonacci Sequence?

The **Fibonacci sequence** is a series of numbers where each number is the sum of the two preceding ones.

Mathematical Formula:

```
F(n) = F(n-1) + F(n-2)
```

With base cases:

```
F(0) = 0,
F(1) = 1
```

Example:

```
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
```

Why Use It?

- Appears in nature (e.g., spirals, growth patterns)
- Used in algorithm design, dynamic programming, and recursion examples

◈ *Example: Fibonacci using Recursion*

C++ Example:

```cpp
int fibonacci(int n) {
    if(n <= 1)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Python Example:

```python
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

⚠ *Things to Remember*

- Always have a **base case** to stop recursion.
- Recursive calls should move toward the base case.
- Too many recursive calls may cause **stack overflow**.

🔄 *Iteration vs Recursion*

Feature	Iteration	Recursion
Approach	Loop (for/while)	Function calls itself
Memory usage	Less	More (call stack)
Readability	Simple for small problems	Elegant for recursive logic
Speed	Generally faster	Slower due to call overhead

🔷 *11. Type Conversion*

📌 *What is Type Conversion?*

Type conversion is the *process of converting one data type into another*. This is essential when performing operations involving multiple data types.

✅ *Why Use Type Conversion?*

- To avoid type mismatch errors
- To perform meaningful operations across types
- To work with APIs/libraries that require specific types

🔷 Types of Type Conversion

Type	C++	Python
Implicit Conversion	Automatic by compiler	Automatic by interpreter
Explicit Conversion	Cast using syntax	Use functions like `int()`, etc

---

🔷 *1. Implicit Type Conversion*

Also known as **type promotion**. Happens automatically.

C++ Example:

```
int x = 5;
double y = x + 2.5;  // x implicitly converted to double
```

Python Example:

```
x = 5
y = x + 2.5  # x converted to float
print(y)  # Output: 7.5
```

🔷 *2. Explicit Type Conversion (Type Casting)*

Manually converting a value from one type to another.

C++ Syntax: `(new_type) value` **Python Syntax:** `new_type(value)`

C++ Example:

```
float pi = 3.14;
int approx = (int) pi;  // approx = 3
```

Python Example:

```
pi = 3.14
approx = int(pi)  # approx = 3
```

🔄 Common Type Casting Functions in Python

- `int()` → Convert to integer
- `float()` → Convert to float
- `str()` → Convert to string
- `bool()` → Convert to boolean

🔄 Common Type Casting in C++

- `int(value)` *(C-style)*
- `static_cast<new_type>(value)` *(Recommended)*

⚠️ *Note*

Improper type casting can lead to:

- Data loss (e.g., float to int)
- Unexpected behavior (e.g., string to number conversion errors)

🔷 *Data Structures (C++ vs Python)*

📌 What are Data Structures?

Data structures are specialized formats to organize, process, retrieve, and store data. They are essential for writing efficient algorithms and solving complex problems.

☑️ Why Use Data Structures?

- To store and access data efficiently.
- To implement algorithms with optimal performance.
- To logically organize data (e.g., lists of students, inventory systems, etc.).
- To reduce time and space complexity in programs.

📊 Comparison of Common Data Structures

Structure	C++	Python	Purpose / Usage
Array	`int arr[5];`	`arr = [1, 2, 3, 4, 5]`	Fixed-size sequential data
Vector	`vector<int> v;`	`list = [1, 2, 3]`	Dynamic array (auto-resizing)
List (STL)	`list<int> l;`	`list = [1, 2, 3]`	Doubly linked list (in C++), built-in list in Python
Set	`set<int> s;`	`s = {1, 2, 3}`	Unique elements only, unordered
Map / Dict	`map<string, int> m;`	`d = {'a': 1}`	Key-value pairs

🔷 1. Arrays

➤ Definition

An array is a fixed-size sequential collection of elements of the same type.

➤ Use

Used when the size of the data is known and doesn't change frequently.

➤ C++ Example

```cpp
#include <iostream>
using namespace std;

int main() {
    int arr[3] = {10, 20, 30};
    cout << arr[1];  // Output: 20
    return 0;
}
```

➤ Python Example

```
arr = [10, 20, 30]
print(arr[1])  # Output: 20
```

◈ 2. Vectors (C++) / Lists (Python)

➤ Definition

A dynamic array that grows or shrinks in size automatically.

➤ Use

Best for unknown or variable-size collections.

➤ C++ Example

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> v = {1, 2, 3};
    v.push_back(4);
    cout << v[3];  // Output: 4
    return 0;
}
```

➤ Python Example

```
v = [1, 2, 3]
v.append(4)
print(v[3])  # Output: 4
```

◈ 3. Sets

➤ Definition

A set is a collection of unordered and unique elements.

➤ Use

Used when duplication is not allowed, e.g., storing unique usernames.

➤ C++ Example

```cpp
#include <set>
#include <iostream>
using namespace std;

int main() {
    set<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(10);  // Duplicate, ignored
    for(int i : s) cout << i << " ";  // Output: 10 20
    return 0;
}
```

➤ Python Example

```python
s = {10, 20, 10}
s.add(30)
print(s)  # Output: {10, 20, 30}
```

◈ 4. Map (C++) / Dictionary (Python)

➤ Definition

Key-value data structure. Each unique key maps to a specific value.

➤ Use

Efficient lookups, such as word-frequency, user data, etc.

➤ C++ Example

```cpp
#include <map>
#include <iostream>
using namespace std;

int main() {
    map<string, int> age;
    age["Ali"] = 25;
    age["Sara"] = 22;
    cout << age["Ali"];  // Output: 25
    return 0;
}
```

➤ Python Example

```
age = {"Ali": 25, "Sara": 22}
print(age["Ali"])  # Output: 25
```

🧠 Summary Table

Feature	C++ Syntax	Python Syntax
Array	`int arr[5];`	`arr = [1, 2, 3]`
Vector/List	`vector<int> v;`	`list = [1, 2, 3]`
Set	`set<int> s;`	`s = {1, 2, 3}`
Map/Dict	`map<string, int> m;`	`d = {'a': 1}`

🔷 *Object-Oriented Programming (OOP)*

📌 What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **"objects"**, which contain **data** (attributes) and **methods** (functions).

☑️ Why Use OOP?

- Models real-world entities using classes and objects.
- Promotes **code reuse** with **inheritance**.
- Enhances **security** through **encapsulation**.
- Improves code organization and readability.
- Enables **polymorphism** (one interface, many implementations).
- Simplifies large and complex codebases via modularity.

🔷 ***Key OOP Concepts***

Concept	Description
Class	Blueprint for creating objects. Defines properties (variables) and behaviors (methods).
Object	Instance of a class. Holds data and can perform actions using methods.
Encapsulation	Bundling of data and functions that operate on the data.
Abstraction	Hiding internal implementation and showing only relevant details.

Concept	Description
Inheritance	Acquiring properties and behaviors from another class (parent → child).
Polymorphism	Same function or operator behaves differently based on context or input.

◈ *Class and Object in C++*

```cpp
#include <iostream>
using namespace std;

class Person {
public:
    string name;
    int age;

    void introduce() {
        cout << "My name is " << name << " and I am " << age << " years old.";
    }
};

int main() {
    Person p1;
    p1.name = "Ali";
    p1.age = 22;
    p1.introduce();  // Output: My name is Ali and I am 22 years old.
    return 0;
}
```

◈ *Class and Object in Python*

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

    def introduce(self):
        print(f"My name is {self.name} and I am {self.age} years old.")

p1 = Person("Ali", 22)
p1.introduce()  # Output: My name is Ali and I am 22 years old.
```

🔄 *C++ vs Python: OOP Comparison Table*

Feature	C++	Python
Class Definition	`class ClassName { };`	`class ClassName:`

Feature	C++	Python
Constructor Name	Same as class name	`__init__()` method
Destructor	`~ClassName()`	`__del__()` (rarely used)
Access Specifiers	`public`, `private`, `protected`	No strict enforcement; all public
Object Creation	`ClassName obj;`	`obj = ClassName()`
Function Overload	Supported	Not directly supported
Inheritance Syntax	`class B : public A {}`	`class B(A):`

Summary

- **OOP** helps structure code in a more understandable and maintainable way.
- **C++** offers more control (e.g., access modifiers), while **Python** simplifies the syntax.
- Both support key OOP principles:
  - Inheritance
  - Encapsulation
  - Abstraction
  - Polymorphism