Introudction to Python!

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*
> — Martin Fowler

📘 Topics to be Covered

1. **What is Python ?**
2. **Key Features of Python**
3. **Limitations of Python**
4. **Scope of Python**
5. **Python — A Programming or a Scripting Language?**
6. **Data types in Python**
7. **Expressions in Python**
8. **Programming Errors**

1. What is Python?

Python is a high-level, interpreted programming language known for its **simplicity**, **readability**, and **versatility**.

- Created by **Guido van Rossum**
- First released in **1991**
- Now used by companies like **Google**, **Netflix**, **NASA**, **Tesla**, and **Meta**

Python makes it **easy for beginners to start**, while still being **powerful enough** for experts to build complex systems.

Python is one of the most popular programming language. why ?

- Used in **cutting-edge fields**: AI, Machine Learning, Data Science, Automation, and more.
- Beginner-friendly syntax.
- Fast development and prototyping.
- A huge collection of libraries for **everything**.
- A massive, supportive community.

2. Key Features of Python

- **Interpreted** – No compilation required; run code line-by-line.
- **Dynamically Typed** – No need to declare variable types.
- **High-Level Language** – Write code like you're writing English.
- **Object-Oriented** – Build reusable and organized code.
- **Cross-Platform** – Runs on Windows, Mac, Linux seamlessly.

3. Limitations of Python

Even the best tools have limitations:

- **Slower than C++/Java** because it is interpreted.
- **Not ideal for mobile app development.**
- **GIL (Global Interpreter Lock)**: Limits true multithreading.
- **Runtime Errors** due to dynamic typing.
- **High memory usage** compared to lower-level languages.

4. Scope of Python

Python shines in:

Domain	Tools/Libraries	What You Can Build
Data Science	`pandas`, `numpy`	Analyze real-world data
Machine Learning	`scikit-learn`, `xgboost`	Predict house prices, recommend videos
AI / Deep Learning	`tensorflow`, `pytorch`	Build chatbots, autonomous systems
Web Development	`Flask`, `Django`	Build websites and REST APIs
Automation	`selenium`, `pyautogui`	Auto-fill forms, schedule emails
Games	`pygame`	Build 2D games and interactive apps

5. Python — A Programming or a Scripting Language?

- Python is a general-purpose programming language that is often used as a scripting language.
- Python as a scripting language - Used for quick automation or system-level tasks.
  ```python
  # Rename all .txt files in a folder
  import os
  folder = "C:/Users/Sanjeev/Desktop/textfiles"
  for filename in os.listdir(folder):
      if filename.endswith(".txt"):
          new_name = filename.replace(" ", "_")
          os.rename(os.path.join(folder, filename), os.path.join(folder, new_name))
  ```
- Python as a programming language - Used to build applications by organizing code using modules/classes/functions.
  ```python
  # Simple Student Management System
  class Student:
      def __init__(self, name, marks):
          self.name = name
          self.marks = marks

      def get_grade(self):
          if self.marks >= 90:
              return "A"
          elif self.marks >= 75:
              return "B"
          else:
              return "C"

  def main():
      students = [
          Student("Aman", 88),
          Student("Rahul", 95)
      ]

      for s in students:
          print(f"{s.name} got grade: {s.get_grade()}")

  if __name__ == "__main__":
      main()
  ```
- **Python is an object-oriented programming language like Java** – you can write class-based code, just like Java.
- **Python is interpreted**, but **Java is compiled** – it runs via an interpreter rather than being compiled to machine code directly.

Let's understand this more clearly with an analogy:

Analogy: Chef & Recipe

Concept	Python	Java
Cooking Style (OOP)	Object-Oriented	Object-Oriented
Cooking Method (Execution)	Interpreted *(live cooking)*	Compiled *(pre-cooked and served)*

6. Data types in Python

In Python, datatypes define what kind of value a variable can hold. Python is dynamically typed, so you don't have to declare types explicitly—they're inferred at runtime. Data Types Example Fundamental (or primitive) data types are the basic building blocks of data in Python. They are simple, core types from which more complex structures can be made.

Integer, float, boolean, None, and string are **primitive data types** because they represent a single value.

Other data types like list, tuple, set, frozenset and dictionary are often called **data structures or containers** because they hold multiple pieces of data together.

A literal is a value. A variable is a name pointing to that value.

A variable in Python is defined through assignment. There is no concept of declaring a variable outside of that assignment.

In [ ]:

Integers

Integer literals are created by any number without a decimal or complex component.

In [11]:
```python
# integers
x = 1234

print(f'Type : {type(x)}')
print(f'val : {x}')
```

```
Type : <class 'int'>
val : 1234
```

In Python, everything — even primitive data types like int, str, bool, etc. — is a class

Every value (even 5, "hello", True) is an instance of a class.

This makes Python powerful and flexible.

Floats

Float literals can be created by adding a decimal component to a number.

In [12]:
```python
# float
x = 1.0

print(f'Type : {type(x)}')
print(f'val : {x}')
```

```
Type : <class 'float'>
val : 1.0
```

Boolean

Boolean can be defined by typing True/False without quotes

In [13]:
```python
# boolean
b1 = True
b2 = False

print(f'Type : {type(b1)}')
print(f'val : {b1}')
```

```
Type : <class 'bool'>
val : True
```

In [19]:
```python
# Boolean evaluation
print(True and False)
print(True or False)
print(not True)
print('a' == 'a')
```

```
False
True
False
True
```

Strings

String literals can be defined with any of single quotes ('), double quotes (") or triple quotes (''' or """). All give the same result with two important differences.

If you quote with single quotes, you do not have to escape double quotes and vice-versa. If you quote with triple quotes, your string can span multiple lines.

In [14]:
```python
name = "Hello I'm double quotes !"          # Double quotes
print(f'name Type : {type(name)}')
print(f'name val : {name}')
#--------------
greet = 'Hello I\'m single quotes!'         # Single quotes
print(f'greet Type : {type(greet)}')
print(f'greet val : {greet}')
#--------------
multi_line = '''
Hi
This is Python
'''             # Triple quotes for multi-line strings
print(f'multi_line Type : {type(multi_line)}')
print(f'multi_line val : {multi_line}')
```

```
name Type : <class 'str'>
name val : Hello I'm double quotes !
greet Type : <class 'str'>
greet val : Hello I'm single quotes!
multi_line Type : <class 'str'>
multi_line val :
Hi
This is Python
```

Complex

Complex literals can be created by using the notation x + yj where x is the real component and y is the imaginary component.

In [15]:
```python
# complex numbers: note the use of `j` to specify the imaginary part
x = 1.0 - 2.0j
type(x)
```

Out[15]: complex

In [16]:
```python
print(x)
```

```
(1-2j)
```

In [17]:
```python
print(x.real, x.imag)
```

```
1.0 -2.0
```

Dynamic Typing

In Python, while the value that a variable points to has a type, the variable itself has no strict type in its definition. You can re-use the same variable to point to an object of a different type.

In [18]:
```python
ten = 10
print(ten)
```

```python
ten = 'ten'
print(ten)
```

```
10
ten
```

None Type

Represents the absence of a value.

In [20]:
```python
data = None
print(data)
print(type(data))
```

```
None
<class 'NoneType'>
```

Type Checking and Conversion

In [21]:
```python
a = "10"
print(type(a))
a = int(a)
print(type(a))
```

```
<class 'str'>
<class 'int'>
```

In [7]:
```python
#Converting integer to string
str(9)
```

Out[7]: '9'

In [8]:
```python
#Converting string to float
float("4.5")
```

Out[8]: 4.5

In [9]:
```python
#Converting bool to integer
int(True)
```

Out[9]: 1

Sometimes, conversion of a value may not be possible. For example, it is not possible to convert the variable greeting defined below to a number:

In [11]:
```python
greeting = "hello"
int(greeting) # ValueError: invalid literal for int() with base 10: 'hello'
```

However, in some cases, mathematical operators such as + and * can be applied on strings. The operator + concatenates multiple strings, while the operator * can be used to concatenate a string to itself multiple times:

In [12]:
```python
"Hi" + " there!"
```

Out[12]: 'Hi there!'

In [13]:
```python
"5" + '3'
```

Out[13]: '53'

In [14]:
```python
"5"*8
```

Out[14]: '55555555'

Block Structure and Whitespace

The code that is executed when a specific condition is met is defined in a "block."

In Python, the block structure is signalled by changes in indentation. Each line of code in a certain block level must be indented equally and indented more than the surrounding scope. The standard (defined in PEP-8) is to

use 4 spaces for each level of block indentation. Statements preceding blocks generally end with a colon (:).

Because there are no semi-colons or other end-of-line indicators in Python, breaking lines of code requires either a continuation character (\ as the last char) or for the break to occur inside an unfinished structure (such as open parentheses).

In [3]:
```python
total = 10 + 20 + 30 + \
        40 + 50
print(total)
```

150

In Python, primitive data types and container data types are fundamental concepts that help you store and manage data in different ways.

Since we have already seen Primitive data types in last lecture, let's see some common Container data types.

Containers in Python

In Python, containers are built-in data types that hold collections of items. They allow you to group multiple values together into a single object, making it easier to manage, organize, and manipulate data.

They are part of Python's core language — available by default without importing anything.

Let's see some common containers.

Type	Description	Mutable?	Ordered?	Duplicates?
list	Ordered collection of items	✅ Yes	✅ Yes	✅ Yes
tuple	Immutable ordered collection	❌ No	✅ Yes	✅ Yes
dict	Key-value pairs	✅ Yes	✅ Yes	❌ No (keys)
set	Unordered collection of unique elements	✅ Yes	❌ No	❌ No
frozenset	Immutable version of a set	❌ No	❌ No	❌ No
str	Immutable sequence of characters	❌ No	✅ Yes	✅ Yes

Let's see in brief some examples for each of these items.

In [1]:
```python
# 1. list – A Mutable Ordered Container

fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)  # ['apple', 'banana', 'cherry', 'orange']
```

['apple', 'banana', 'cherry', 'orange']

In [2]:
```python
# 2. tuple – An Immutable Ordered Container

coordinates = (10, 20)
# coordinates[0] = 30  # ❌ Error, tuples can't be modified
```

In [3]:
```python
# 3. dict – A Key-Value Pair Container

student = {"name": "Alice", "age": 22}
student["grade"] = "A"
print(student)  # {'name': 'Alice', 'age': 22, 'grade': 'A'}
```

{'name': 'Alice', 'age': 22, 'grade': 'A'}

In [4]:
```python
# 4. set – A Collection of Unique Elements

unique_numbers = {1, 2, 3, 2, 1}
print(unique_numbers)  # {1, 2, 3}
```

{1, 2, 3}

```
In [5]:   # 5. frozenset – Immutable Set

          fs = frozenset([1, 2, 3])
          # fs.add(4)  # ✘ Error: frozenset is immutable
```

```
In [6]:   # 6. str – Technically a Container of Characters

          message = "hello"
          print(message[1])  # 'e'
          # message[0] = 'H'  # ✘ Error: strings are immutable
```

e

We will see few containers in very details in the next lecture.

7. Expressions in Python

An expression is a combination of operators and operands that is interpreted to produce some other value.

In any programming language, an expression is evaluated as per the precedence and associativity of its operators.

So that if there is more than one operator in an expression, their precedence decides which operation will be performed first.

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.

Let's discuss different types of expressions in Python.

```
In [ ]:   #Operator types in python
          '''
          Exponent: **
          Remainder: %
          Multiplication: *
          Division: /
          Floor Division: //
          Addition: +
          Subtraction: -
          '''
```

```
In [31]:  # 1. Constant Expressions: These are the expressions that have constant values only.

          x = 15 + 1.3

          print(x)
```

16.3

```
In [32]:  # 2. Arithmetic Expressions: An arithmetic expression is a combination of numeric values, operators, 

          # Arithmetic Expressions
          x = 40
          y = 12

          add = x + y
          sub = x - y
          pro = x * y
          div = x / y

          print(add)
          print(sub)
          print(pro)
          print(div)
```

52
28
480
3.3333333333333335

```
In [33]:  # 3. Relational Expressions: Also called Boolean Expression because it produce a boolean output in the
          a = 21
          b = 13
          c = 40
```

```
d = 37

p = (a + b) >= (c - d)
print(p)
```

True

In [34]:
```
# 4. Logical Expressions: These are kinds of expressions that result in either True or False. It basi
P = (10 == 9)
Q = (7 > 5)

# Logical Expressions
R = P and Q
S = P or Q
T = not P

print(R)
print(S)
print(T)
```

False
True
True

In [35]:
```
# 5. Bitwise Expressions: These are the kind of expressions in which computations are performed at bi

a = 12

x = a >> 2
y = a << 1

print(x, y)
```

3 24

Python Operator Precedence and Associativity

Precedence Level	Operators	Description	Associativity
1 (Highest)	`()`	Parentheses (grouping)	N/A
2	`x[index]`, `x[attr]`, `x(...)`, `x(...)`	Subscription, attribute reference, function call	Left-to-right
3	`**`	Exponentiation	Right-to-left
4	`+x`, `-x`, `~x`	Unary plus, minus, bitwise NOT	Right-to-left
5	`*`, `/`, `//`, `%`	Multiplication, division, floor division, modulo	Left-to-right
6	`+`, `-`	Addition, subtraction	Left-to-right
7	`<<`, `>>`	Bitwise shift operators	Left-to-right
8	`&`	Bitwise AND	Left-to-right
9	`^`	Bitwise XOR	Left-to-right
10	`|`	Bitwise OR	Left-to-right
11	`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, identity, membership tests	Left-to-right
12	`not`	Logical NOT	Right-to-left
13	`and`	Logical AND	Left-to-right
14	`or`	Logical OR	Left-to-right
15	`if - else`	Conditional expressions	Right-to-left
16	`lambda`	Lambda expressions	Right-to-left
17 (Lowest)	`:=`	Assignment expression (walrus operator)	Right-to-left

```
In [21]: 2+3%4*2 # (2+ ((3%4) *2))
```

```
Out[21]: 8
```

```
In [30]: 2*(4/5)
```

```
Out[30]: 1.6
```

```
In [ ]: 1%2**3*2+1
```

8. Programming errors

1. Syntax Error

Syntax errors occur if the code is written in a way that it does not comply with the rules / standards / laws of the language (python in this case).

```
In [ ]: 9value = 2
```

2. Run-time Error

Run-time errors occur when a code is syntactically correct, but there are other issues with the code such as:

1. Misspelled or incorrectly capitalized variable and function names
2. Attempts to perform operations (such as math operations) on data of the wrong type (ex. attempting to subtract two variables that hold string values)
3. Dividing by zero
4. Attempts to use a type conversion function such as int on a value that can't be converted to an int

```
In [19]: multiplication_result = x * 4
```

📚 References

- https://nustat.github.io/Intro_to_programming_for_data_sci/Variable_expressions_statements.html