Python : Introduction et concepts de base

1. Introduction :

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

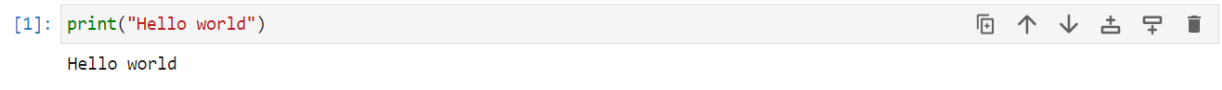
2. Installation :

For installing python we suggest using anaconda which is a desktop application that is included with every installation of Anaconda Distribution. It is built on top of conda, the open-source package and environment manager, and allows you to manage your packages and environments from a graphical user interface (GUI). This is especially helpful when you're not comfortable with the command line.

Link to install anaconda :
https://docs.anaconda.com/free/anaconda/install/windows/#installation

After installing the anaconda just open it then launch Jupyter.

3. Our first python code:



4. Data Types and Variables:

Variables in programming are essential, and they help us store different kinds of information. In Python, we have various data types to hold data. This chapter explores common data types like integers, floating-point numbers, strings, and booleans. We'll learn how to create and set variables, assign values, and perform basic operations with these data types.

By the end of this chapter, you'll understand Python's data types and variables, allowing you to build more advanced programs. Now, let's delve into numbers, strings, and boolean values.

These three are the basic components in many programming languages, serving as the foundation for creating diverse programs, from simple calculators to complex data analysis tools. We'll focus on numbers first, which represent numerical values in programming. Python has two types: **integers** (whole numbers like 1, 2, 3) and **floating-point** numbers (decimal numbers like 3.14, 2.5). The following code provides a demonstration:

```
[2]: # Integers
x=10
y=5

print(x+y)

# Floats
y=3.5
z=10.1

print(y+z)
```

```
15
13.6
```

Now, let's move on to **strings**. **Strings** are used to represent text in programming. They are created by enclosing text in single or double quotes. Strings can contain any type of character, including letters, numbers, and symbols.
Here's an example:

```
[3]: # Strings
name = "Khalid" # 'Khalid'
message = "Hello "+name

print(message)
```

```
Hello Khalid
```

Strings are very flexible and can be used in many ways. For instance, you can use string formatting to put values into a string:

```
[4]: name = 'Khalid'
     age = 24

     message = 'hello my name is {} and I am {} yo !'.format(name, age)

     print(message)
```

```
hello my name is Khalid and I am 24 yo !
```

Now, let's discuss **boolean** values. These are used to show either true or false conditions in programming. They're commonly used in if statements and loops to manage how a program runs. In Python, the boolean values are True and False. For instance, consider the following example:

```
x = 3
y = 5

is_greater = x>y
print(is_greater)
```

```
False
```

Certainly! Here's a reformulation of the information in a format that you can share as a course material for your students:

❖ Variables and naming conventions

- What are Variables?

In the world of programming, variables are like containers that hold information. Imagine them as labeled boxes in which we can store and manipulate different types of data—numbers, words, yes/no answers, and more.

- Why do we Need Variables?

Variables make our programs flexible and adaptable. For instance, think about storing a user's name or age. Without variables, our programs would be rigid and less powerful.

Basic Example:

```
# declaring and initializing variable
x = 10
# changing the value of the variable
x = 5
# decalring y variable and using the previous variable x to calculate y
y = x * 2

print(x)
print(y)
```

```
5
10
```

Here, we create a variable `x`, change its value, and use it in a simple calculation for another variable `y`.

❖ Naming Conventions

- Importance of Naming Conventions

Naming conventions are like the grammar of programming. They make our code readable and understandable. Imagine reading a book without proper punctuation—it would be confusing!

- Common Naming Conventions in Python

- Descriptive Names: Reflect the purpose of the variable.
- Lowercase with Underscores: Enhances readability (`my_variable`).
- Avoid Single-letter Names: Unless used in a clear mathematical context.
- Uppercase for Constants: Clearly distinguish constants (`MAX_SIZE`, `PI`).

Example:

```
# Good naming conventions
user_name = "John"
age_of_user = 25

# Avoiding naming conventions
n = "John"
aou = 25
```

❖ Best Practices

- Consistency is Key

Maintain a consistent naming style throughout your codebase. Consistency makes it easier for anyone to understand and work with the code.

- Clarity Over Conciseness

Prioritize clarity in variable names over brevity. A longer, descriptive name is better if it makes the code more understandable.

- Context Matters

Consider the context in which the code will be used. Shorter names may be acceptable for short-lived variables in small scopes, but always aim for clarity.

- Documentation and Refactoring

In addition to good naming conventions, provide clear comments and documentation for complex sections of code. Don't hesitate to refactor if the code becomes hard to read or maintain.

❖ Type conversion and casting

Type conversion and casting are important concepts in programming that help developers change the type of data they're working with. Let's break it down.

Type conversion is like transforming data from one form to another. In Python, you can use built-in functions like int(), float(), and str() to do this. For instance, if you have the text "30" and you want it to be a number for math or comparisons, you can convert it to an integer using int().

```
age_as_string = "30"

age = int(age_as_string)

print(age_as_string*2)
print(age*2)
```

```
3030
60
```

Now, casting is a bit different. It's when you explicitly switch a value from one type to another. This is handy when you want to change the type of a variable right where it is. For example, if you have the number 3.14 but only want the whole part without the decimal, you can cast it to an integer using int().

```
Pi = 3.14
Pi = int(Pi)
print(Pi)
```

```
3
```

❖ Operators and Expressions

Operators are symbols or keywords that perform operations on values, while expressions combine values, variables, and operators to evaluate to a single value. Understanding these elements is crucial for writing programs that calculate and manipulate data. Whether you're a beginner or an experienced programmer, this chapter covers the fundamental concepts and techniques needed to work proficiently with operators and expressions.

We'll start by looking at different types of operators and their precedence rules. Then, we'll explore expressions, learning how to create complex calculations and comparisons. Finally, we'll discuss common pitfalls and best practices for working with operators and expressions. Let's begin!

❖ Arithmetic Operators

Arithmetic operators are fundamental in programming, allowing you to perform mathematical calculations on numerical values. Here are the basic arithmetic operators in Python:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulo (%)
- Exponentiation (**)

❖ Comparison Operators

Comparison operators help compare values to determine equality, inequality, or order. In Python, these operators include:

- Equal to (==)
- Not equal to (!=)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

❖ Logical Operators

Logical operators combine conditions or expressions to evaluate True or False. In Python, these operators are:

- And (and)
- Or (or)
- Not (not)

❖ Order of Operations

Order of operations, or operator precedence, defines the sequence in which operations are performed in an expression.

1. Parentheses ()
2. Exponentiation **
3. Multiplication *, Division /, Integer Division //, and Modulo %
4. Addition + and Subtraction -

5. Control Flow

Control flow in programming refers to the sequence in which code is executed. It allows you to make decisions, repeat actions, and execute code based on specific conditions. In Python, control flow is essential for creating flexible and powerful code. This chapter explores various control flow statements in Python and demonstrates how to use them effectively.

❖ If/Else Statements:

If/else statements are fundamental to control flow in Python. They enable the execution of different code blocks based on specific conditions. For instance, an if/else statement can be used to determine if a person is an adult or a minor based on their age. It's crucial to use the correct syntax, such as == for value comparison and = for value assignment.

```
x = int(input("Please enter an integer: "))

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

❖ For and While Loops:

For and while loops are common tools for automating repetitive tasks in Python. A for loop iterates over a sequence, like a list or string, executing code for each element. Meanwhile, a while loop continues executing code as long as a specified condition is true. It's important to avoid infinite loops by updating the condition appropriately.

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
```

```
cat 3
window 6
defenestrate 12
```

```
count = 0
while count < 5:
    print(count)
    count = count + 1
```

```
0
1
2
3
4
```

❖ Break, Continue, and Pass Statements:

These statements are key to flow control in Python. The break statement exits a loop prematurely when a specific condition is met. The continue statement skips iterations based on a condition, and the pass statement acts as a placeholder for code yet to be implemented. Effectively using these statements enhances the efficiency and flexibility of your Python programs.

```python
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

```python
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found an odd number", num)
```

```
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

```python
while True:
    pass
```

Understanding control flow is foundational to writing efficient, readable, and maintainable code. Whether you're working on a simple script or a complex program, mastering control flow in Python is essential for creating robust and effective solutions.

❖ Functions:

Functions are blocks of reusable code designed for specific tasks. They allow you to break down your program into more manageable parts, enhancing readability and maintainability. In Python, you define functions with the 'def' keyword, specifying arguments and return values.

❖ Modules:

Modules are collections of related functions and variables that can be reused in various programs. They help organize code and share functionality across projects. In Python, you import modules with the 'import' statement.

In this section, we'll explore functions and modules, covering syntax, parameters, return values, and creating your own modules. We'll also discuss built-in functions and modules provided by Python and how to use them.
By the end, you'll have a solid grasp of employing functions and modules for creating flexible, efficient, and reusable code.

❖ Creating and Calling Functions:

Functions are crucial for breaking down code into manageable parts, making it readable and maintainable. To create a function in Python, use 'def' followed by the function name and parameters.
For example, a function called 'greet' takes a parameter 'name' and prints a greeting when called.

```
def greet():
    print('salaaaam !!')

greet()
```

```
salaaaam !!
```

❖ Built-in Functions vs User-Defined Functions:

Python has built-in functions and user-defined functions. Built-in functions come pre-defined, providing a range of functionalities like 'print()' and 'len()'.

Here are some other examples of built-in functions in Python:

- abs() – Returns the absolute value of a number
- sum() – Returns the sum of a sequence of numbers
- type() – Returns the data type of a given object

User-defined functions are created by programmers for specific tasks, offering flexibility and customization.

Use built-in functions for standard functionalities and user-defined functions for task-specific needs.

❖ Importing and Using Modules:

A module is a file containing Python code defining functions, classes, and objects. Modules help organize and reuse code across programs.

Python comes with built-in modules like 'math' and 'random'. Import a module using 'import' and access its attributes using dot notation.

For example, importing the 'math' module and printing its 'pi' attribute.

```
import math

print(math.pi)
```

```
3.141592653589793
```

You can also import specific functions or classes using 'from,' such as importing 'randint' from the 'random' module.

```
from random import randint

print(randint(1,10))
```

```
9
```

If a needed module isn't built into Python, install it using a package manager like 'pip' and import it into your code.

```
!pip install requests
```

For instance, importing the 'requests' module for making HTTP requests.

Using modules in your Python code saves time by leveraging pre-built functionality. Next time you face a task beyond the basics, check if a module can assist you.

Exemple :

```
def say_hello(tahia = "Salam",name='khalid', age=24) :
    message = tahia+" "+name+" !! You are "+str(age)+" yo."
    return message
```

```
say_hello()
```

```
'Salam khalid !! You are 24 yo.'
```

```
say_hello(tahia = "Hello")
```

```
'Hello khalid !! You are 24 yo.'
```

```
say_hello(name="ismail", age=22)
```

```
'Salam ismail !! You are 22 yo.'
```

```
say_hello(tahia='Azul', name='hassan', age=30)
```

```
'Azul hassan !! You are 30 yo.'
```