**Data Types:**

In programming, a data type is a classification or categorization that specifies which type of value a variable can hold. Data types are essential because they determine how data is stored in memory and what operations can be performed on that data. Python, like many programming languages, supports several built-in data types. Here are some of the common data types in Python:

1. **Numeric Data Types:**
   * **int**: Represents integers (whole numbers). Example: x = 5
   * **float**: Represents floating-point numbers (numbers with decimal points). Example: y = 3.14
   * **complex**: Represents complex numbers. Example: z = 2 + 3j
2. **Sequence Types:**
   * **str**: Represents strings (sequences of characters). Example: text = "Hello, World"
   * **list**: Represents lists (ordered, mutable sequences). Example: my\_list = [1, 2, 3]
   * **tuple**: Represents tuples (ordered, immutable sequences). Example: my\_tuple = (1, 2, 3)
3. **Mapping Type:**
   * **dict**: Represents dictionaries (key-value pairs). Example: my\_dict = {'name': 'John', 'age': 30}
4. **Set Types:**
   * **set**: Represents sets (unordered collections of unique elements). Example: my\_set = {1, 2, 3}
   * **frozenset**: Represents immutable sets. Example: my\_frozenset = frozenset([1, 2, 3])
5. **Boolean Type:**
   * **bool**: Represents Boolean values (True or False). Example: is\_valid = True
6. **Binary Types:**
   * **bytes**: Represents immutable sequences of bytes. Example: data = b'Hello'
   * **bytearray**: Represents mutable sequences of bytes. Example: data = bytearray(b'Hello')
7. **None Type:**
   * **NoneType**: Represents the None object, which is used to indicate the absence of a value or a null value.
8. **Custom Data Types:**
   * You can also define your custom data types using classes and objects.

**Strings :**

**1. String Data Type in Python:**

* In Python, a string is a sequence of characters, enclosed within single (' '), double (" "), or triple (''' ''' or """ """) quotes.
* Strings are immutable, meaning you cannot change the characters within a string directly. Instead, you create new strings.
* You can access individual characters in a string using indexing, e.g., my\_string[0] will give you the first character.
* Strings support various built-in methods, such as len(), upper(), lower(), strip(), replace(), and more, for manipulation.

**2. String Manipulation and Formatting:**

* Concatenation: You can combine strings using the + operator.
* Substrings: Use slicing to extract portions of a string, e.g., my\_string[2:5] will extract characters from the 2nd to the 4th position.
* String interpolation: Python supports various ways to format strings, including f-strings (f"...{variable}..."), %-formatting ("%s %d" % ("string", 42)), and str.format().
* Escape sequences: Special characters like newline (\n), tab (\t), and others are represented using escape sequences.
* String methods: Python provides many built-in methods for string manipulation, such as split(), join(), and startswith().

**Numeric Data Type :**

**1. Numeric Data Types in Python (int, float):**

* Python supports two primary numeric data types: int for integers and float for floating-point numbers.
* Integers are whole numbers, and floats can represent both whole and fractional numbers.
* You can perform arithmetic operations on these types, including addition, subtraction, multiplication, division, and more.
* Be aware of potential issues with floating-point precision, which can lead to small inaccuracies in calculations.
* Python also provides built-in functions for mathematical operations, such as abs(), round(), and math module for advanced functions.

**Regex :**

**1. Regular Expressions for Text Processing:**

* Regular expressions (regex or regexp) are a powerful tool for pattern matching and text processing.
* The re module in Python is used for working with regular expressions.
* Common metacharacters: . (any character), \* (zero or more), + (one or more), ? (zero or one), [] (character class), | (OR), ^ (start of a line), $ (end of a line), etc.
* Examples of regex usage: matching emails, phone numbers, or extracting data from text.
* re module functions include re.match(), re.search(), re.findall(), and re.sub() for pattern matching and replacement.

**Keywords in Python:**

Keywords are reserved words in Python that have predefined meanings and cannot be used as variable names or identifiers. These words are used to define the structure and logic of the program. They are an integral part of the Python language and are case-sensitive, which means you must use them exactly as specified.

Here are some important Python keywords:

1. **and**: It is a logical operator that returns True if both operands are true.
2. **or**: It is a logical operator that returns True if at least one of the operands is true.
3. **not**: It is a logical operator that returns the opposite of the operand's truth value.
4. **if**: It is used to start a conditional statement and is followed by a condition that determines whether the code block is executed.
5. **else**: It is used in conjunction with if to define an alternative code block to execute when the if condition is False.
6. **elif**: Short for "else if," it is used to check additional conditions after an if statement and is used in combination with if and else.
7. **while**: It is used to create a loop that repeatedly executes a block of code as long as a specified condition is true.
8. **for**: It is used to create a loop that iterates over a sequence (such as a list, tuple, or string) and executes a block of code for each item in the sequence.
9. **in**: Used with for, it checks if a value is present in a sequence.
10. **try**: It is the beginning of a block of code that is subject to exception handling. It is followed by except to catch and handle exceptions.
11. **except**: Used with try, it defines a block of code to execute when an exception is raised in the corresponding try block.
12. **finally**: Used with try, it defines a block of code that is always executed, whether an exception is raised or not.
13. **def**: It is used to define a function in Python.
14. **return**: It is used within a function to specify the value that the function should return.
15. **class**: It is used to define a class, which is a blueprint for creating objects in object-oriented programming.
16. **import**: It is used to import modules or libraries to access their functions, classes, or variables.
17. **from**: Used with import to specify which specific components from a module should be imported.
18. **as**: Used with import to create an alias for a module, making it easier to reference in the code.
19. **True**: It represents a boolean value for "true."
20. **False**: It represents a boolean value for "false."
21. **None**: It represents a special null value or absence of value.
22. **is**: It is used for identity comparison, checking if two variables refer to the same object in memory.
23. **lambda**: It is used to create small, anonymous functions (lambda functions).
24. **with**: It is used for context management, ensuring that certain operations are performed before and after a block of code.
25. **global**: It is used to declare a global variable within a function's scope.
26. **nonlocal**: It is used to declare a variable as nonlocal, which allows modifying a variable in an enclosing (but non-global) scope.

# Understanding Variables in Python:

In Python, a variable is a named storage location used to store data. Variables are essential for programming as they allow us to work with data, manipulate it, and make our code more flexible and reusable.

# Assigning a value to a variable

my\_variable = 42

# Accessing the value of a variable

print(my\_variable) # Output: 42

### Variable Scope and Lifetime:

**Variable Scope:** In Python, variables have different scopes, which determine where in the code the variable can be accessed. There are mainly two types of variable scopes:

1. **Local Scope:** Variables defined within a function have local scope and are only accessible inside that function.

def my\_function():

x = 10 # Local variable

print(x)

my\_function()

print(x) # This will raise an error since 'x' is not defined outside the function.

**Global Scope:** Variables defined outside of any function have global scope and can be accessed throughout the entire code.

y = 20 # Global variable

def another\_function():

print(y) # This will access the global variable 'y'

another\_function()

print(y) # This will print 20

**Variable Lifetime:** The lifetime of a variable is determined by when it is created and when it is destroyed or goes out of scope. Local variables exist only while the function is being executed, while global variables exist for the entire duration of the program.

### Variable Naming Conventions and Best Practices:

It's important to follow naming conventions and best practices for variables to write clean and maintainable code:

* Variable names should be descriptive and indicate their purpose.
* Use lowercase letters and separate words with underscores (snake\_case) for variable names.
* Avoid using reserved words (keywords) for variable names.
* Choose meaningful names for variables.

# Good variable naming

user\_name = "John"

total\_items = 42

# Avoid using reserved words

class = "Python" # Not recommended

# Use meaningful names

a = 10 # Less clear

num\_of\_students = 10 # More descriptive

### Practice Exercises and Examples:

#### Example: Using Variables to Store and Manipulate Configuration Data in a DevOps Context

In a DevOps context, you often need to manage configuration data for various services or environments. Variables are essential for this purpose. Let's consider a scenario where we need to store and manipulate configuration data for a web server.

# Define configuration variables for a web server

server\_name = "my\_server"

port = 80

is\_https\_enabled = True

max\_connections = 1000

# Print the configuration

print(f"Server Name: {server\_name}")

print(f"Port: {port}")

print(f"HTTPS Enabled: {is\_https\_enabled}")

print(f"Max Connections: {max\_connections}")

# Update configuration values

port = 443

is\_https\_enabled = False

# Print the updated configuration

print(f"Updated Port: {port}")

print(f"Updated HTTPS Enabled: {is\_https\_enabled}")

In this example, we use variables to store and manipulate configuration data for a web server. This allows us to easily update and manage the server's configuration in a DevOps context.

# Python Functions, Modules and Packages

## 1. Differences Between Functions, Modules, and Packages

### Functions

A function in Python is a block of code that performs a specific task. Functions are defined using the def keyword and can take inputs, called arguments. They are a way to encapsulate and reuse code.

def greet(name):

return f"Hello, {name}!"

message = greet("Alice")

print(message)

In this example, greet is a function that takes a name argument and returns a greeting message.

### Modules

A module is a Python script containing Python code. It can define functions, classes, and variables that can be used in other Python scripts. Modules help organize and modularize your code, making it more maintainable.

Suppose you have a Python file named my\_module.py:

# my\_module.py

def square(x):

return x \*\* 2

pi = 3.14159265

You can use this module in another script:

import my\_module

result = my\_module.square(5)

print(result)

print(my\_module.pi)

In this case, my\_module is a Python module containing the square function and a variable pi.

### Packages

A package is a collection of modules organized in directories. Packages help you organize related modules into a hierarchy. They contain a special file named \_\_init\_\_.py, which indicates that the directory should be treated as a package.

**Example:**

Suppose you have a package structure as follows:

my\_package/

\_\_init\_\_.py

module1.py

module2.py

You can use modules from this package as follows:

from my\_package import module1

result = module1.function\_from\_module1()

In this example, my\_package is a Python package containing modules module1 and module2.

## 2. How to Import a Package

Importing a package or module in Python is done using the import statement. You can import the entire package, specific modules, or individual functions/variables from a module.

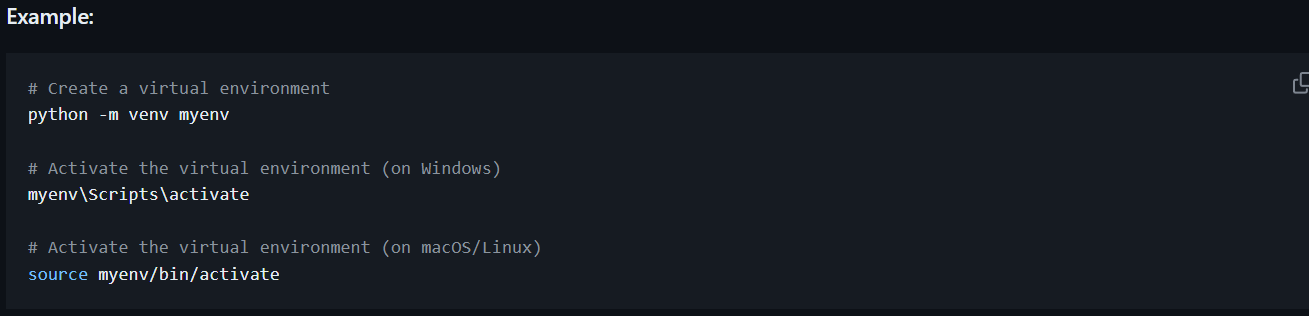


In this example, we import the math module and then use functions and variables from it. You can also import specific elements from modules using the from module import element syntax.

## 3. Python Workspaces

Python workspaces refer to the environment in which you develop and run your Python code. They include the Python interpreter, installed libraries, and the current working directory. Understanding workspaces is essential for managing dependencies and code organization.

Python workspaces can be local or virtual environments. A local environment is the system-wide Python installation, while a virtual environment is an isolated environment for a specific project. You can create virtual environments using tools like virtualenv or venv.



Once activated, you work in an isolated workspace with its Python interpreter and library dependencies.