Scripting Lesson 6: Introduction to Python

Course	DevOps Engineering
Topic / Session number	Scripting / Lesson 6
Objectives / Goals	1. What is Python? 2. Python in DevOps 3. Variables in Programming a. What Are Variables b. How Variables Work 4. Data Types in Programming 5. Type Conversion - Casting 6. Working with Strings a. String Methods

Objective 1: What is Python?

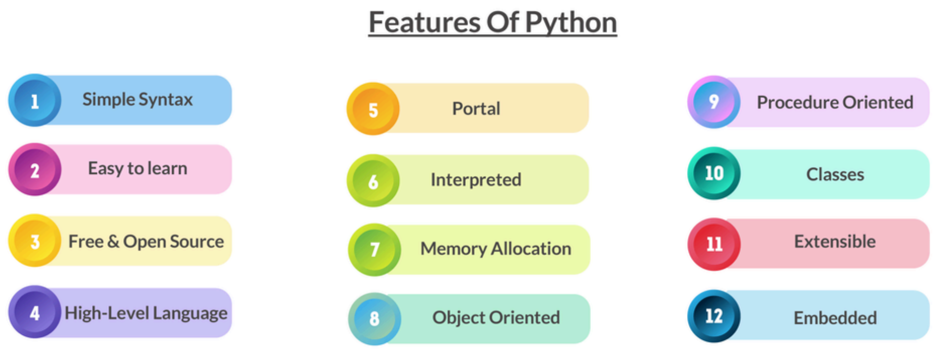
Python is a high-level, interpreted programming language known for its readability, simplicity, and versatility. It was created by **Guido van Rossum** and first released in **1991**. Python is designed to be easy to understand and use, which makes it an excellent choice for both beginners and experienced programmers. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

Key Features of Python

1. **Readable Syntax**: Python uses clear, easy-to-read syntax that emphasizes readability. It's often referred to as a language that reads like English, which helps reduce the cost of program maintenance.
2. **Interpreted**: Python code is executed line-by-line by the Python interpreter. This means you don't need to compile your code before running it, which speeds up development.
3. **Dynamically Typed**: Python doesn't require you to declare the type of a variable when you create it. The type is determined at runtime, making the language more flexible.

```python
x = 10   # x is an integer
x = "Hello"   # x is now a string
```

4. **Cross-platform**: Python can run on various platforms, including Windows, macOS, and Linux, without needing any changes to the code.
5. **Large Standard Library**: Python comes with a rich set of built-in libraries and frameworks that simplify many tasks, such as working with files, connecting to web servers, managing databases, and more.
6. **Community Support**: Python has a massive, active community that contributes to its growth. There are thousands of third-party libraries, frameworks, and tools available to extend its capabilities.



Example of Same Code in Different Programming Languages

Python

```python
print("Hello, World!")
```

Java

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Go

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

Analogy

Think of Python as a **powerful, flexible tool** (like a Swiss army knife) for building software. Its ease of use and broad capabilities make it an ideal language for many different applications, from simple scripts to complex web applications, data analysis, artificial intelligence, and more.

Python's focus on readability and simplicity means you don't need to worry about complex syntax, allowing you to focus on solving the problem at hand. It's like a **user-friendly interface** that helps you get things done efficiently.

Why is Python Popular?

1. **Beginner-Friendly**: Its clear syntax makes it one of the easiest programming languages for newcomers.
2. **Versatile**: You can use Python for web development, automation, data science, machine learning, artificial intelligence, software development, and much more.
3. **Huge Ecosystem**: There are vast libraries like **NumPy** (for numerical computing), **Pandas** (for data analysis), **Flask** and **Django** (for web development), and **TensorFlow** (for machine learning), among others.

★ Objective 2: Python in DevOps

Let’s break it down with some simple, everyday examples that will help you understand how Python works/used in DevOps.

Automation and Scripting (The Smart Robot)

- **Analogy**: Imagine a robot that helps you clean your house. Instead of you doing the cleaning every day, you program the robot to do it for you. In DevOps, Python is like this smart robot that automatically handles repetitive tasks.
- **Real-world example**: Python automates setting up new servers, installing software, and deploying updates, just like the robot automatically cleans your house without you lifting a finger.

Server Provisioning (Setting Up a New Desk)

- **Analogy**: Picture setting up a new desk at work. You need a chair, a computer, a lamp, and other items to make it functional. Instead of doing this by hand, Python is like an assistant who sets up the desk for you, making sure everything is in place.
- **Real-world example**: Python automates the process of preparing a new server, making sure the necessary software is installed and configurations are correct, just like the assistant setting up everything for your desk.

CI/CD Pipelines (The Smooth Conveyor Belt)

- **Analogy**: Think of a conveyor belt in a factory where products move automatically from one station to another until they’re ready for shipment. Python helps the conveyor belt move smoothly by making sure each step happens in the right order.
- **Real-world example**: In DevOps, Python automates the flow of code through testing, building, and deployment, making sure everything works smoothly without manual work.

Configuration Management (The Recipe Book)

- **Analogy**: Imagine a chef following a recipe book to make the same dish every time. In DevOps, Python is like a recipe book, making sure every server is set up the same way, following the same steps every time.
- **Real-world example**: Python helps manage the setup of servers by ensuring they have the same software and settings, just like a chef makes the same dish with the same ingredients every time.

Cloud Infrastructure Automation (The Digital Builder)

- **Analogy**: Think of a digital builder who can instantly build houses by clicking buttons on a computer. Instead of hiring workers and filling out forms, this builder (Python) creates everything you need with just a few clicks.
- **Real-world example**: Python automates creating and managing cloud resources like virtual machines or storage, just like the builder automatically constructing houses in a virtual world.

Monitoring and Alerting (The Security Guard)

- **Analogy**: Imagine a security guard walking around your building, making sure everything is safe. If there’s a problem, the guard calls for help. Python works like this security guard, constantly checking the health of servers and sending alerts when something’s wrong.
- **Real-world example**: Python monitors the servers, checking things like performance, and sends an alert if something goes wrong, just like the guard calling for backup if there’s a problem.

Containerization and Orchestration (The Moving Company)

- **Analogy**: Imagine you’re moving to a new house. You pack all your things in boxes (containers) and the moving company helps transport everything to your new home. Python helps manage these boxes and makes sure everything gets to the right place.
- **Real-world example**: Python automates the packing of applications into containers (like Docker), and moves them to different environments. It also helps scale them up or down, just like the moving company helping with your move.

Summary

Python is like your helpful assistant in DevOps, doing all the repetitive work for you. Whether it's cleaning the house, setting up a desk, following a recipe, building houses, keeping things safe, or helping with a move — Python makes DevOps tasks easier and faster!

Objective 3: Variables in Programming

A **variable** in programming is a named storage location in a computer's memory that can hold a value, which may change during the execution of a program. The value of a variable is associated with a specific data type (such as integer, string, etc.) and is accessible by its name.

What Are Variables

Think of a **variable** as a **storage box** in a warehouse. This box has a **label** (the variable name) so you can easily find it. Inside the box, you can place various items (the value), and you can change the contents of the box whenever you need to.

- **Label** = **Variable Name**: Just like you would label a box with a name to know what's inside, a variable has a name so that you can refer to it in your code.
- **Contents** = **Value**: The contents inside the box are the actual data stored in the variable. This could be a number, a piece of text, or even a list of items.
- **Changing the contents** = **Reassigning the value**: Just as you can remove the contents from a box and put something else inside, you can change the value stored in a variable as your program runs.
  Example:
  If you have a variable called `age`, it’s like having a box labeled "age," and initially, you place the number `25` inside it. Later on, you might update the contents of the box to `26` as time passes (or the user enters their age again).

  In this way, variables allow you to store and manipulate data dynamically throughout the life of your program.

Key Characteristics of Variables

1. **Name**: The variable has a name that allows you to refer to it. This name is how the variable is accessed in the code.
2. **Value**: The value stored in a variable can be of different types, such as integers, floating-point numbers, strings, or more complex objects.
3. **Data Type**: Every variable is associated with a data type (e.g., integer, string, boolean, list) that defines what kind of data the variable can hold and what operations can be performed on it.
4. **Memory Location**: The variable refers to a specific location in the computer's memory where the value is stored.

Example in Python

```python
# Assigning a value to a variable
age = 25          # 'age' is a variable storing the integer 25
name = "Alice"    # 'name' is a variable storing the string "Alice"
height = 5.9      # 'height' is a variable storing the float 5.9

# Using the variables
print(name)       # Outputs: Alice
print(age)        # Outputs: 25
print(height)     # Outputs: 5.9
```

How Variables Work

- **Initialization**: You give a variable an initial value when you create it. This is called "initializing" the variable.
- **Assignment**: You can later change the value of the variable through "assignment." The new value overwrites the old one.

  ```python
  age = 25      # Initial value
  age = 30      # Reassigned to a new value (30)
  ```

- **Accessing**: You can use the variable's name to access its value in your program.

```python
print(age)    # This will output: 30
```

Naming Variables:

- **Valid Names**: Variable names should start with a letter (a-z, A-Z) or an underscore (_) and can include numbers. They are case-sensitive, meaning `age`, `Age`, and `AGE` would be considered different variables.

```python
user_name = "John"    # Valid
userName = "Alice"    # Valid
_username = "Bob"     # Valid
age1 = 20             # Valid
```

- **Invalid Names**: Variable names cannot start with a number and cannot contain spaces or special characters (except for underscores).

```python
1age = 25             # Invalid: starts with a number
user name = "Alice"   # Invalid: contains a space
my@age = 30           # Invalid: contains a special character (@)
```

Why Are Variables Important?

- **Reusability**: Variables allow you to store data in a way that can be reused throughout your program. Instead of hardcoding values in multiple places, you use variables to store them in one location and reference them where needed.

```python
price = 50   # Price of a product
tax = 0.08   # Sales tax rate

total = price + (price * tax)   # Use variables to calculate total price
```

- **Dynamic Behavior**: Variables enable programs to change behavior dynamically based on user input or other conditions during execution. For example, a variable may store the score in a game that changes as the player progresses.
- **Maintainability**: Using variables makes the code easier to maintain because you can change a value in one place (the variable assignment) rather than having to modify it everywhere it's used.

In Summary:

A **variable** is a fundamental concept in programming that acts as a container for data. It gives a name to a value stored in memory, which allows the value to be accessed, modified, and reused throughout the program. Variables are essential for writing flexible and efficient code.

★ Objective 4: Data Types in Programming

A **data type** in programming defines the kind of data a variable can hold, as well as the operations that can be performed on that data. It specifies how the computer should interpret and store the value in memory.

In simple terms, data types determine **what kind of value** you're working with and what you can do with that value. For example, whether the value is a whole number, a piece of text, or a more complex structure like a list or a dictionary.

Here are some of the most common **data types in python** you'll encounter:

Integer (`int`)

- **Definition**: Represents whole numbers (both positive and negative) without a fractional part.
- **Example**:

```python
age = 25
```

```python
year = 2025
count = -100
```

Float (`float`)

- **Definition**: Represents numbers that have a decimal point or are in scientific notation.
- **Example**:

```python
pi = 3.14159
temperature = -5.2
weight = 72.5
```

String (`str`)

- **Definition**: Represents a sequence of characters (letters, numbers, punctuation, etc.). It’s essentially text.
- **Example**:

```python
name = "Alice"
greeting = "Hello, World!"
phone_number = "123-456-7890"
```

Boolean (`bool`)

- **Definition**: Represents one of two values: `True` or `False`. It’s often used for making decisions in code (e.g., conditions in if statements).
- **Example**:

```python
is_active = True
is_logged_in = False
```

List (`list`)

- **Definition**: Represents an ordered collection of items, which can be of different data types. Lists are mutable, meaning their contents can be changed after creation.
- **Example**:

```python
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed_list = [1, "apple", 3.14, True]
```

Tuple (`tuple`)

- **Definition**: Similar to a list but immutable. Once a tuple is created, its contents cannot be changed.
- **Example**:

```python
coordinates = (10, 20)
rgb_color = (255, 0, 0)
```

Dictionary (`dict`)

- **Definition**: Represents an unordered collection of key-value pairs. The key is unique, and you can use it to look up its corresponding value.
- **Example**:

```python
person = {"name": "Alice", "age": 30, "city": "New York"}
car = {"make": "Toyota", "model": "Corolla", "year": 2020}
```

Set (`set`)

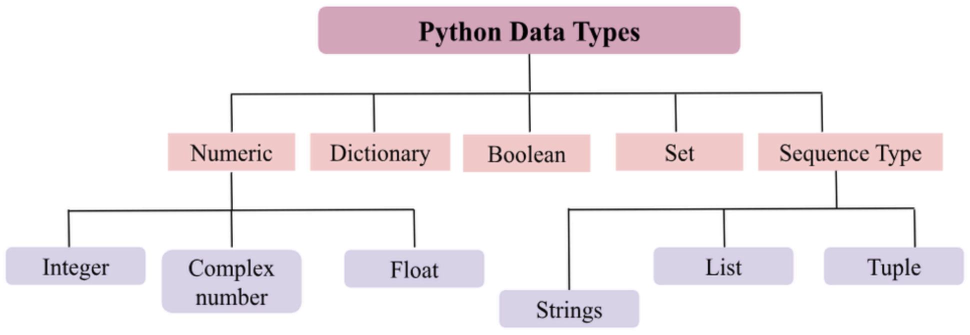
- **Definition**: Represents an unordered collection of unique items. Sets do not allow duplicate values and do not maintain the order of elements.
- **Example**:

```python
fruits = {"apple", "banana", "cherry"}
numbers = {1, 2, 3, 4, 5}
```

None (`NoneType`)

- **Definition**: Represents the absence of a value or a null value. In Python, `None` is a special constant used to denote the absence of a value.
- **Example**:

```python
result = None
is_empty = None
```



Why Are Data Types Important?

1. **Memory Management**: Different data types require different amounts of memory. For example, an integer might take up less memory than a string or a list.
2. **Operations**: The data type of a variable determines which operations can be performed on it. For instance, you can add two integers, but you can't add an integer to a string unless you specifically convert them to compatible types.
   - Valid:

     ```python
     total = 5 + 3  # Valid, total is 8
     ```

   - Invalid:

     ```python
     result = "5" + 3  # Error, can't add string to an integer
     ```

3. **Type Safety**: Many programming languages (like Python, Java, and C++) check the data type of values before performing operations to avoid errors, ensuring that the operations you perform make sense.
4. **Code Optimization**: Using the correct data type helps the program run efficiently and avoids unnecessary memory usage.

Conclusion

Data types are foundational to programming because they help define how data is stored, processed, and manipulated in your program. By choosing the right data type for your variables and understanding the operations allowed with each type, you can write more efficient, reliable, and understandable code.

★ Objective 5: Type Conversion - Casting

Type Conversion (Casting)

In many languages, including Python, you can **convert** a value from one data type to another. This is called **type casting**. For example, converting a float to an integer, or a string to a number.

Example in Python:

```python
# Converting a string to an integer
num_str = "10"
num_int = int(num_str)  # Converts string "10" to integer 10

# Converting an integer to a float
num = 5
num_float = float(num)  # Converts integer 5 to float 5.0

# Converting a float to a string
pi = 3.14159
pi_str = str(pi)  # Converts float 3.14159 to string "3.14159"
```

In Python, you can get type information about variables, expressions, or objects using built-in functions like `type()` and `isinstance()`. Here’s a detailed breakdown of how to retrieve type information along with examples.

Using `type()` Function

The `type()` function returns the type of an object or variable. It can be used to check the type of built-in objects, user-defined classes, and more.

Syntax:

```python
type(object)
```

Checking the type of a variable

```python
x = 42
print(type(x))  # Output: <class 'int'>
```

In this example, the type of `x` is `int`, as `42` is an integer.

Checking the type of a string

```python
name = "Alice"
print(type(name))  # Output: <class 'str'>
```

Here, the type of `name` is `str`, because `"Alice"` is a string.

Checking the type of a list

```python
numbers = [1, 2, 3]
print(type(numbers))  # Output: <class 'list'>
```

The `numbers` variable is a list, as indicated by `<class 'list'>`.

Checking the type of a custom class instance

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

p = Person("John")
print(type(p))  # Output: <class '__main__.Person'>
```

In this case, the type of `p` is `<class '__main__.Person'>`, indicating it is an instance of the `Person` class defined in the main module.

Using `isinstance()` Function

The `isinstance()` function is used to check if an object is an instance of a specific class or subclass. It's more flexible than `type()` when dealing with class inheritance.

Syntax:

```python
isinstance(object, classinfo)
```

- `object`: The object you want to check (in Python every value is an object).
- `classinfo`: The class, type, or tuple of classes to check against.

Checking if a variable is an instance of a specific type

```python
x = 42
print(isinstance(x, int))  # Output: True
print(isinstance(x, str))  # Output: False
```

Here, `x` is an integer (`int`), so `isinstance(x, int)` returns `True`, and `isinstance(x, str)` returns `False`.

Checking if an object is an instance of a class

```python
class Animal:
    pass

class Dog(Animal):
    pass

dog = Dog()
print(isinstance(dog, Dog))    # Output: True
print(isinstance(dog, Animal))  # Output: True
print(isinstance(dog, object))  # Output: True
```

In this case, `dog` is an instance of `Dog`, which is a subclass of `Animal`, and both classes ultimately inherit from the base `object` class.

Using `isinstance()` with multiple classes

```python
x = 42
print(isinstance(x, (int, float)))  # Output: True
print(isinstance(x, (str, list)))   # Output: False
```

Here, `x` is an integer, so `isinstance(x, (int, float))` returns `True`. However, it’s not a `str` or `list`, so the second check returns `False`.

Checking Type Info for Built-in Types

You can also retrieve and work with the type of Python's built-in objects like functions, methods, and others.

Checking the type of a function

```python
def greet():
    return "Hello"

print(type(greet))  # Output: <class 'function'>
```

Here, the type of `greet` is `<class 'function'>` because it's a function.

Checking the type of a method

```python
class MyClass:
    def my_method(self):
        pass

obj = MyClass()
print(type(obj.my_method))  # Output: <class 'method'>
```

The method `my_method` of the object `obj` has the type `<class 'method'>`.

Checking Types of Collections (Lists, Dictionaries, Tuples, Sets)

Python provides several built-in collection types. You can use `type()` or `isinstance()` to check their types.

Checking the type of a tuple

```python
tuple_example = (1, 2, 3)
print(type(tuple_example))  # Output: <class 'tuple'>
```

The variable `tuple_example` is a tuple, as shown by the output `<class 'tuple'>`.

Checking the type of a dictionary

```python
dict_example = {"name": "Alice", "age": 25}
print(type(dict_example))  # Output: <class 'dict'>
```

Here, `dict_example` is a dictionary, as indicated by the output `<class 'dict'>`.

Checking the type of a set

```python
set_example = {1, 2, 3}
print(type(set_example))  # Output: <class 'set'>
```

The variable `set_example` is a set, which is confirmed by the output `<class 'set'>`.

Checking Types with Type Annotations (Python 3.5+)

Python 3.5 and later versions allow for type annotations to specify expected types of variables and function return types.

Type annotations in a function

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

```

```python
print(type(add))  # Output: <class 'function'>
```

This example annotates the function `add` to accept two integers (`a` and `b`) and return an integer.

Using `typing` module for more complex type annotations

```python
from typing import List

def get_names(names: List[str]) -> List[str]:
    return names

names = ["Alice", "Bob", "Charlie"]
print(get_names(names))  # Output: ['Alice', 'Bob', 'Charlie']
```

In this example, we use the `List[str]` annotation to specify that the function accepts and returns a list of strings.

Summary

- `type()` is used to **get the exact type of an object**, useful for checking primitive data types, user-defined objects, or built-in functions.
- `isinstance()` is more flexible and allows you to **check whether an object is an instance of a class or one of its subclasses**.
- Python supports complex data structures like lists, dictionaries, and tuples, and you can check their types easily with `type()` or `isinstance()`.
- **Type annotations** (introduced in Python 3.5) are useful for documenting the types of variables and function signatures.

These tools help in debugging, ensuring type safety, and working with complex codebases.

⭐ Objective 6: Working with Strings

In Python, **strings** are sequences of characters enclosed in either single quotes (`'`) or double quotes (`"`). They are used to store and manipulate text-based data.

Creating Strings

You can create a string by enclosing text in quotes:

```python
text1 = "Hello"
text2 = 'World'
```

Accessing Characters in a String

Strings are **indexed** (starting from 0), so you can access individual characters using their index:

```python
greeting = "Hello"
print(greeting[0])  # Output: H
```

String Slicing

You can extract a substring by specifying a range of indices:

```python
greeting = "Hello"
print(greeting[1:4])  # Output: ell
```

Concatenation

You can combine (concatenate) strings using the `+` operator:

```python
part1 = "Hello"
part2 = "World"
result = part1 + " " + part2  # Output: Hello World
```

Repetition

You can repeat a string using the `*` operator:

```python
word = "Python"
print(word * 3)  # Output: PythonPythonPython
```

String Formatting

f-strings (available in Python 3.6+) - allows variable interpolation (inserting the values of variables in text with automatic conversion if necessary):

```python
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
# Output: My name is Alice and I am 25 years old.
```

Escaping Special Characters

Use a backslash (`\`) to escape characters like quotes or special symbols:

```python
text = "He said, \"Hello!\""  # Output: He said, "Hello!"
```

String Methods

Python provides many built-in methods to manipulate strings:

`len()` - Get Length of a String

The `len()` function returns the number of characters in a string.

Example:

```python
text = "Hello, World!"
print(len(text))  # Output: 13
```

`upper()` - Convert to Uppercase

The `upper()` method returns a new string with all characters converted to uppercase.

Example:

```python
text = "hello"
print(text.upper())  # Output: "HELLO"
```

3. `lower()` - Convert to Lowercase

The `lower()` method returns a new string with all characters converted to lowercase.

Example:

```python
text = "HELLO"
print(text.lower())  # Output: "hello"
```

`strip()` - Remove Leading and Trailing Whitespaces

The `strip()` method removes any leading (spaces at the beginning) and trailing (spaces at the end) whitespace from a string.

Example:

```python
text = "   hello   "
print(text.strip())  # Output: "hello"
```

`replace()` - Replace Substring with Another

The `replace()` method replaces all occurrences of a specified substring with another substring.

Example:

```python
text = "Hello, World!"
print(text.replace("World", "Python"))  # Output: "Hello, Python!"
```

`split()` - Split String into a List

The `split()` method splits a string into a list where each word is a list item. By default, it splits by arbitrarily many whitespace characters.

Example:

```python
text = "apple,banana,cherry"
print(text.split(","))  # Output: ['apple', 'banana', 'cherry']
```

`join()` - Join List of Strings into a Single String

The `join()` method joins a list of strings into a single string, with a specified separator.

Example:

```python
words = ['apple', 'banana', 'cherry']
separator = ", "
print(separator.join(words))  # Output: "apple, banana, cherry"
```

`find()` - Find Substring Index

The `find()` method returns the index of the first occurrence of the substring. If the substring is not found, it returns `-1`.

Example:

```python
text = "Hello, World!"
print(text.find("World"))  # Output: 7
print(text.find("Python"))  # Output: -1
```

`startswith()` - Check if String Starts with a Substring

The `startswith()` method checks if the string starts with a specific substring and returns `True` or `False`.

Example:

```python
text = "Hello, World!"
print(text.startswith("Hello"))  # Output: True
print(text.startswith("World"))  # Output: False
```

`endswith()` - Check if String Ends with a Substring

The `endswith()` method checks if the string ends with a specific substring and returns `True` or `False`.

Example:

```python
text = "Hello, World!"
print(text.endswith("World!"))  # Output: True
print(text.endswith("hello"))   # Output: False
```

`capitalize()` - Capitalize the First Character

The `capitalize()` method capitalizes the first character of the string and makes all other characters lowercase.

Example:

```python
text = "hello"
print(text.capitalize())  # Output: "Hello"
```

`count()` - Count Occurrences of a Substring

The `count()` method returns the number of occurrences of a substring in the string.

Example:

```python
text = "hello, hello, hello"
print(text.count("hello"))  # Output: 3
```

`isalnum()` - Check if All Characters Are Alphanumeric

The `isalnum()` method returns `True` if all characters in the string are alphanumeric (letters and numbers), and `False` otherwise.

Example:

```python
text = "Hello123"
print(text.isalnum())  # Output: True

text = "Hello 123"
print(text.isalnum())  # Output: False (space is not alphanumeric)
```

`isdigit()` - Check if All Characters Are Digits

The `isdigit()` method returns `True` if all characters in the string are digits, and `False` otherwise.

Example:

```python
text = "12345"
print(text.isdigit())  # Output: True

text = "123a45"
print(text.isdigit())  # Output: False
```

`title()` - Convert to Title Case

The `title()` method converts the first character of each word to uppercase and the rest to lowercase.

Example:

```python
text = "hello world"
print(text.title())  # Output: "Hello World"
```

Summary of Common String Functions/Methods:

- `len()`: Get the length of a string.
- `upper()`: Convert string to uppercase.
- `lower()`: Convert string to lowercase.
- `strip()`: Remove leading and trailing whitespaces.
- `replace()`: Replace a substring with another substring.
- `split()`: Split the string into a list of substrings.
- `join()`: Join a list of strings into a single string.
- `find()`: Find the index of a substring.
- `startswith()`: Check if the string starts with a substring.
- `endswith()`: Check if the string ends with a substring.
- `capitalize()`: Capitalize the first letter.
- `count()`: Count occurrences of a substring.
- `isalnum()`: Check if all characters are alphanumeric.
- `isdigit()`: Check if all characters are digits.
- `title()`: Convert to title case.