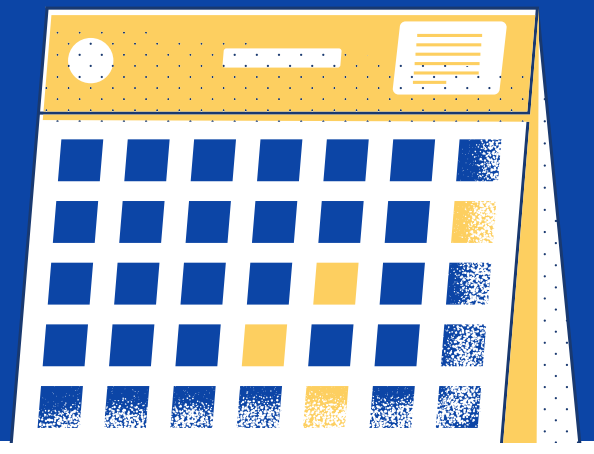
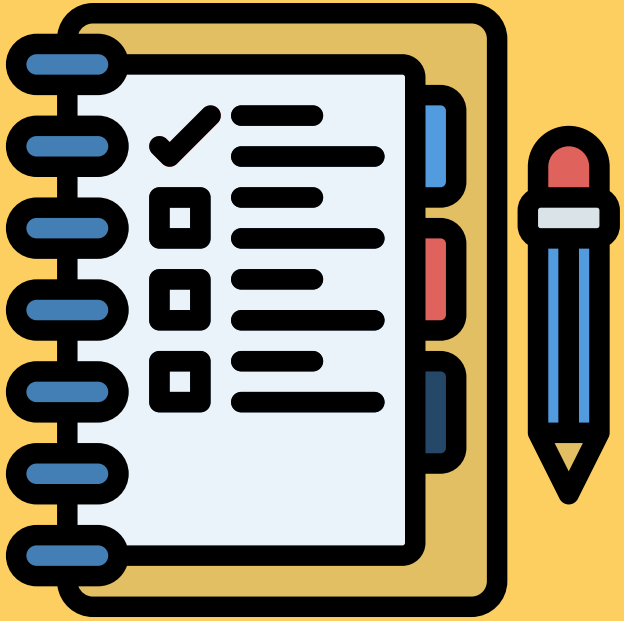
Data Fundamentals

Python: Data Types, Operators, and Data Structures

GOALS



- **Comprehension**: Understand the basic concepts of Python programming including data types, operators, built-in functions, and style guidelines.
- **Application**: Learn how to apply these concepts to create, manipulate and retrieve information from Python data structures.
- **Development**: Foster a deeper understanding of Python programming, nurturing the ability to use advanced operators and built-in functions effectively.
- **Promote Best Practices**: Learn about style guidelines in Python programming for maintaining clean and understandable code.
- **Continuous Learning**: Provide resources for further exploration and learning.

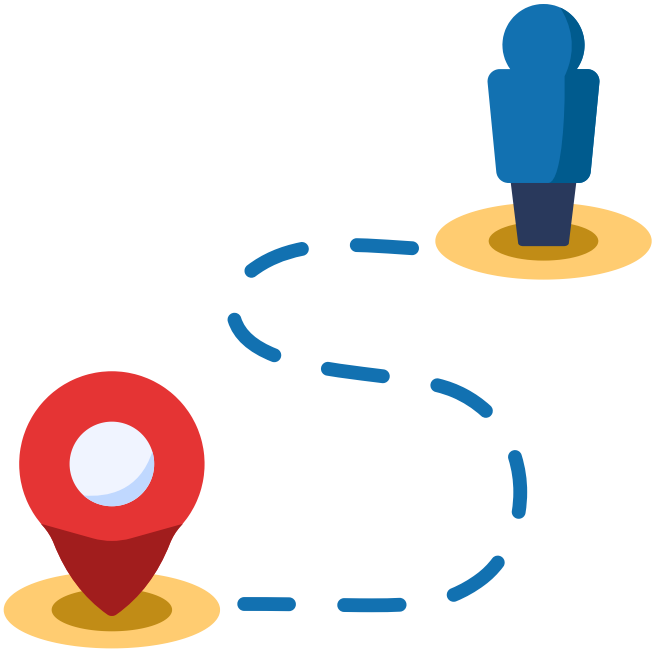


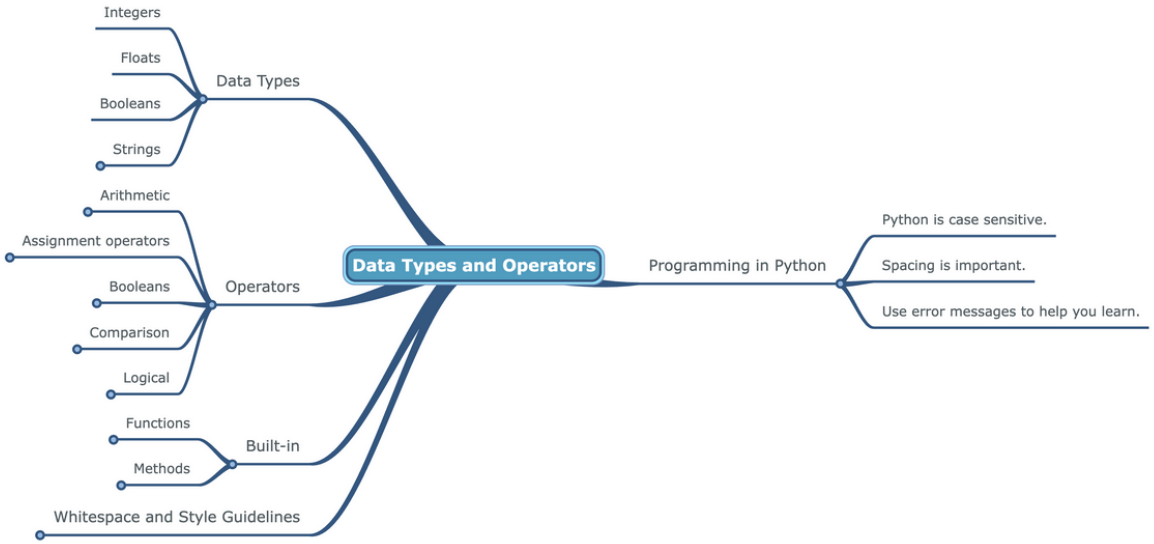
AGENDA

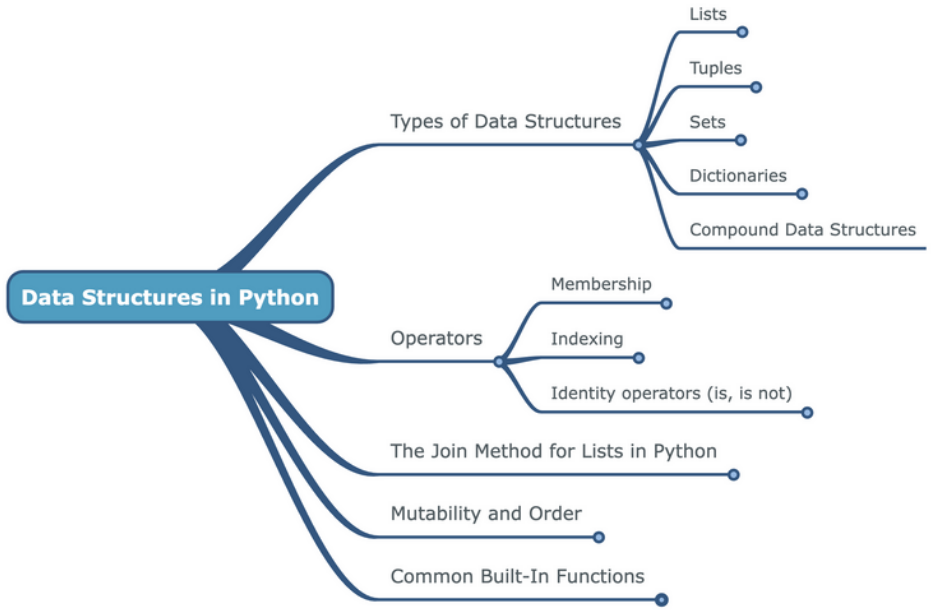
- Welcome
- Roadmap
- Introduction to Python Programming
- Python Data Types
- Python Operators
- Python Built-in Functions and Methods
- Python Style Guidelines
- Python Data Structures
- Advanced Python Operators
- Common Built-In Functions for Data Structures
- Q&A

Behind every data point, there's a story waiting to be told.

ROADMAP




Integers
Floats
Booleans
Strings
Data Types
Arithmetic
Assignment operators
Booleans
Comparison
Logical
Operators
Functions
Methods
Built-in
Whitespace and Style Guidelines
Data Types and Operators
Programming in Python
Python is case sensitive.
Spacing is important.
Use error messages to help you learn.


Lists
Tuples
Sets
Dictionaries
Compound Data Structures
Types of Data Structures
Data Structures in Python
Membership
Indexing
Identity operators (is, is not)
Operators
The Join Method for Lists in Python
Mutability and Order
Common Built-In Functions

PYTHON

Python is a popular high-level programming language known for its simplicity, readability, and versatility. It was created by Guido van Rossum and initially released in 1991.

FUNDAMENTAL PYTHON CONCEPTS

```python
def print_numbers():
    for i in range(1, 6):
        print(i)
```

```python
def print_numbers():
for i in range(1, 6):
    print(i)
```

Understanding Error Messages

```
  Cell In[30], line 11
    for i in range(1, 6):
    ^
IndentationError: expected an indented block
```

Python's Case-Sensitivity & Significance of Spaces

Built-In Functions, Data Types, Operators

```python
# Built-in Functions

print("Hello, world!") # Hello, world!
length = len("Hello, world!") # 13
print(length) # 13

numbers = [1, 2, 3, 4, 5]
print(max(numbers)) # 5

# Data Types
integer_type = 10 # Integer
float_type = 20.5 # Float
string_type = "Hello, world!" # String
list_type = [1, 2, 3, 4, 5] # List
dict_type = {"one": 1, "two": 2, "three": 3} # Dictionary

# Operators
# Arithmetic Operators
print(10 + 5) # Addition, outputs: 15
print(10 - 5) # Subtraction, outputs: 5
print(10 * 5) # Multiplication, outputs: 50
print(10 / 5) # Division, outputs: 2.0

# Comparison Operators
print(10 > 5) # Greater than, outputs: True
print(10 < 5) # Less than, outputs: False
print(10 == 5) # Equal to, outputs: False

# Logical Operators
print(10 > 5 and 5 < 2) # Logical AND, outputs: False
print(10 > 5 or 5 < 2) # Logical OR, outputs: True
print(not 10 > 5) # Logical NOT, outputs: False
```

PYTHON'S BASIC DATA TYPES

INTEGERS

These are positive or negative whole numbers without a decimal point.
For example, **123**, **-456**, **0**.

```
x = 10
print(type(x)) # <class 'int'>
```

FLOATS

These are real numbers with a decimal point. For example, **3.14**, **-0.99**, **0.0**.

```
y = 3.14
print(type(y)) # <class 'float'>
```

BOOLEANS

These represent the truth values **True** and **False**.

```
a = True
b = False
print(type(a)) # <class 'bool'>
```

STRINGS

These are sequences of character data. The string type in Python is called **str**.

```
s = "Hello, World!"
print(type(s)) # <class 'str'>
```

Single or Double Quotes

In Python, strings can be defined using either single quotes **' '** or double quotes **" "**.

```
s1 = 'Hello, World!'
s2 = "Hello, World!"
print(s1 == s2) # True

quote = "Linus Torvalds once said, 'Talk is cheap. Show me the code.'"
```

INCLUDING SPECIAL CHARACTERS IN STRINGS

In Python, certain special characters can be included in strings by using the backslash **** escape character. Here are some examples:

- **\n** stands for a newline.
- **\t** stands for a tab.
- **\"** stands for a double quote.
- **\'** stands for a single quote.
- **** stands for a single backslash.

```
print("Hello, \nWorld!") # outputs:
# Hello,
# World!
print("Hello, \tWorld!") # outputs: Hello,  World!
print("He said, \"Hello, World!\"") # outputs: He said, "Hello, World!"
print('It\'s a beautiful day!') # outputs: It's a beautiful day!
print("This is a backslash: \\") # outputs: This is a backslash: \
```

PYTHON OPERATORS

ARITHMETIC OPERATORS

These operators are used to perform basic mathematical operations.

- **+** : Addition
- **-** : Subtraction
- ***** : Multiplication
- **/** : Division
- **%** : Modulo (remainder of the division)
- ****** : Exponentiation
- **//** : Floor division (division that rounds down)

```
a = 10
b = 3

print(a + b)  # 13
print(a - b)  # 7
print(a * b)  # 30
print(a / b)  # 3.3333333333333335
print(a % b)  # 1
print(a ** b) # 1000
print(a // b) # 3
```

ASSIGNMENT OPERATORS

These operators are used to assign values to variables. In addition to basic assignment (**=**), Python supports combined assignment and arithmetic operators.

- **=** : Assigns values from right side to left side
- **+=** : Adds right side to the left side and then assigns to left side
- **-=** : Subtracts right side from the left side and then assigns to left side
- ***=** : Multiplies right side with the left side and then assigns to left side

```
a = 5
a += 3    # equivalent to a = a + 3
print(a)  # 8

a -= 2    # equivalent to a = a - 2
print(a)  # 6

a *= 2    # equivalent to a = a * 2
print(a)  # 12
```

COMPARISON OPERATORS

These operators are used to compare values. They return either True or False depending on the condition.

- **==** : Checks if the value of two operands are equal
- **!=** : Checks if the value of two operands are not equal
- **>** : Checks if the value of left operand is greater than the value of right operand
- **<** : Checks if the value of left operand is less than the value of right operand
- **>=** : Checks if the value of left operand is greater than or equal to the value of right operand
- **<=** : Checks if the value of left operand is less than or equal to the value of right operand

```
a = 10
b = 3

print(a == b)  # False
print(a != b)  # True
print(a > b)   # True
print(a < b)   # False
print(a >= b)  # True
print(a <= b)  # False
```

LOGICAL OPERATORS

These operators are used to combine conditional statements.

- **and** : Returns True if both statements are true
- **or** : Returns True if one of the statements is true
- **not** : Reverse the result, returns False if the result is true

```
a = 10
b = 3
c = 20

print(a > b and b < c) # True because both conditions are true
print(a < b or b < c)  # True because one of the conditions is true
print(not(a < b))      # True because a < b is False, and the 'not' operator reverses the result
```

PYTHON BUILT-IN FUNCTIONS AND METHODS

BUILT-IN FUNCTIONS

Built-in functions in Python are pre-defined functions provided as part of the Python standard library, which are available to use in any Python program without the need for import statements. They are designed to provide simple, commonly used functionality.

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

# len()
print(len("Hello, World!")) # 13

# type()
print(type("Hello, World!")) # <class 'str'>
```

```
# int(x)
print(int("123")) # 123

# float(x)
print(float("123.456")) # 123.456

# bool(x)
print(bool("")) # False
print(bool("some text")) # True

# str(x)
print(str(123)) # "123"
```

BUILT-IN STRING METHODS

1. **lower()**: Converts all the characters in a string to lowercase.
2. **upper():** Converts all the characters in a string to uppercase.
3. **split(**separator**)**: Splits a string into a list where each word is a list item. It uses the provided separator to determine the place to split the string. If no separator is provided, it uses a space as the default separator.
4. **find(**substring**)**: Returns the index of the first occurrence of the specified substring within the string, or -1 if the substring is not found.
5. **format()**: Formats specified values in a string. It can be used to inject variables or literals into a string with placeholders, denoted by curly **{}** brackets.

```
# String methods
s = "Hello, World!"

# lower()
print(s.lower()) # hello, world!

# upper()
print(s.upper()) # HELLO, WORLD!
```

```
# split()
print(s.split()) # ['Hello,', 'World!']

# find()
print(s.find("World")) # 7

# format()
print("Hello, {}".format("World")) # Hello, World
```

PYTHON STYLE GUIDELINES

PYTHON CODING STYLE

- Indentation: Use 4 spaces per level.
- Line Length: Maximum 79 characters.
- Whitespace: Add around operators and after commas for clarity.
- Imports: One module per line.
- Use lowercase and underscores (e.g., my_variable).
- Constants are capitalized (e.g., MY_CONSTANT).
- Avoid single-character names, except for counters.

```
def my_function():
    for i in range(5):   # 4 spaces of indentation
        print(i)         # 8 spaces of indentation


# Instead of
def function(arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9, arg10): pass
# Use
def function(arg1, arg2, arg3, arg4, arg5,
             arg6, arg7, arg8, arg9, arg10): pass

# Instead of
for x in my_list: print(x)
# Use
for item in my_list: print(item)
```

```
# Instead of
sum=a+b
# Use
sum = a + b


# Instead of
import os, sys
# Use
import os
import sys
```

PYTHON DATA STRUCTURES

LISTS

These are one of the built-in data types in Python used to store collections of data, which are **ordered** and **mutable**.

```python
# Define a list
my_list = [1, 2, 2, 3, 4, 5] #

# Identity
print("Identity of list: ", id(my_list)) # Identity of list:  140539859316864

# Mutable
my_list[0] = 'a'  # Change the first element of the list
print("List after mutation: ", my_list) # List after mutation:  ['a', 2, 2, 3, 4, 5]

# Identity after mutation
print("Identity of list: ", id(my_list)) # Identity of list:  140539859316864

# Append
my_list.append('b')
print("List after appending: ", my_list) # List after appending:  ['a', 2, 2, 3, 4, 5, 'b']

# Remove
my_list.remove(2)
print("List after removing 2: ", my_list) # List after removing 2:  ['a', 2, 3, 4, 5, 'b']

# Index
print("Index of 3 in the list: ", my_list.index(3)) # Index of 3 in the list:  2

# Count
print("Count of 2 in the list: ", my_list.count(2)) # Count of 2 in the list:  1

# Extend
my_list.extend(['c', 'd'])
print("List after extending: ", my_list) # List after extending:  ['a', 2, 3, 4, 5, 'b', 'c', 'd']
```

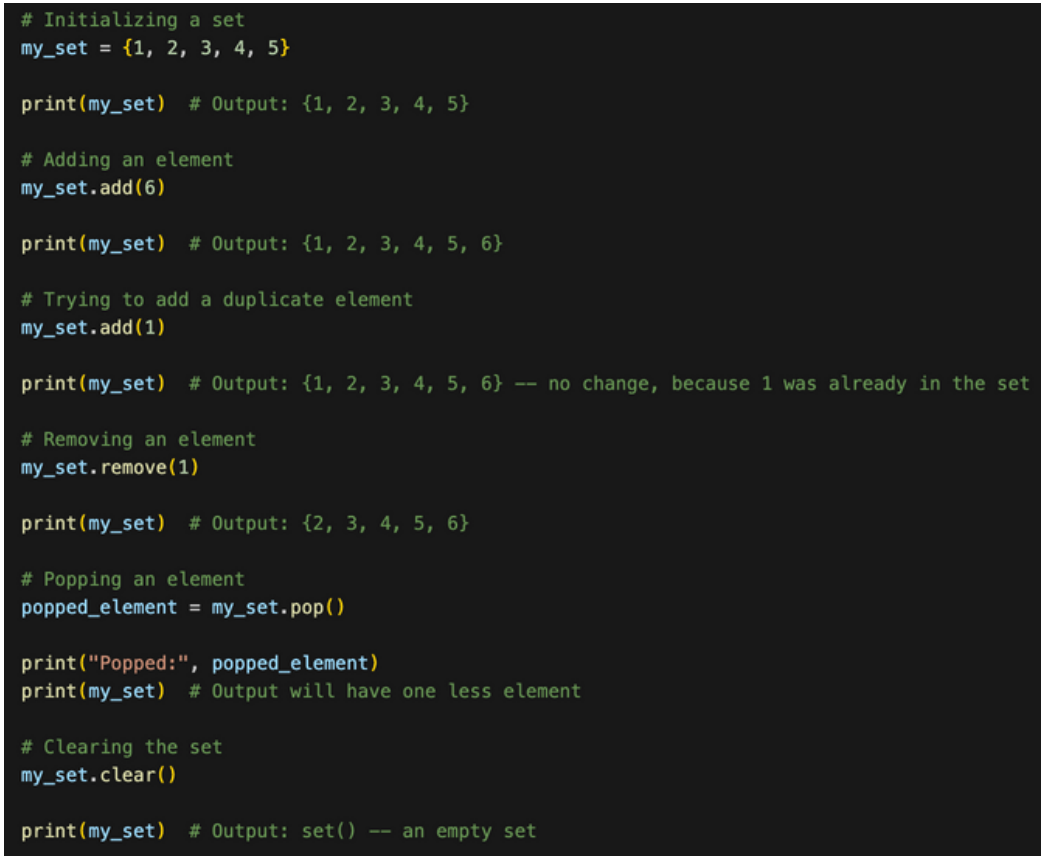
TUPLES

Tuples are ordered and immutable data structures in Python, allowing multiple items to be stored in a single variable. They guarantee data stability, which is particularly beneficial in multithreaded contexts.

```python
# Define a tuple
my_tuple = (1, 2, 2, 3, 4)
# Identity before mutation
print("Identity of tuple: ", id(my_tuple)) # Identity of tuple:  4400358800

print("Tuple before mutation: ", my_tuple) # Tuple before mutation:  (1, 2, 2, 3, 4)

my_tuple+=('a','b') # Concatenate the tuple with another tuple

# Identity after mutation
print("Identity of tuple: ", id(my_tuple)) # Identity of tuple:  4400376896

print("Tuple after mutation: ", my_tuple) # Tuple after mutation:  (1, 2, 2, 3, 4, 'a', 'b')

# Count the occurrences of 2
count_2 = my_tuple.count(2)
print(f"Number of occurrences of 2: {count_2}")

# Find the index of the first occurrence of 3
index_3 = my_tuple.index(3)
print(f"Index of the first occurrence of 3: {index_3}")
```

SETS

Sets in Python are mutable, unordered collections of unique elements, ideal for tracking elements without considering order, eliminating duplicates, and enabling fast lookup with O(1) complexity.



```python
# Initializing a set
my_set = {1, 2, 3, 4, 5}

print(my_set)  # Output: {1, 2, 3, 4, 5}

# Adding an element
my_set.add(6)

print(my_set)  # Output: {1, 2, 3, 4, 5, 6}

# Trying to add a duplicate element
my_set.add(1)

print(my_set)  # Output: {1, 2, 3, 4, 5, 6} -- no change, because 1 was already in the set

# Removing an element
my_set.remove(1)

print(my_set)  # Output: {2, 3, 4, 5, 6}

# Popping an element
popped_element = my_set.pop()

print("Popped:", popped_element)
print(my_set)  # Output will have one less element

# Clearing the set
my_set.clear()

print(my_set)  # Output: set() -- an empty set
```

DICTIONARIES

From Python 3.7 onwards, dictionaries are ordered by the insertion order of items. This means that the order in which items are added into a dictionary is preserved. Before Python 3.7, there was no guarantee of order in a dictionary.

```
# Example Dictionary
my_dict = {
    "name": "John",
    "age": 25,
    "city": "New York"
}

# Dictionaries also come with built-in methods to make it easier to work with them.

# The keys() method returns a view object containing the keys of the dictionary.
print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'city'])

# The values() method returns a view object containing the values of the dictionary.
print(my_dict.values())  # Output: dict_values(['John', 25, 'New York'])

# The items() method returns a view object containing key-value pairs of the dictionary.
print(my_dict.items())  # Output: dict_items([('name', 'John'), ('age', 25), ('city', 'New York')])

# The get() method returns the value associated with the specified key.
# If the key is not found, it returns a default value or None if not specified.
print(my_dict.get('name'))  # Output: 'John'
print(my_dict.get('country'))  # Output: None
print(my_dict.get('country', 'USA'))  # Output: 'USA'
```

ADVANCED PYTHON OPERATORS

MEMBERSHIP OPERATORS

- **in**: The in operator checks if a value exists within a sequence (such as a string, list, tuple, or set). It returns True if the value is found, and False otherwise.
- **not in**: The not in operator checks if a value does not exist within a sequence. It returns True if the value is not found, and False if it is found.

```
# Membership operators with a list
my_list = [1, 2, 3, 4, 5]
print(3 in my_list)         # Output: True
print(6 not in my_list)     # Output: True

# Membership operators with a string
my_string = "Hello, world!"
print("world" in my_string)     # Output: True
print("Python" not in my_string)     # Output: True
```

INDEXING OPERATORS

[]: Indexing operators allow you to access individual elements or slices of elements within a sequence. You can use positive or negative integers to access elements by their position. Positive integers start from 0 (the first element), while negative integers start from -1 (the last element).

```
# Indexing operators with a list
my_list = [1, 2, 3, 4, 5]
print(my_list[2])          # Output: 3
print(my_list[-1])         # Output: 5
print(my_list[1:4])        # Output: [2, 3, 4]

# Indexing operators with a string
my_string = "Hello, world!"
print(my_string[7])        # Output: 'w'
print(my_string[-6:])      # Output: 'world!'
```

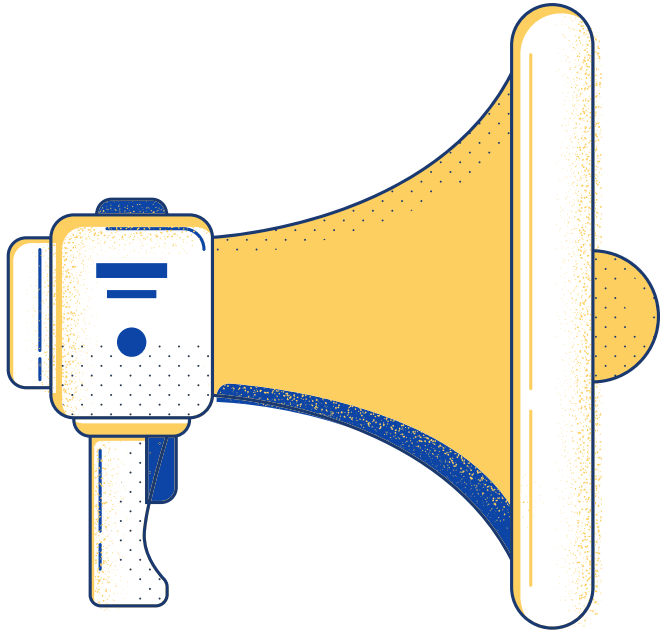
IDENTITY OPERATORS

- **is**: The **is** operator checks if two variables refer to the same object in memory. It returns **True** if the variables are the same object, and **False** otherwise.
- **is not**: The **is not** operator checks if two variables do not refer to the same object. It returns **True** if the variables are different objects, and **False** if they refer to the same object.

```
# Identity operators with integers
a = 10
b = 10
c = 20
print(a is b)            # Output: True
print(a is c)            # Output: False
print(a is not c)        # Output: True

# Identity operators with lists
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = list1
print(list1 is list2)    # Output: False
print(list1 is list3)    # Output: True
```

COMMON BUILT-IN FUNCTIONS FOR DATA STRUCTURES

```python
# List
numbers = [5, 2, 8, 1, 9]
print("Length of list:", len(numbers)) # Length of list: 5
print("Maximum element in list:", max(numbers)) # Maximum element in list: 9
print("Minimum element in list:", min(numbers)) # Minimum element in list: 1
print("Sorted list:", sorted(numbers)) # Sorted list: [1, 2, 5, 8, 9]
print()

# String
word = "Hello"
print("Length of string:", len(word)) # Length of string: 5
print("Maximum character in string:", max(word)) # Maximum character in string: o
print("Minimum character in string:", min(word)) # Minimum character in string: H
print("Sorted string:", sorted(word)) # Sorted string: ['H', 'e', 'l', 'l', 'o']
print()

# Tuple
fruits = ("apple", "banana", "cherry")
print("Length of tuple:", len(fruits)) # Length of tuple: 3
print("Maximum element in tuple:", max(fruits)) # Maximum element in tuple: cherry
print("Minimum element in tuple:", min(fruits)) # Minimum element in tuple: apple
print("Sorted tuple:", sorted(fruits)) # Sorted tuple: ['apple', 'banana', 'cherry']
print()

# Dictionary
grades = {"Alice": 90, "Bob": 85, "Charlie": 92}
print("Length of dictionary:", len(grades)) # Length of dictionary: 3
print("Maximum key in dictionary:", max(grades)) # Maximum key in dictionary: Charlie
print("Minimum key in dictionary:", min(grades)) # Minimum key in dictionary: Alice
print("Sorted keys in dictionary:", sorted(grades)) # Sorted keys in dictionary: ['Alice', 'Bob', 'Charlie']
```



Q&A Session: Let's explore and understand together

RESOURCES

- Data Types and Operators
- Data Structures in Python
- Python cheatsheet
- Python cheatsheet 2
- Google colab

Your presence today has added value to our shared learning journey. Thank you for joining us!