Programming Refresher

Python Data Types and Operators

Learning Objectives

By the end of this lesson, you will be able to:

- Identify various Python data types to handle diverse data efficiently
- Assign different data types to Python variables to manage data effectively
- Write and test Python code that demonstrates the use of arithmetic, assignment, comparison, logical, and miscellaneous operators
- Manipulate strings using Python string functions to enhance text processing capabilities

Business Scenario

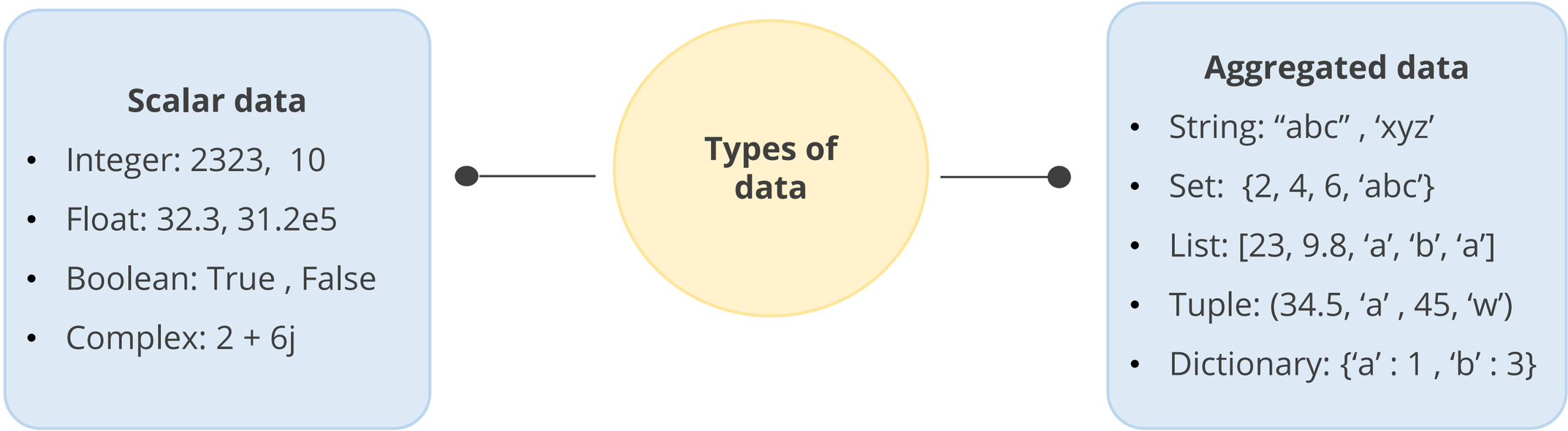
A software development company is organizing a training session on Python data types, operators, and string functions. The goal is to enhance the team's understanding of these fundamental concepts. This will improve their ability to handle diverse data, perform efficient operations, and manipulate strings effectively. The training will equip the team with essential Python programming skills.

As a result, they will achieve higher code quality, optimized performance, and increased productivity when delivering exceptional software solutions to clients.

Data Types and Data Assignment

Data Types

Variables can store various kinds of data; each has a specific function.
Objects represent data.



Note

The data type of any object can be checked by using the built-in function **type().**

Scalar Data

Scalar data types represent a single value. The following are the syntax and examples of scalar data:

```
int = 42
print(int)
```

```
42
```

```
float = 3.14
print(float)
```

```
3.14
```

```
bool = True
print(bool)
```

```
True
```

- **Integer** represents whole numbers without a fractional part.
- **Float** represents real numbers with a fractional part.
- **Boolean** represents two values, either *True* or *False*.

Scalar Data

Scalar data types represent a single value. The following are the syntax and examples of scalar data:

```
complex = 3 + 4j
real = complex.real
imaginary = complex.imag
print("Complex Number:",complex)
print("Real Part:", real)
print("Imaginary Part:", imaginary)
```

```
Complex Number: (3+4j)
Real Part: 3.0
Imaginary Part: 4.0
```

- The **complex** data type in Python represents complex numbers.
- Complex numbers are expressed as **a + bj**, where **a** is the real part, and **b** is the imaginary part.

Aggregated Data

Aggregated data types represent collections of values. The following are the syntax and examples of aggregated data:

```
string = "Hello, World!"
print(string)
```

```
Hello, World!
```

```
set = {1, 2, 3, "a", "b", "c"}
print(set)
```

```
{'a', 1, 2, 3, 'c', 'b'}
```

```
list = [1, 2, 3, "a", "b", "c"]
print(list)
```

```
[1, 2, 3, 'a', 'b', 'c']
```

- **String** represents sequences of characters enclosed in single, double, or triple quotes.
- **Set** is an unordered collection of unique items in Python.
- **List** is an ordered collection of items, which can be of different types.

Aggregated Data

Aggregated data types represent collections of values. The following are the syntax and examples of aggregated data:

```
tuple = (1, 2, 3, "a", "b", "c")
print(tuple)
```

```
(1, 2, 3, 'a', 'b', 'c')
```

```
dict = {"name": "Alice", "age": 35, "city": "New York"}
print(dict)
```

```
{'name': 'Alice', 'age': 35, 'city': 'New York'}
```

- **Tuple** is an ordered collection of items that are of different types.
- **Dictionaries** are an unordered collection of key-value pairs.

Data Assignment

Python variables are references to objects, but actual data is contained in the objects.

```
x = 34
y = x
print('x = ', x, ' ; id of x: ', id(x))
print('y = ', y, ' ; id of x: ', id(y))
```

```
x =  34  ; id of x:  140210482584912
y =  34  ; id of x:  140210482584912
```

Here, *x* and *y* point to the same memory location where 34 (an integer object) is stored.

```
y = 78
print('x = ', x, ' ; id of x: ', id(x))
print('y = ', y, ' ; id of x: ', id(y))
```

```
x =  34  ; id of x:  140210482584912
y =  78  ; id of x:  140210482774800
```

Here, *y* points to a new integer object with 78 as a value, and *x* points to the previous object with 34.

These references can be verified by using the **id()** function.

Operators in Python

Operators

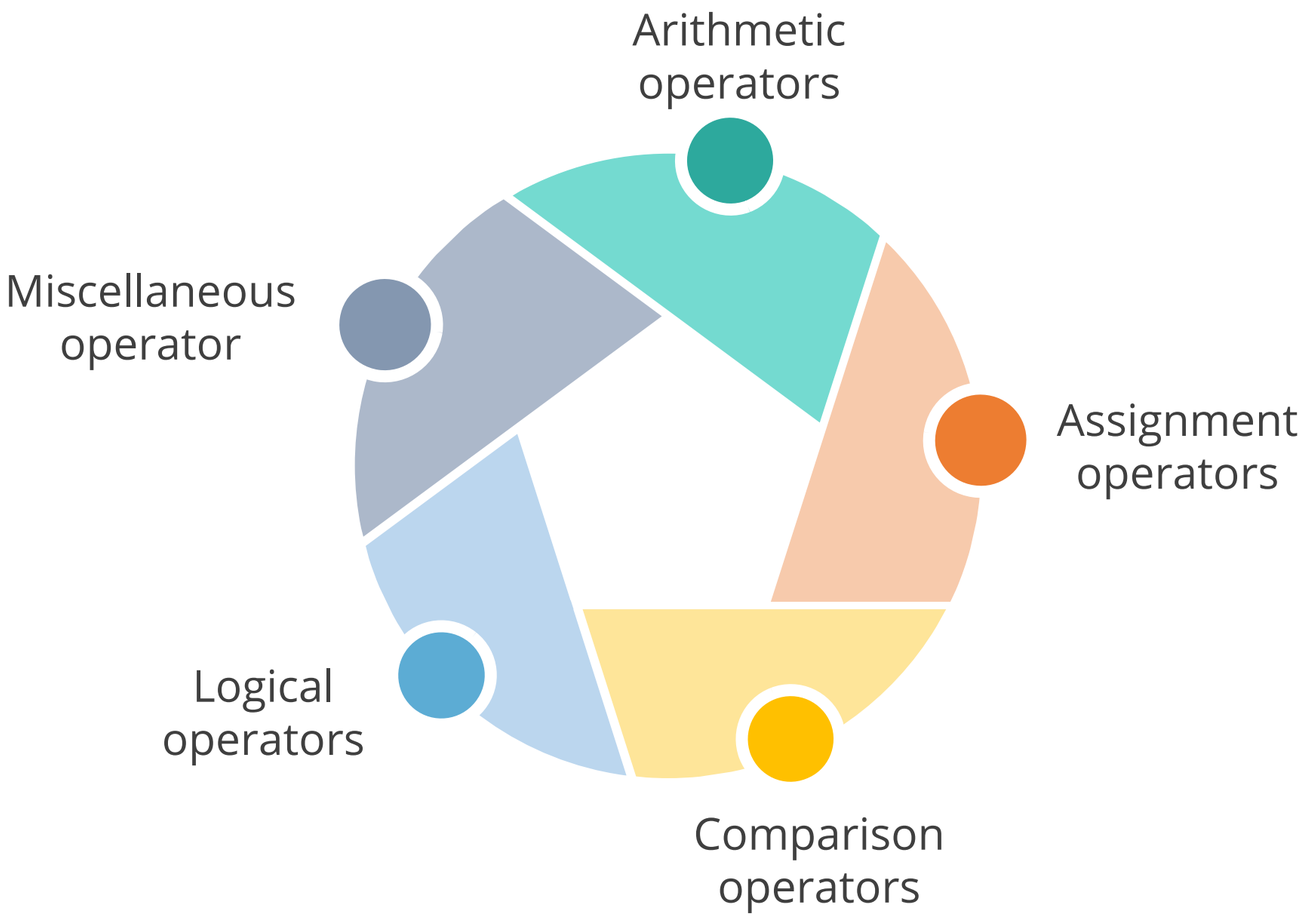
Operators in Python are special symbols or keywords that perform computations.

The following are the purposes of operators:

- Operators are symbols that tell the interpreter to perform specific mathematical, relational, or logical operations to produce a result.
- Operators operate on operands, which are the values or variables on which the operation is performed.

Operators

The different types of operators are given below:



Arithmetic Operators

Different arithmetic operators are given below with an example:

Operator	Description	Example
+	Addition	>> x = 12 ; y = 50 >> x + y 62
-	Subtraction	>> x = 45 ; y = 24 >> x - y 21
*	Multiplication	>> x = 50 ; y = 4 >> x * y 200
/	Division	>> x = 50 ; y = 4 >> x / y 12.5

Arithmetic Operators

Different arithmetic operators are given below with an example:

Operator	Description	**Example**
%	Modulo	>> x = 50 ; y = 4 >> x % y 2
//	Integer divide	>> x = 50 ; y = 4 >> x // y 12
**	Exponent	>> x = 5 ; y = 4 >> x ** y 625

Assignment Operators

The *equal to* "=" operator is used for data assignment in Python.

Example:

```
a = 10
print(a)
a += 5
print(a)
```

```
10
15
```

- Assignment operators can be combined with arithmetic operators.
- Here, a += 5 is the same as a = a + 5.
- Similar operators are: -= , *=, /=, %=, //=, and **= .

Comparison Operators

Comparison operators are used to compare two values.

Operator	Description	**Example**
==	Returns **True** when the two values are equal	>> x = 20 ; y = 20 >> x == y True
!=	Returns **True** when the two values are not equal	>> x = 45 ; y = 24 >> x != y True

Note

These operators return true or false based on the comparison.

Comparison Operators

Comparison operators are used to compare two values.

Operator	Description	**Example**
<	Returns **True** when the first value is less than the second	>> x = 20 ; y = 30 >> x < y True
>	Returns **True** when the first value is greater than the second	>> x = 40 ; y = 30 >> x > y True

Note

These operators return true or false based on the comparison.

Comparison Operators

Comparison operators are used to compare two values.

Operator	Description	**Example**
<=	Returns **True** when the first value is less than or equal to the second	>> x = 10 ; y = 30 >> x <= y True
>=	Returns **True** when the first value is greater than or equal to the second value	>> x = 30 ; y = 30 >> x >= y True

Note

These operators return true or false based on the comparison.

Logical Operators

Logical operators are used for combining conditional statements.

```
a=1
b=2
a==1 and b==2
```

```
True
```

```
a=1
b=4
a==1 or b==2
```

```
True
```

```
b=0
not b
```

```
True
```

Operator	Description
and	Returns **True** if both statements are true
or	Returns **True** if one of the statements is true
not	Reverses the results, returns **True** if the result is false

Miscellaneous Operators

Miscellaneous operators are used for specific operations compared to arithmetic, logical, and comparison operators. There are two types of miscellaneous operators:

```
a=['a','b','c']
b=['a','b','c']
a is b
```

```
False
```

```
a=['a','b','c']
b=['a','b','c']
a is not b
```

```
True
```

```
a = [1, 2, 3]
b = a
a is b
```

```
True
```

1. Identity Operators

Identity operators compare variables to see whether they are the same object at the same memory address.

- **is:** Returns **True** if both variables are the same object
- **is not:** Returns **True** if both variables are not the same object

Miscellaneous Operators

```
a=[20, 45, 10]
10 in a
```

```
True
```

```
a=[20, 45, 10]
10 not in a
```

```
False
```

```
a=[20, 45, 10]
30 not in a
```

```
True
```

2. Membership Operators

Membership operators test if a sequence is present in an object.

in: Returns **True** if a value is present in the object

not in: Returns **True** if a value is not present in the object

Strings in Python

Strings

Strings are a sequence of characters that can be either letters or alphanumeric.

Strings in Python are enclosed in either single or double quotation marks.

```
message_1 = "Hi! Welcome to Python Programming!!"
message_2 = 'Hi! Welcome to Python Programming!!'

print(message_1)
print(message_2)
```

```
Hi! Welcome to Python Programming!!
Hi! Welcome to Python Programming!!
```

In the above example, the values of message_1 and message_2 are the same.

Strings

Triple single quotes (‘’’... ’’’) or triple double quotes (""" ... """) can create multiline strings in Python.

```
message_1 = 'Hello Class!! \nHi! Welcome to Python Programming!!'
print(message_1)
```

```
Hello Class!!
Hi! Welcome to Python Programming!!
```

```
message_2 = '''Hello Class!!
Hi! Welcome to Python Programming!!'''
print(message_2)
```

```
Hello Class!!
Hi! Welcome to Python Programming!!
```

In the above example, the values of message_1 and message_2 are similar.

Accessing Characters in Strings

Strings can be accessed by using subscripts or indexes. Indexing in Python starts with 0.

The characters of the string can be accessed as:

```
string = "Hello World!"
print(string[0])
```

```
H
```

```
print(string[4])
```

```
o
```

The last character or reverse of the string can be accessed as:

```
print(string[-1])
```

```
!
```

```
print(string[-2])
```

```
d
```

Explanation

0	1	2	3	4	5	6	7	8	9	10	11
H	e	l	l	o		W	o	r	l	d	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

String Functions

Methods	Description
capitalize	Returns a string in sentence case, which means that the first letter is upper case, and the rest is lower case
lower	Returns a copy of the string converted to lowercase
upper	Returns a copy of the string converted to uppercase
strip	Returns a copy of the string with leading and trailing whitespace removed
join	Concatenates any number of strings
split	Divides a string into a list of substrings based on a specified delimiter

Strings are immutable Python objects.
All string methods return a new value and do not change the original string.

Assisted Practice : Data Types, Operators, and Strings

Objective: In this demonstration, we will learn more about data types, operators, and strings in Python.

Tasks to perform:

1. Introduce data types

2. Use operators in Python

3. Understand strings

Note

Please refer to the **Reference Material** section to download the **Jupyter Notebook** files for the mentioned topic.

ASSISTED PRACTICE

File Handling in Python

File Handling in Python

File operations are fundamental actions performed on files, including creating, reading, writing, and closing.

These operations are crucial for data manipulation, storage, and transfer in programming, and Python provides built-in functions and methods for handling these tasks efficiently.

Creating and Opening a File

To open a file in Python, use the **open()** function. If the file does not exist, opening it in write (w) or append (a) mode can create it.

Syntax:

```
file = open('example.txt', 'w')
```

Reading from a File

To read the entire content of a file, use the **read()** method:

Syntax:

```
content = file.read()
print(content)
```

To read one line at a time, use the **readline()** method, and for reading all lines into a list, use **readlines()**:

Syntax:

```
line = file.readline()
print(line)
lines = file.readlines()
print(lines)
```

Writing to a File and Closing a File

Use the **write()** method to write a string to a file, and **writelines()** for a list of strings:

Syntax:

```
file.write('Hello, world!\n')
lines = ['First line\n', 'Second line\n']
file.writelines(lines)
```

Close a file after the operations are complete to free up system resources.

Syntax:

```
file.close()
```

Assisted Practice : File Handling in Python

Objective: In this demonstration, we will learn more about file handling in Python.

Tasks to perform:

1. Read a file
2. Write to a file
3. Append to a file
4. Read after writing
5. Read line by line
6. Write strings and close the file

Note

Please refer to the **Reference Material** section to download the **Jupyter Notebook** and the **Dataset** files for the mentioned topic.

ASSISTED PRACTICE

Error Handling in Python

Introduction to Error Handling

It is the process of managing errors or exceptions that occur during the execution of a program providing an appropriate message to the developer.

The program encounters unexpected situations like a missing file, invalid user input, or division by zero, and an error gets raised.

The program stops abruptly when these errors occur without error handling.

The program allows for managing errors gracefully, recovering from them, or displaying user-friendly messages, ensuring it runs more smoothly and reliably.

Error Handling in Python: Example

Python provides a way to manage errors using the below blocks:

```
try:
    # Code that might raise an exception
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError:
    # Code that runs if the file is not found
    print("File not found.")
finally:
    # Code that will always run
    file.close()
```

- **try**: This block contains code that might raise an exception (e.g., attempting to open a file that might not exist).
- **except**: If the try block raises a specific exception (e.g., FileNotFoundError), this block will execute, handling the error gracefully.
- **finally**: This block always runs, regardless of whether an exception occurred or not; it is used for clean-up actions, like closing files or releasing resources.

Assisted Practice : Error Handling in Python

Objective: In this demonstration, we will explore how error handling works in Python by using try, except, and finally blocks to manage file operations.

Tasks to perform:

1. Prompt the user for a file name
2. Attempt to open and read the file contents
3. Catch the FileNotFoundError if the file does not exist
4. Display the file contents if the file is found
5. Ensure that the file is always closed, whether or not an error occurred

Note

Please refer to the **Reference Material** section to download the **Jupyter Notebook** and the **Dataset** files for the mentioned topic.

ASSISTED PRACTICE

Key Takeaways

- In Python, data is represented as objects.
- Python supports different types of operators, such as arithmetic, comparison, logical, and miscellaneous operators.
- Strings are immutable objects in Python.
- Strings can be accessed by using subscripts or indexes.
- All string methods return a new value and do not change the original string.

Knowledge Check

Knowledge Check 1

_____ operators test if a sequence is present in an object.

A. Membership

B. Logical

C. Comparison

D. All of the above

Knowledge Check 1

_____ operators test if a sequence is present in an object.

A. Membership

B. Logical

C. Comparison

D. All of the above

The correct answer is **A**

Membership operators test if a sequence is presented in an object.

Knowledge Check 2

String indexing in Python starts with _____.

A. 2

B. 1

C. 0

D. All of the above

Knowledge Check

2

String indexing in Python starts with _____.

A. 2

B. 1

C. 0

D. All of the above

The correct answer is **C**

String indexing in Python starts with 0.

Thank You!