5 Variables

5.1 Introduction to Variables:

Variables are used in Python to manage and store data. Variables are named locations that are used to store references to the object stored in memory. They serve as names for values that are symbolic, enabling developers to work with data in a more meaningful and readable manner. Variables are containers for storing data values. They are created by assigning a value to a name. Every variable needs to have a unique name or identifier. The interpreter allocates memory and determines what can be stored in the reserved memory based on the data type of a variable. Thus, you can store characters, decimals, or integers in variables by designating different data types for them.

5.1.1 Creating Variables:

Variables are created by assigning a value to a name.

```
1 age = 25
```

The equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable, and the operand to the right of the = operator is the value stored in the variable. Variable names should be descriptive and follow certain conventions (e.g., avoiding spaces and starting with a letter).

```
1 my_variable = 10
```

5.2 Multiple *Assignment*

Python allows you to assign a single value to several variables simultaneously. For example:

```
1 a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example:

```
1 a,b,c = 1,2,"john"
```

This is a concise way to initialize multiple variables. Here, two integer objects with values 1 and 2 are assigned to variables a and b, respectively, and one string (strings are text data) object with the value "john" is assigned to the variable c.

5.3 Identifiers

The names we choose for variables and functions are commonly known as Identifiers. In other words, A Python identifier is a name used to identify a variable, function, class, module, or other object. To keep it simple, see the following example again:

```
1 a,b,c = 1,2,"john"
```

Here, a,b, and are the identifiers storing their corresponding values. Similarly, other objects like functions, class, and module are also defined with name that we call as identifier. For example, function “print()”, the identifier is the work print, that contains codes which will result in printing out the content given as input inside the brackets. We will see more examples in late chapters.

5.3.1 Variable Naming Conventions:

Always follow proper naming conventions for variables given in python guidelines (e.g., lowercase with underscores for readability), and choose meaningful and descriptive variable names to enhance code readability. The following rules are followed for Identifiers:

- It can contain letters, digits, and underscores (_)
- Python does not allow punctuation characters such as @, $, and % within identifiers.
- It must start with a letter A to Z or a to z or underscore (_), followed by zero or more letters, underscores, and digits (0 to 9)
- Digits at the start is not allowed. For e.g: var is accepted, but 1var is not a valid identifier.
- Python is a case-sensitive programming language. Thus, Var and var are two different identifiers in Python.
- There is no limit to the Identifier's length.
- Identifiers can't be a keyword (described in next section).

Also, it is case-sensitive language, which means a is different from A when used as an Identifier in the following example.

```
1 A = 3
2 a = 2
3 print(A)
4 print(a)
```

5.4 Reserved Words

Keywords are reserved words that Python uses for special purposes. The following are keywords in Python

```
False |  class | finally | is   | return | None   | continue | for  | lambda
 try  | True   | del     | from | non    | local  | while    |  and |  del
global|  not   | with    | as   | elif   |  if    | or       |yield | pass
else  | import |assert   |break | except |  in    | raise
```

6 Data Types and Structure:

Now, let's try to understand the main types of data that we can use in Python. Each value stored under a variable in Python has a data type. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. A text, like "Hello world", is called a string. Whole numbers are called integers, and numbers with a decimal point are called floats, e.g. 0.6 and -4.986764.

Python has following main standard data types; we can always check the data type by using the type() function.

- Numeric Types: integer (int), decimal (float), complex
- Text Type, called as a string (str)
- Sequence Types: list, tuple, range
- Mapping Type: dictionary (dict)
- Set Types: set, frozenset
- Boolean Type (bool)

6.1.1 Integers:

Integers are whole numbers without decimal points. For example,

```
count = 10
age = 25
```

Arithmetic operations on integers follow standard rules.

6.1.1.1.1 Floats:

Floats represent numbers with decimal points. For example,

```
temperature = 98.6
height = 1.75
```

Floats are used for more precise numeric calculations.

6.1.1.1.2 Strings:

Strings are sequences of characters enclosed in single or double quotes. In simple words, strings refer to text data. Python allows for either a pair of single or double quotes.

```
name = 'Alice'
name = "John"
```

Strings are versatile and can represent text, names, or any sequence of characters.

6.1.2 Setting the Specific Data Type

We can also specify the data type at the time of assignment, with the use of constructor functions. See the examples below:

```
count = int(10)
age = int(25)
temperature = float(98.6)
height = float(1.75)
name = str('Alice')
name = str('John')
```

6.1.3 Type Conversion:

Changing the type of a variable is known as type conversion.

```
x = 10
y = str(x) # Convert integer to string
```

Common type conversion functions include `int()`, `float()`, and `str()`.

6.2 Lists and Tuples:

6.2.1 Lists

Lists are ordered, mutable collections in Python, represented by square brackets.

```
my_list = [1, 2, 3, 'apple', 'banana']
```

The list refers to a collection of data which are normally related. Instead of storing these data as separate variables, we can store them as a list. For instance, suppose our program needs to store the age of five users. Instead of storing them as different variables, it makes more sense to store them as a list. To declare a list, you write

```
listName = [values separated by comma]
```

Note that we use square brackets [ ] when declaring a list. A comma separates multiple values. For example:

```
userAge = [21, 22, 23, 24, 25]
```

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets '[]'. To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data types. Elements of a list can be accessed, added, removed, or modified using various methods. We can also declare a list without assigning any initial values to it. We simply write

```
listName = []
```

What we have now is an empty list with no items in it. We have to use the *append()* method mentioned later to add items to the list.

6.2.2 Tuples

Tuples are ordered, immutable collections represented by parentheses. For example,

```
my_tuple = (1, 2, 3, 'apple', 'banana')
```

Tuples are useful for data that should not be modified after creation. A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses. The main differences between lists and tuples are: Lists are enclosed in brackets '[ ]' and their elements and size can be changed, while tuples are enclosed in parentheses '( )' and cannot be updated. Tuples can be thought of as read-only lists.

6.3 Dictionaries

Dictionaries are unordered collections of key-value pairs, represented by curly braces. For instance, if we want to store the username and age of five users, we can store them in a dictionary. A dictionary key can be almost any Python type, but it is usually a number or a

string, and they must be unique (within one dictionary). Values, on the other hand, can be any arbitrary Python object.

To declare a dictionary, you write

```
dictionaryName = {dictionary key : data}
```

Dictionaries are enclosed by curly braces '{ }' and values can be assigned and accessed using square braces '[]'. For example,

```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'Wonderland'}
```

The values can be accessed using keys, and they can be added, modified, or removed. Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are unordered.

Since, dictionary keys has to be unique, you should not declare a dictionary like this

myDictionary = {"Peter":38, "John":51, "Peter":13}.

This is because "Peter" is used as the dictionary key twice.

Instead of using curly braces method, the dict() function can also be used to declare a dictionary. For example,

```
userNameAndAge = dict(Peter = 38, John = 51, Alex = 13, Alvin = "Not Available")
```

Note: When you use this method to declare a dictionary, you use parentheses ( ) instead of braces { } and you do not put quotation marks for the dictionary keys.

6.4 Sets:

Sets are unordered collections of unique elements, represented by curly braces. For example,

```
my_set = {1, 2, 3, 'apple', 'banana'}
```

Sets are similar to lists or dictionaries. They are created using curly braces and are unordered, which means that they can't be indexed. Sets are generally useful for operations like union, intersection, and difference. Due to the way they're stored, it's faster to check whether an item is part of a set using the in operator rather than part of a list.

Note: Sets cannot contain duplicate elements.

6.5 Summary of Data Structures

As we have seen in the previous sections, Python supports the following collection types: Lists, Dictionaries, Tuples, and Sets. Now, when to use which type, some of the cases are given below:

- Use lists if you have a collection of data that does not need random access. Try to choose lists when you need a simple, iterable collection that is modified frequently.
- Use a set if you need uniqueness for the elements.
- Use tuples when your data cannot/should not change.
- Use dictionary: i) when you need a logical association between a key: value pair, ii) when you need fast lookup for your data, based on a custom key, or iii) when your data is being constantly modified. Remember, dictionaries are mutable.

7 Common functions/operations associated with different Data types

7.1 String Functions and formatting

We cannot combine strings and numbers like this:

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

But we can combine strings and numbers by using different string format methods as discussed below.

7.1.1 Formatting Strings using F-string method

F-strings, short for formatted string literals, are a way of embedding expressions inside string literals, using curly braces `{}`. They were introduced in Python 3.6 and provide a more readable and concise way to format strings.

Key Features of F-Strings:
1. Prefix with `f` or `F`: To create an f-string, you start the string with an `f` or `F` before the opening quote.

```
name = "Alice"
greeting = f"Hello, {name}!"
print(greeting)
```

```
Output:
Hello, Alice!
```

2. Embed Expressions: You can directly embed expressions within `{}` inside the string.

```
x = 5
y = 10
result = f"{x} multiplied by {y} equals {x * y}."
print(result)
```

```
Output:
5 multiplied by 10 equals 50.
```

3. Supports Various Data Types: F-strings can include integers, floats, strings, lists, dictionaries, and even function.

```
pi = 3.14159
info = f"The value of pi is approximately {pi}."
print(info)
```

```
Output:
The value of pi is approximately 3.14159.
```

4. String Formatting: F-strings support formatting options, such as specifying the number of decimal places, aligning text, and adding commas to numbers. This is discussed separately.

```
value = 1234567.89
formatted = f"Formatted value: {value:,.2f}"
print(formatted)
```

```
Output:
Formatted value: 1,234,567.89
```

5. Multi-line Support: F-strings can be used across multiple lines for readability.

```
name = "Alice"
age = 30
bio = (
  f"Name: {name}\n"
  f"Age: {age}\n"
  f"Occupation: Developer"
  )
print(bio)
```

```
Output:
Name: Alice
Age: 30
Occupation: Developer
```

6. Efficient: F-strings are generally faster and more efficient compared to older string formatting methods like `%` formatting or `str.format()`.

Example of F-Strings:

```
name = "Bob"
age = 25
height = 175.5
info = f"My name is {name}, I am {age} years old, and I am {height:.1f}
cm tall."
print(info)
```

```
Output:
My name is Bob, I am 25 years old, and I am 175.5 cm tall.
```

In this example:
- `{name}`, `{age}`, and `{height:.1f}` are placeholders where the values of `name`, `age`, and `height` are inserted.
- `{height:.1f}` formats the height to one decimal place.

F-strings make code cleaner and more readable, especially when you need to insert multiple variables or expressions into strings. In Python, before the introduction of f-strings, there were other methods for formatting strings.

7.1.2 Formatting Strings using the % formatting Operator

This is the oldest method of formatting strings, similar to the print f-style formatting in C. You use the `%` operator to insert values into a string.

Basic Syntax:

```
"string" % value
```

The syntax for using the % operator is "string to be formatted" %(values or variables to be inserted into string, separated by commas). There are three parts to this syntax. First, we write the string to be formatted in quotes. Next, we write the % symbol. Finally, we have a pair of parentheses ( ) within which we write the values or variables to be inserted into the string.

- Placeholders:
  - `%s`: String or any object with a string representation.
  - `%d`: Integer.
  - `%f`: Floating-point number.
  - `%x`: Hexadecimal.

Examples:

```
name = "Alice"
age = 30

# Inserting a string
greeting = "Hello, %s!" % (name)
print(greeting)
```

```
Output:
Hello, Alice!
```

```
# Inserting multiple values
info = "My name is %s and I am %d years old." % (name, age)
print(info)
```

```
Output:
My name is Alice and I am 30 years old.
```

```
# Formatting a float with specific precision
pi = 3.14159
formatted pi = "Pi is approximately %.2f." % pi
print(formatted_pi)
```

```
Output:
Pi is approximately 3.14.
```

7.1.3 Formatting Strings using .format() method

This method was introduced in Python 2.7 and 3.0, the `str.format()` method is more powerful and flexible than `%` formatting. This method uses curly braces `{}` as placeholders in the string, which are replaced by the values passed to `format()`. The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are.

Basic Syntax:

```
"string {}".format(value)
```

Placeholders:
- `{}`: Positional or keyword arguments.
- `{index}`: Refers to the position of the argument in the `format()` method.
- `{keyword}`: Refers to the named keyword arguments.
- `{field_name:format_spec}`: Allows for more advanced formatting.

Examples:

```
name = "Alice"
age = 30
# Inserting a string
greeting = "Hello, {}!".format(name)
print(greeting)
```

```
Output:
Hello, Alice!
```

```
# Inserting multiple values
info = "My name is {} and I am {} years old.".format(name, age)
print(info)
```

```
Output:
My name is Alice and I am 30 years old.
```

```
# Using positional arguments
info = "My name is {0} and I am {1} years old.".format(name, age)
print(info)
```

```
Output:
My name is Alice and I am 30 years old.
```

```
# Using keyword arguments
info = "My name is {name} and I am {age} years
old.".format(name="Alice", age=30)
print(info)
```

```
Output:
My name is Alice and I am 30 years old.
```

```
# Formatting a float with specific precision
pi = 3.14159
formatted_pi = "Pi is approximately {:.2f}".format(pi)
print(formatted_pi)
```

```
Output:
Pi is approximately 3.14.
```

7.1.4 Comparison with F-Strings

Following are the points when F-strings are compared with other two methods:

- Readability: F-strings are more concise and readable, especially when embedding expressions.
- Performance: F-strings are generally faster because they are evaluated at runtime, whereas `str.format()` and `%` formatting involve more overhead.
- Complexity: `str.format()` allows for more complex formatting and is useful when formatting needs go beyond basic insertion of variables.

Example of All Three Methods:

```
name = "Alice"
age = 30
pi = 3.14159
# Using % formatting
old_style = "My name is %s, I am %d years old, and Pi is %.2f." %
(name, age, pi)
# Using str.format()
new_style = "My name is {}, I am {} years old, and Pi is
{:.2f}.".format(name, age, pi)
# Using f-strings (only in Python 3.6+)
f_string_style = f"My name is {name}, I am {age} years old, and Pi is
{pi:.2f}."

print(old_style)
print(new_style)
print(f_string_style)
```

```
Output:
My name is Alice, I am 30 years old, and Pi is 3.14.
My name is Alice, I am 30 years old, and Pi is 3.14.
My name is Alice, I am 30 years old, and Pi is 3.14.
```

All three methods achieve the same result, but f-strings are often preferred for their simplicity and efficiency in modern Python code.

7.1.5 Format specifiers

F-strings in Python offer a variety of format specifiers that allow you to control how variables are represented within a string. These format specifiers can be used to format numbers, strings, dates, and other data types.

Following are the Common F-String Format Specifiers used on string formatting.

1. General Format:

- `{value}`: The default format, which simply converts the value to a string.

```
name = "Alice"
f"Hello, {name}!"
```

```
Output:
'Hello, Alice!'
```

2. Number Formatting:

- Decimal Places:
  - `{value:.2f}`: Formats a floating-point number to 2 decimal places.

```
pi = 3.14159
f"Pi is approximately {pi:.2f}."
```

```
Output:
'Pi is approximately 3.14.'
```

- Width and Precision:
  - `{value:10.2f}`: Formats a floating-point number with a total width of 10 characters, including 2 decimal places.

```
num = 123.456
f"Number: {num:10.2f}"
```

```
Output:
'Number:     123.46'
```

- Thousands Separator:
  - `{value:,}`: Adds a comma as a thousands separator.

```
large_number = 1000000
f"Large number: {large_number:,}"
```

```
Output:
'Large number: 1,000,000'
```

- Percentage:
  - `{value:.2%}`: Converts a decimal to a percentage, with 2 decimal places.

```
percentage = 0.85
f"Success rate: {percentage:.2%}"
```

```
Output:
'Success rate: 85.00%'
```

3. String Formatting:
- Align Text:
  - `{value:<10}`: Left-align the text within a field of 10 characters.
  - `{value:>10}`: Right-align the text within a field of 10 characters.
  - `{value:^10}`: Center-align the text within a field of 10 characters.

```
text = "Hi"
f"Left: [{text:<10}], Right: [{text:>10}], Center: [{text:^10}]"
```

```
Output: 'Left: [Hi        ], Right: [        Hi], Center: [    Hi    ]'
```

- Truncate Strings:
  - `{value:.5}`: Truncate the string to the first 5 characters.

```
text = "Hello, World!"
```

```
f"Truncated: {text:.5}"
```

```
Output:
'Truncated: Hello'
```

Combining Format Specifiers:

You can combine multiple format specifiers in a single placeholder by placing them in the `{}` braces, separated by colons.

```
number = 1234.56789
f"Formatted number: {number:,.2f}"
```

```
Output:
'Formatted number: 1,234.57'
```

In this example:
- `,` adds a thousands separator.
- `.2f` formats the number to 2 decimal places.

7.2 Accessing elements of Data structures

Following methods can be used to access elements of data structures.

7.2.1 Slicing/Indexing

Slicing is a concept to carve out a substring from a given string. Subsets of strings can be accessed, extracted or taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end. For example,

```
str = "Hello, World!"
print(str)  # This will print the whole string
print(str[0]) # This will print the first character of the string
print(str[2:5]) # This will print character from 3rd to 5th.
```

```
Output:
Hello, World!
H
llo
```

So, you can return a range of characters by using the slice syntax. Specify the start index and the end index, separated by a colon, to return a part of the string.

Note: The end index is excluded

```
print(str[2:])
```

This will print string starting from 3rd character as shown below:

```
llo, World!
```

By leaving out the end index, the range will go to the end.

```
print(str[:5])
```

This will slice from start to 5th character:

```
Hello
```

By leaving out the start index, the range will start at the first character.

```
print(str[-5:-2])
```

```
Output:
orl
```

Use negative indexes to start the slice from the end of the string. It will print string starting from 5rd character from the right end to 3rd character from the right end

Note: comma and space are also counted as characters.

7.2.2 Double colon in slicing

The double colon is a special case in Python's extended slicing feature. This extended slicing notation, written as

string[start:stop:step]

uses three arguments, start, stop, and step, to extract a subsequence. It accesses every step-th element between indices start (included) and stop (excluded). For example, the expression s[2:4] from string 'hello' carves out the slice 'll' and the expression s[:3:2] carves out the slice 'hl'. Here are some more examples.

```
s = 'hello world'
print(s[::2])
```

This is default start index, default stop index, step size is two—prints every second element:

```
Output:
hlowrd
```

```
print(s[::3])
```

This is default start index, default stop index, step size is three—prints every third element

```
Output:
hlwl
```

```
print(s[2::2])
```

This reads "start index of two, default stop index, step size is two—prints every second element starting from index 2".

```
Output:
lowrd
```

Note: All three arguments are optional, so you can skip them to use the default values (start=0, stop=len(lst), step=1). This will show as double colon :: where you drop all the arguments. This will actually print the whole string as it is.

7.3 Built-in functions for Strings

The following common functions of Python will be shown in tabular format for ease of understanding.

Method/Function name and Description	Code Examples	Output
capitalize() Converts the first character to upper case	`txt = "hello, and welcome to my world."` `x = txt.capitalize()` `print (x)`	`Hello, and welcome to my world.`
swapcase() Swaps cases, lower case becomes upper case and vice versa	`txt = "Hello My Name Is PETER"` `x = txt.swapcase()` `print(x)`	`hELLO mY nAME iS peter`
casefold() Converts string into lower case	`txt = "Hello, And Welcome To My World!"` `x = txt.casefold()` `print(x)`	`hello, and welcome to my world!`
title() Converts the first character of each word to upper case	`txt = "Welcome to my world"` `x = txt.title()` `print(x)`	`Welcome To My World`
istitle() Returns True if the string follows the rules of a title	`print('This Is A String'.istitle())` `print('This Is a String'.istitle())`	`True` `False`

Method/Function name and Description	Code Examples	Output
upper() Converts a string into upper case	`a = "Hello, World!"` `print(a.upper())`	`HELLO, WORLD!`
isupper() Returns True if all characters in the string are upper case	`print(a.isupper())`	`True`
lower() Converts a string into lower case	`print(a.lower())` `print('Hello Python'.lower())`	`hello, world!` `hello python`
islower() Returns True if all characters in the string are lower case	`print('abcd'.islower())` `print('Abcd'.islower())` `print('ABCD'.islower())`	`True` `False` `False`
count() Returns the number of times a specified value occurs in a string count(sub,[start,[end]]) returns the number of times the substring sub appears in the string	`# In the example below, 's' occurs at index 3,6, and 10` `# Count the entire string` `print('This is a string'.count('s'))` `# Count from index 4 to end of string` `print('This is a string'.count('s',4))` `# Count from index 4 to 10-1` `print('This is a string'.count('s',4,10))` `# Count 'T'. There's only 1 'T' as the function is case sensitive.` `print('This is a string'.count('T'))`	`3` `2` `1` `1`
find() Searches the string for a specified value and returns the position of where it was found, returns -1 if sub is not found. Its only available for strings	`# Check the entire string` `print('This is a string'.find('s'))` `print('This is a string'.find('m'))` `# Check the index 4 to end of string` `print('This is a string'.find('s',4))` `# Check the index 7 to 11-1` `print('This is a string'.find('s',7,11))` `# Sub is no found` `print('This is a string'.find('p'))`	`3` `-1` `6` `10` `-1`
rfind() Searches the string for a specified value and returns the last position of where it was found	`txt = "Mi casa, su casa."` `x = txt.rfind("casa")` `print(x)`	`12`

Method/Function name and Description	Code Examples	Output
index() Searches the string for a specified value and returns the position of where it was found, returns ValueError if sub is not found. Its available for lists, tuples and strings	`# Check the entire string` `print('This is a string'.index('s'))` `# Check the index 4 to end of string` `print('This is a string'.index('s',4))` `# Check the index 7 to 11-1` `print('This is a string'.index('s',7,11))` `# Sub is no found (This will throw error if executed)` `print('This is a string'.index('p'))` `print('This is a string'.index('m'))`	3 6 10
rindex() Searches the string for a specified value and returns the last position of where it was found	`txt = "Mi casa, su casa."` `x = txt.rindex("casa")` `print(x)`	12
isalnum() Returns True if all characters in the string are alphanumeric	`print('abcd1234'.isalnum())` `print('a b c d 1 2 3 4'.isalnum())` `print('abcd'.isalnum())` `print('1234'.isalnum())`	True False True True
isalpha() Returns True if all characters in the string are in the alphabet	`print('abcd'.isalpha())` `print('abcd1234'.isalpha())` `print('1234'.isalpha())` `print('a b c'.isalpha())`	True False False False
isdigit() Returns True if all characters in the string are digits	`print('1234'.isdigit())` `print('abcd1234'.isdigit())` `print('abcd'.isdigit())` `print('1 2 3 4'.isdigit())`	True False False False
isnumeric() Returns True if all characters in the string are numeric	`txt = "565543"` `x = txt.isnumeric()` `print(x)`	True
isidentifier() Returns True if the string is an identifier	`txt = "Demo"` `x = txt.isidentifier()` `print(x)`	True
isspace() Returns True if all characters in the string are whitespaces	`print(''.isspace())` `print(' '.isspace())` `print('a b'.isspace())`	False True False

Method/Function name and Description	Code Examples	Output
replace() replace(old,new[,count]) This function will return a copy of the string with all the occurrences of substring old replaced by new #count is optional. If given, only the first count occurrences are replaced. This function is case-sensitive.	`a = "Hello, World!"print(a.replace("H", "J"))` `# Replace all occurrence` `print('This is a string'.replace('s','p'))` `# Replace first 2 occurrences` `print('This is a string'.replace('s','p',2))`	`Jello, World!` `Thip ip a ptring` `Thip ip a string`
split() The .split() function in Python is a string method used to divide a string into a list of substrings based on a specified delimiter. By default, the delimiter is any whitespace (space, newline, tab, etc.), but you can specify any string as the delimiter. string.split(separator, maxsplit) **separator** (optional): The delimiter at which the string is split. If not specified, the default is any whitespace. **maxsplit** (optional): The maximum number of splits to do. The default value is -1, which means "all occurrences". *Notes: If the separator is not found in the string, the result is a list containing the original string.*	`# Using default whitespace as the delimiter` `text = "This is a sample string"` `words = text.split()` `print(words)` `# Specifying a delimiter` `data = "apple,banana,cherry"` `fruits = data.split(',')` `print(fruits)` `# Using maxsplit` `text = "one two three four"` `parts = text.split(' ', 2)` `print(parts)`	`['This', 'is', 'a', 'sample', 'string']` `['apple', 'banana', 'cherry']` `['one', 'two', 'three four']`

Method/Function name and Description	Code Examples	Output
strip() Returns a trimmed version of the string *string.strip([chars])* **chars** (optional): A string specifying the set of characters to be removed. If omitted, all leading and trailing whitespace characters are removed. The *.strip()* function in Python is a string method used to remove leading and trailing whitespace characters (spaces, tabs, newlines, etc.) from a string. You can also specify a set of characters to remove instead of just whitespace. *Notes:* - *.strip()* removes characters from both the beginning and the end of the string. If you only want to remove characters from the beginning, use *.lstrip()*, and if only from the end, use *.rstrip()*. - The order of characters in the *chars* parameter does not matter; it removes any occurrence of those characters at the beginning or end of the string.	`# Removing leading and trailing whitespace` `text = "   Hello, World!   "` `stripped_text = text.strip()` `print(f"'{stripped_text}'")` `# Remove  'g' from the end` `print('This is a string'.strip('g'))` `# Removing specific characters` `text = "///Hello, World!///"` `stripped_text = text.strip('/')` `print(f"'{stripped_text}'")` `' This is a string '.strip('s') # Nothing is removed since 's' is not at the start or end of the string` `# Removing a combination of characters` `text = "abcHello, World!cba"` `stripped text = text.strip('abc')` `print(f"'{stripped_text}'")` `# Using .lstrip() to remove leading characters` `text = "   Hello, World!   "` `left_stripped_text = text.lstrip()` `print(f"'{left_stripped_text}'")` `# Using .rstrip() to remove trailing characters` `text = "   Hello, World!   "` `right_stripped_text = text.rstrip()` `print(f"'{right_stripped_text}'")`	`'Hello, World!'` `This is a strin` `'Hello, World!'` `'Hello, World!'` `'Hello, World! '` `  Hello, World!'`
splitlines() This function returns a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends argument is given as True The .splitlines() method in Python splits a string into a list of lines. It recognizes different newline characters	`txt = "Thank you for the music\nWelcome to the jungle"` `lines = txt.splitlines()  # Split the string into lines` `print(lines)` `# If you want to keep the line breaks in the resulting list, use` `lines_with_breaks = txt.splitlines(True)` `print(lines_with_breaks)`	`['Thank you for the music', 'Welcome to the jungle']` `['Thank you for the music\n', 'Welcome`

Method/Function name and Description	Code Examples	Output
such as \n, \r, or \r\n and splits the string at those points. By default, it does not include the line breaks in the resulting list. However, you can choose to include them by passing 'True' as an argument.		to the jungle']
startswith(prefix,[start[,end]]) Returns True if the string starts with the specified value; otherwise, returns False. The prefix can also be a tuple of prefixes to look for. This function is case-sensitive.	#'Post' occurs at index 0 to 3 #check the entire string print('Postman'.startswith('Post')) #check from index 3 to end of string print('Postman'.startswith('Post',3)) #check from index 2 to 6-1 print('Postman'.startswith('Post',2,6)) #check from index 2 to 6-1 print('Postman'.startswith('stm',7)) #Using a tuple of preffixes (check from index 3 to end of string) print('Postman'.startswith(('Post','tma'),3))	True False False False True
endswith(suffix,[start,[end]]) Returns True if the string ends with the specified suffix, otherwise returns False. Suffix can also be a tuple of suffixes to look for. This function is case sensitive.	#'man' occurs at index 4 to 6 #check the entire string print('Postman'.endswith('man')) #check from index 3 to end of string print('Postman'.endswith('man',3)) #check from index 2 to 6-1 print('Postman'.endswith('man',2,6)) #check from index 2 to 7-1 print('Postman'.endswith('man',7)) #Using a tuple of suffixed (check from index 2 to 6-1) print('Postman'.endswith(('man','ma'),2,6))	True True False False True
String concatenation To concatenate, or combine, two strings you can use the + operator.	a = "Hello" b = "World" c = a + b print(c) # To add a space between them, add a " ": a = "Hello"	HelloWorld

Method/Function name and Description	Code Examples	Output
	b = "World" c = a + " " + b print(c)	Hello World
maketrans() Returns a translation table to be used in translations The maketrans() method returns a mapping table that can be used with the translate() method to replace specified characters. str.maketrans(x, y, z) Only x given: Required. If only one parameter is specified, this has to be a dictionary describing how to perform the replace. If two or more parameters are specified, this parameter has to be a string specifying the characters you want to replace. y (optional): Optional. A string with the same length as parameter x. Each character in the first parameter will be replaced with the corresponding character in this string. z (optional): Optional. A string describing which characters to remove from the original string.	# Creating a method to replace any "S" characters with a "P" character: txt = "Hello Sam!" mytable = str.maketrans("S", "P") print(txt.translate(mytable)) txt = "Hi Sam!" x = "mSa" y = "eJo" mytable = str.maketrans(x, y) print(txt.translate(mytable))	Hello Pam! Hi Joe!
translate() Returns a translated string	txt = "Good night Sam!" x = "mSa" y = "eJo" z = "odnght" mytable = str.maketrans(x, y, z) print(txt.translate(mytable))	G i Joe!

Method/Function name and Description	Code Examples	Output
zfill() Fills the string with a specified number of 0 values at the beginning. adds zeros (0) at the beginning of the string, until it reaches the specified length. If the value of the len parameter is less than the length of the string, no filling is done.	`a = "hello"` `b = "welcome to the jungle"` `c = "10.000"` `print(a.zfill(10))` `print(b.zfill(10))` `print(c.zfill(10))`	`00000hel` `lo` `welcome` `to the` `jungle` `000010.0` `00`

7.4 Common Operations on Lists, Tuples, Dictionaries, and Sets

- **Accessing Elements:**
  - Use indexing to access elements in lists and tuples.
  - Use keys to access values in dictionaries.
- **Adding Elements:**
  - Lists: `my_list.append(4)`
  - Tuples: Tuples are immutable; new tuples must be created.
  - Dictionaries: `my_dict['gender'] = 'female'`
  - Sets: `my_set.add('orange')`
- **Removing Elements:**
  - Lists: `my_list.remove('apple')`
  - Tuples: Tuples are immutable; elements cannot be removed.
  - Dictionaries: `del my_dict['age']`
  - Sets: `my_set.remove('apple')`
- **Iterating Through Elements:**
  - Lists and Tuples: Use `for` loops.
  - Dictionaries: Iterate through keys, values, or items.
  - Sets: Iterate through elements.

7.4.1 Built-in List and Tuple functions

Method/Function name and Description	Example Codes	Output
join() The will returns a string in which the arguments provided are joined by a separator, sep='-'. This function work on strings, list as well as tuple.	myTuple=('a','b','c') myList=['d','e','f'] myString="Hello World" # The following code will perform the join function. It join with a "-" joined tuple = "-".join(myTuple) joined_list = "-".join(myList) joined_string = "-".join(myString) print(joined tuple) print(joined_list) print(joined_string)	a-b-c d-e-f H-e-l-l-o- -W-o-r-l-d
find() and index() find/index(sub,[start,[end]]) It will return the index in the string where the first occurrence of the substring 'sub' is found. Both functions behave the same way if a sub-string is found. find() returns -1 if sub is not found. index() returns ValueError if sub is not found. The find() is only available for strings where, whereas index() is available for lists, tuples, and strings. This function is case-sensitive	# The following code will check the entire string print('This is a string'.find('s')) print('This is a string'.index('s')) print('This is a string'.find('m')) # print('This is a string'.index('m')). This code will throw a value error as 'm' is not present in the string. # The following code will check the index 4 to end of string print('This is a string'.find('s',4)) # The following code will check the index 7 to 11-1 print('This is a string'.find('s',7,11)) # The following code will tell us what will happen if the 'sub' is not found print('This is a string'.find('p')) # print('This is a string'.index('p')) somelist2 = "Ok let's try this" type(somelist2) print(somelist2.index("try"))	3 3 -1 6 10 -1 9 9

Method/Function name and Description	Example Codes	Output
	print(somelist2.find("try")) print(somelist2.find("t")) print(somelist2.index("t")) # Codes with list and tuples. somelist = ['Ok', "let's", 'try', 'this', 'out'] type(somelist) print(somelist.index("try")) sometuple = ('Ok', "let's", 'try', 'this', 'out') type(sometuple) print(sometuple.index("try"))	5 5 2 2
append( ) The function will add the item to the end of a list	myList = ['a', 'b', 'c', 'd'] print (myList) myList.append('e') print (myList) myList.append("How are you") print(myList)	['a', 'b', 'c', 'd'] ['a', 'b', 'c', 'd', 'e'] ['a', 'b', 'c', 'd', 'e', 'How are you']
extend() The function will extend a list with another list.	list1 = ['a','b','c','d'] list2 = ['f', 'g','h'] print(list1) print(list2) list1.extend(list2) print(list1) print(list2) # this is list unchanged	['a', 'b', 'c', 'd'] ['f', 'g', 'h'] ['a', 'b', 'c', 'd', 'f', 'g', 'h'] ['f', 'g', 'h']
insert() The insert function will add an element at the mentioned position.	myList = ['a', 'b', 'c', 'd', 'e'] print(myList) myList.insert(1, 'Hi') print (myList)	['a', 'b', 'c', 'd', 'e'] ['a', 'Hi', 'b', 'c', 'd', 'e']
reverse() The reverse function will reverse the order of elements.	myList = [1, 2, 3, 4] print (myList) myList.reverse() print (myList)	[1, 2, 3, 4] [4, 3, 2, 1]
The in operator The in operator check if an item is in a list	# Check if an item is in a List myList = ['a', 'b', 'c', 'd'] print('c' in myList) print('e' in myList) # Check if an item is in a tuple	True False

Method/Function name and Description	Example Codes	Output
	`myTuple = ('a', 'b', 'c', 'd')` `print('c' in myTuple)` `print('e' in myTuple)`	`True` `False`
len() Find the number of items in a list, string, or tuple.	`myList = ['a', 'b', 'c', 'd']` `print (len(myList))` `myTuple = ('a', 'b', 'c', 'd')` `print (len(myTuple))`	`4` `4`
sort() and sorted() The sort() will sort a list alphabetically or numerically. The sorted() will return a new sorted list without sorting the original list. It requires a list as the argument.	`myList1 = [3, 0, -1, 4, 6]` `print(myList1)` `myList1.sort()` `print(myList1)` `myList2 = [3, 0, -1, 4, 6]` `myList3 = sorted(myList) # Original list is not sorted` `print(myList2)` `print(myList3) # New sorted list`	`[3, 0, -1, 4, 6]` `[-1, 0, 3, 4, 6]` `[3, 0, -1, 4, 6]` `[-1, 0, 3, 4, 6]`
Addition Operator: + It will concatenate list, tuple or string.	`myList = ['a', 'b', 'c', 'd']` `print(myList + ['e', 'f'])` `print (myList) # original list has not changed` `myTuple = ('a', 'b', 'c', 'd')` `print (myTuple + ('e', 'f'))` `print (myTuple) # original tuple has not changed`	`['a', 'b', 'c', 'd', 'e', 'f']` `['a', 'b', 'c', 'd']` `('a', 'b', 'c', 'd', 'e', 'f')` `('a', 'b', 'c', 'd')`
Multiplication Operator: * It will duplicate a list and concatenate it to the end of the list.	`myList = ['a', 'b', 'c', 'd']` `print (myList*3)` `print (myList)` `myTuple = ('a', 'b', 'c', 'd')` `print(myTuple*3)` `print (myTuple)` `# The + and * symbols do not modify the tuple.` `# The tuple stays as ['a', 'b', 'c', 'd'] in both cases.`	`['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd']` `['a', 'b', 'c', 'd']` `('a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd')` `('a', 'b', 'c', 'd'`

7.4.2 The del(), pop(), and remove() functions.

These three functions require separate mention, as they can be confusing because of similar-looking names and roles. However, they do have differences. They are used to take elements out, but each works in a slightly different way.

Function	**del()**	**pop()**	**remove()**
Description	Remove items from a list. Requires the index of the item as an argument.	Get the value of an item and remove it from the list. It requires an index of the item as the argument	Remove an item from a list. Requires the value of the item as the argument. Removes the first matching value, not a specific index. It requires searching the list, and raises ValueError if no such value occurs in the list.
Examples	`# To remove the sixth item from myList and print the updated list` `myList = [1, 20, 3, 4, 5, 6, 'How are you']` `del myList[5]` `print(myList)` `myList=['a','b','c','d','e','f']` `# Delete the third item (index =2)` `del myList[2]` `print(myList)` `# Delete items from index 1 to 5` `del myList[1:5]` `print(myList)` `# Delete items from index 0 to 3` `del myList[:3]` `print(myList)`	`myList=['a','b','c','d','e','f']` `item = myList.pop(1)` `print(item)` `print(myList)` `myList = ['a', 'b', 'c', 'd', 'e']` `# Remove the third item` `member = myList.pop(2)` `print (member)` `print (myList)` `# Remove the last item` `member = myList.pop( )` `print (member)` `print (myList)` `c=[1,2,3]` `c.pop(1) #removes the item at a specific index and returns it` `print(c.pop(1))`	`myList = ['a', 'b', 'c', 'd', 'e']` `# Remove the item 'c'` `myList.remove('c')` `print(myList)` `a=[1,2,3]` `a.remove(2)` `print(a)`

Function	del()	pop()	remove()
	# Delete items from index 2 to end del myList[2:] print(myList) b=[1,2,3] del b[1] # removes the item at a specific index print(b) # Delete the entire tuple myTuple = ('a', 'b', 'c', 'd') del myTuple # print (myTuple) This is will throw NameError as the tuple has been deleted	print(c)	
Output	[1, 20, 3, 4, 5, 'How are you'] ['a', 'b', 'd', 'e', 'f'] ['a'] [] [] [1, 3]	b ['a', 'c', 'd', 'e', 'f'] c ['a', 'b', 'd', 'e'] e ['a', 'b', 'd'] 3 [1]	['a', 'b', 'd', 'e'] [1, 3]

7.4.3 Accessing/modifying Dictionary items

To access individual items in the dictionary, we use the dictionary key, which is the first value in the {dictionary key : data} pair. For instance,

For a dictionary,

```
userNameAndAge = {"Peter":38, "John":51, "Alex":13, "Alvin":"Not Available"}
# To get John's age, you write
userNameAndAge["John"]
```

You'll get the value 51.

To modify items in a dictionary, we write dictionaryName[dictionary key of item to be modified] = new data. For instance, to modify the "John":51 pair, we write

```
userNameAndAge["John"] = 21
print(userNameAndAge)
```

Our dictionary now becomes

```
{'Peter': 38, 'John': 21, 'Alex': 13, 'Alvin': 'Not Available'}
```

For instance, if we want to add "Joe":40 to our dictionary, we write

```
userNameAndAge["Joe"] = 40.
print(userNameAndAge)
```

Our dictionary now becomes

```
userNameAndAge = {"Peter":38, "John":21, "Alex":13, "Alvin":"Not Available",
"Joe":40}
```

We can also declare a dictionary without assigning any initial values to it. We simply write dictionaryName = { }. What we have now is an empty dictionary with no items in it. To add items to a dictionary, we write

```
dictionaryName[dictionary key] = data.
```

To remove items from a dictionary, we write del dictionaryName[dictionary key]. For instance, to remove the "Alex":13 pair, we write

```
del userNameAndAge["Alex"]
```

Our dictionary now becomes

```
userNameAndAge = {"Peter":38, "John":21, "Alvin":"Not Available", "Joe":40}
```

When declaring the dictionary, dictionary keys and data can be of different data types

```
myDict = {"One":1.35, 2.5:"Two Point Five", 3:"+", 7.9:2}
```

To print the entire dictionary, use this:

```
print(myDict)
```

You'll get the following output:

```
{'One': 1.35, 2.5: 'Two Point Five', 3: '+', 7.9: 2}
```

Items may be displayed in a different order. Items in a dictionary are not necessarily stored in the same order as the way you declared them.

Now, print the item with key = "One".

```
print(myDict["One"])
```

You'll get 1.35

If you print the item with key = 7.9.

```
print(myDict[7.9])
```

You'll get 2

Modify the item with key = 2.5 and print the updated dictionary

```
myDict[2.5] = "Two and a Half"
print(myDict)
```

You'll get

```
{'One': 1.35, 2.5: 'Two and a Half', 3: '+', 7.9: 2}
```

Add a new item and print the updated dictionary

```
myDict["New item"] = "I'm new"
print(myDict)
```

You'll get

```
{'One': 1.35, 2.5: 'Two and a Half', 3: '+', 7.9: 2, 'New item': 'I'm new'}
```

Remove the item with key = "One" and print the updated dictionary

```
del myDict["One"]
print(myDict)
```

You'll get

```
{2.5: 'Two and a Half', 3: '+', 7.9: 2, 'New item': 'I'm new'}
```

7.4.4 In-built Dictionary methods/functions

Method/Function name and Description	**Example Codes**	**Output**
clear( ) Removes all elements of the dictionary, returning an empty dictionary.	`dic1 = {1: 'one', 2: 'two'}` `print (dic1)` `dic1.clear()` `print (dic1)`	`{1: 'one', 2: 'two'}` `{}`
del Deletes the entire dictionary.	`dic1 = {1: 'one', 2: 'two'}` `print(dic1)` `del dic1`	`{1: 'one', 2: 'two'}`
get() Returns a value for the given key. If the key is not found, it'll return the keyword None. Alternatively, you can state the value to return if the key is not found.	`dic1 = {1: 'one', 2: 'two'}` `print(dic1.get(1))` `print(dic1.get(5))` `print(dic1.get(5, "Not Found"))`	`one` `None` `Not Found`
The 'in' operator It will check if an item is in a dictionary.	`dic1 = {1: 'one', 2: 'two'}` `# Checking based on the key` `print(1 in dic1)` `print(3 in dic1)` `# Checking based on the value` `print('one' in dic1.values())` `print('three' in dic1.values())`	`True` `False` `True` `False`
items( ) Returns a list of dictionary's pairs as tuples.	`dic1 = {1: 'one', 2: 'two'}` `dic1.items()`	`dict_items([(1, 'one'), (2, 'two')])`
keys( )	`dic1 = {1: 'one', 2: 'two'}` `dic1.keys()`	`dict_keys([1, 2])`

Method/Function name and Description	Example Codes	Output
Returns list of the dictionary's keys.		
len( ) Find the number of items in a dictionary.	`dic1 = {1: 'one', 2: 'two'}` `print (len(dic1))`	`2`
update( ) Adds one dictionary's key-values pairs to another. Duplicates are removed.	`dic1 = {1: 'one', 2: 'two'}` `dic2 = {1: 'one', 3: 'three'}` `dic1.update(dic2)` `print (dic1)` `print (dic2) # no change`	`{1: 'one', 2: 'two', 3: 'three'}` `{1: 'one', 3: 'three'}`
values( ) Returns list of the dictionary's values	`dic1 = {1: 'one', 2: 'two'}` `dic1.values()`	`dict_values(['one', 'two'])`

7.4.5 In-built Set methods/functions

Method/Function Name	Description	Examples
add() and remove() You can use the add() function to add new items to the set, and remove() to delete a specific element.	`nums = {1,2,1,3,1,4,5,6,}` `# In Sets, duplicate elements will automatically get removed` `print(nums)` `nums.add(-7)` `print(nums)` `nums.remove(3)` `print(nums)`	`{1, 2, 3, 4, 5, 6}` `{1, 2, 3, 4, 5, 6, -7}` `{1, 2, 4, 5, 6, -7}`
len() The len() function can be used to return the number of elements of a set.	`nums = {"a", "b", "c", "d"}` `print(len(nums))`	`4`
Mathematical operations Sets can be combined using mathematical	`first = {1,2,3,4,5,6}` `second = {4,5,6,7,8,9}` `print(first|second)` `print(first & second)` `print(first - second)`	`{1, 2, 3, 4, 5, 6, 7, 8, 9}` `{4, 5, 6}` `{1, 2, 3}` `{8, 9, 7}` `{1, 2, 3, 7, 8, 9}`

Method/Function Name	Description	Examples
operations. The union operator | combines two sets to form a new one containing items in either. The intersection operator & gets items only in both. The difference operator - gets items in the first set but not in the second. The symmetric difference operator ^ gets items in either set, but not both.	`print(second - first)` `print(first^second)`	
The 'in' operator To check the presence of an element in a set	`num_set = {1,2,3,4,5}` `print(3 in num_set)`	`True`

7.5 List Comprehensions

List comprehensions are a useful way of quickly creating lists whose contents obey a rule. For example,

```
cubes = [i**3 for i in range(5)]
print(cubes)
nums = [i*2 for i in range(10)]
print(nums)
```

```
Output:
[0, 1, 8, 27, 64]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Create a list of multiples of 3 from 0 to 20

```
a = [i for i in range(20) if i%3 == 0]
print(a)
```

```
Output:
[0, 3, 6, 9, 12, 15, 18]
```

A list comprehension can also contain an if statement to enforce a condition on values in the list.

```
evens = [i**2 for i in range(10) if i**2 % 2 ==0]
print(evens)
```

```
Output:
[0, 4, 16, 36, 64]
```

7.6 Choosing the Right Data Structure:

- **Lists vs. Tuples:**
  - Use lists when you need a mutable collection; use tuples for immutable data.
- **Dictionaries vs. Sets:**
  - Use dictionaries for key-value pairs; use sets for unique, unordered elements.