Built-In Data Types

Unit 3

Introduction

Data types in Python refer to classifying or categorizing data objects based on their characteristics and behavior.

They define the type of values variables can hold and determine the performed operations on those values.

Python has several built-in data types, including numeric types (int, float, complex), string (str), boolean (bool), and collection types (list, tuple, dict, set).

Each data type has its own set of properties, methods, and behaviors that allow programmers to manipulate and process data effectively in their programs.

Built-in Data Types in Python

Built-in data types in Python are fundamental data structures provided by the Python programming language.

They are pre-defined and available for use without requiring any additional libraries or modules.

Python offers several built-in data types, including:

1. Numeric Data Types: Numeric data types in Python are used to represent numerical values. Python provides three primary numeric data types:
    a. Integer (int): Integers are whole numbers without any decimal points. They can be positive or negative.
    b. Floating-Point (float): Floating-point numbers represent decimal values. They can be positive or negative and may contain a decimal point.
    c. Complex (complex): Complex numbers are used to represent numbers with a real and imaginary part. They are written in the form of a + bj, where a is the real part and b is the imaginary part.

2. String Data Type(str): Represents a sequence of characters enclosed in single quotes (‘ ‘) or double quotes (” “), such as “Hello, World!”, ‘Python’.
3. Boolean Data Type(bool): Represents either True or False, used for logical operations and conditions.
4. Collection Data Types:
    a. list: Represents an ordered and mutable collection of items, enclosed in square brackets ([]).
    b. tuple: Represents an ordered and immutable collection of items, enclosed in parentheses ().
    c. dict: Represents a collection of key-value pairs enclosed in curly braces ({}) with unique keys.
    d. set: Represents an unordered and mutable collection of unique elements, enclosed in curly braces ({}) or using the set() function.

Numeric Data Types

Numeric data types in Python are used to represent numerical values.

Python provides three primary numeric data types –

integer (int),

floating-Point (float) and

complex (complex).

These numeric data types allow for performing various arithmetic operations, such as addition, subtraction, multiplication, and division. They provide the necessary flexibility to work with numerical data in Python programs.

Integers (int)

Python’s integer data type (int) represents whole numbers without any decimal points.

It stores positive and negative whole numbers.

Integers are immutable, meaning their value cannot be changed once assigned.

the range of integers (int) is only limited by the available memory resources of the system.

Python supports arbitrary-precision arithmetic for integers, which means that integers can be arbitrarily large or small, as long as there is enough memory to store them.

Python's integers are not limited to a fixed size like other programming languages. Instead, they can grow or shrink in size dynamically based on the magnitude of the integer.

This means that you can perform calculations with very large or very small integers without worrying about overflow or underflow errors.

Example

```
x = 5
y = -10
sum = x + y
difference = x - y
multiplication = x * y
division = x / y
floor_division= x//y
print("Sum:", sum)
print("Difference:", difference)
print("Multiplication:", multiplication)
print("Division:", division)
print("Floor Division:", floor_divisiont)
```

Output

Sum: -5

Difference: 15

Multiplication: -50

Division: -0.5

Floor Division: -1

What will be the output?

```
a = 10

b = 5

result = a / b

print("Division:", result)
```

Output

Division: 2.0

What will be the output?

```
a = 10

b = 5

result = a / /b

print("Floor Division:", result)
```

Output

Floor Division: 2

What will be the output?

```
a = 11

b = 5

result = a / b

print("Division:", result)
```

Output

Division: 2.2

What will be the output?

```
a = 11

b = 5

result = a / /b

print("Floor Division:", result)
```

Output

Floor Division: 2

What will be the output?

```
a = 0

b = 5

result = a / b

print(" Division:", result)
```

Output

Division: 0.0

What will be the output?

```
a = 0

b = 5

result = a / /b

print("Floor Division:", result)
```

Output

Floor Division: 0

What will be the output?

```
a = 0

b = 0

result = a / b

print("Division:", result)
```

Output

Exception

ZeroDivisionError: division by zero

Floating-Point Numbers (float)

The float data type in Python is used to represent floating-point numbers, which are numbers with decimal points.

Floats are used when more precision is needed than what integers can provide.

Floats are immutable like integers and follow the IEEE 754 standard for representing real numbers.

The value of a floating-point number is stored as a sign bit, a fraction (also known as the mantissa), and an exponent.

For example, the number 3.14 might be represented as +0.314 x 10^1, where 0.314 is the fraction and 1 is the exponent.

The range and precision of floating-point numbers in Python depend on the underlying hardware and Python implementation (e.g., CPython, PyPy).

The sys.float_info object in Python's sys module provides information about the floating-point representation on the current platform.

You can access attributes such as max, min, and epsilon to determine the range and precision of floats.

```
import sys

print("Maximum float:", sys.float_info.max)

print("Minimum float:", sys.float_info.min)

print("Smallest positive float epsilon:", sys.float_info.epsilon)
```

Example

```
a = 3.5

b = 2.0

result = a + b

print("Addition:", result)   Output: 5.5
```

Output

Addition: 5.5

Example

```
a = 3.5

b = 2.0

result = a / b

print("Division:", result)  # Output: 1.75
```

Output

Division: 1.75

Example

```
a = 3.5

b = 2.0

result = a // b

print("Floor Division:", result)  # Output: 1.0
```

Output

Floor Division: 1.0

Example

```
a = 3.5

b = 2.0

result = a % b

print("Modulus:", result)  # Output: 1.5
```

Output

Modulus: 1.5

Complex Numbers (complex)

The complex data type in Python is used to represent numbers with both real and imaginary parts. It is written in the form of a + bj, where a represents the real part and b represents the imaginary part.

Complex numbers are useful in mathematical calculations and scientific computations that involve imaginary quantities.

how to define complex numbers in Python

```
# Define a complex number

z1 = 3 + 4j

print("Complex number:", z1)  # Output: (3+4j)


# You can also use the complex() function

z2 = complex(2, -5)

print("Complex number:", z2)  # Output: (2-5j)
```

In the above examples:

z1 is a complex number with a real part of 3 and an imaginary part of 4.

z2 is a complex number with a real part of 2 and an imaginary part of -5.

Example

```
x = 2 + 3j
y = -1 + 2j
sum = x + y
difference = x - y
multiplication = x * y
division = x / y
print("Sum:", sum)
print("Difference:", difference)
print("Multiplication:", multiplication)
print("Division:", division)
```

Output

Sum: (1+5j)

Difference: (3+1j)

Multiplication: (-8+1j)

Division: (0.8-1.4j)

In Python, multiplication and division of complex numbers follow the rules of complex arithmetic. Here's how they work:

- Multiplication:
  - When you multiply two complex numbers $a+bi$ c+di, the result is calculated as:
  - $a+bi)\times(c+di)=(ac-bd)+(ad+bc)i$
  - In other words, you multiply the real parts and subtract the product of the imaginary parts for the real part of the result, and you add the product of the real and imaginary parts and the product of the imaginary parts and real parts for the imaginary part of the result.
  - This is analogous to multiplying two binomials.

- Division:
  - When you divide one complex number a+bi by another c+di, the result is calculated using the formula:

$$\frac{a+bi}{c+di} = \frac{(a+bi)\times(c-di)}{c^2+d^2}$$

Here, you multiply the numerator and denominator by the conjugate of the denominator, which is c−di.

After expanding and simplifying, you get the real and imaginary parts of the result.

Example

```
z1 = 3 + 4j

z2 = 2 - 1j

# Multiplication

product = z1 * z2

print("Multiplication:", product)  # Output: (10+5j)

# Division

quotient = z1 / z2

print("Division:", quotient)  # Output: (1.6+2.2j)
```

String

A String is a data structure in Python Programming that represents a sequence of characters.

It is an immutable data type, meaning that once you have created a string, you cannot change it.

Python String are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

Remember: Python Programming does not have a character data type, a single character is simply a string with a length of 1.

Create a String in Python

In Python, a string is a sequence of characters enclosed within either single quotes (') or double quotes (").

So, we can create strings using single quotes ('), double quotes ("), or triple quotes ''' or """) for multiline strings.

single_quoted_str = 'Hello, World!'

double_quoted_str = "Hello, World!"

multiline_str = '''This is a

multiline

string.'''

Accessing String in python

We can print String directly in python

Eg:

```
x="NCCS"

print(x)
```

Output:

```
NCCS
```

Accessing characters in Python String

- We can access individual characters of a string using indexing.
  - We can access individual characters using either positive or negative indexing
- Positive Indexing
  - The traditional method of string indexing is to count upwards from zero starting with the leftmost character in the string.
  - The index increments by one with each character as the strings are read from left to right.
  - This is referred to as positive indexing.
  - The first character has an index of 0

N	C	C	S
0	1	2	3

- Negative Indexing
  - The characters in a string can also be indexed counting back from the end of the string. This is known as negative indexing.
  - The final character of the string has a negative index of -1. The second last character occupies position -2, and so on.

N	C	C	S
-4	-3	-2	-1

Example

```
testString = "NCCS"

print("Output: ",testString[0])
```

Output

Output: N

Example

```
testString = "NCCS"

print("Output: ",testString[-1])
```

Output

Output: S

String Slicing

- The String Slicing method is used to access a range of characters in the String.
- Slicing in a String is done by using a Slicing operator, i.e., a colon (:).
- One thing to keep in mind while using this method is that the string returned after slicing includes the character at the start index but not the character at the last index.

eg:

```
string = 'Hello'

print(string[1:4])
```

Output

ell

If we want to print value of last index then

```
string = 'Hello'

print(string[1:5])
```

Why python excluded last position index

- In Python, slicing notation follows design choice of
  - [start:stop]
  - the stop index in slicing notation is exclusive, meaning the slice operation stops before the character at the stop index is reached

Design choice advantages:

- **Consistency with Indexing:** Python indexing starts from 0, and the stop index in slicing follows the same convention. This consistency makes it easier to understand and predict the behavior of slicing operations.
- **Convenient for Range Calculation:** By using the exclusive stop index, the length of the slice can be easily calculated as stop - start, without needing to add 1 or subtract 1.
- **Avoiding Ambiguity:** Inclusive stop index could introduce ambiguity in cases where both the start and stop indices are specified, as it would not be clear whether the character at the stop index should be included in the slice or not.

What will be the output

```
s = 'HelloWorld'

print(s[:])

print(s[::])
```

Output

HelloWorld

HelloWorld

Note:

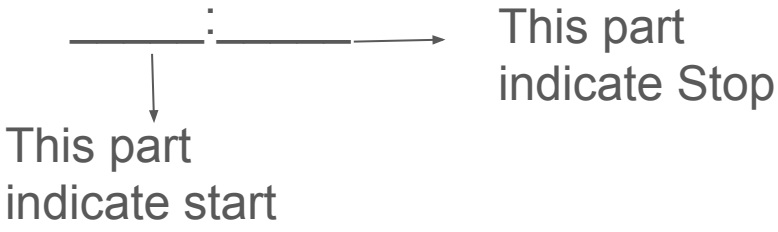
Python String slicing always follows this rule: s[:i] + s[i:] == s for any index ‘i’.

All these parameters are optional - start_pos default value is 0, the end_pos default value is the length of string and step default value is 1.

Let’s study clearly

Here, we haven’t included step so, it defaults to 1

If we write s[:] then,

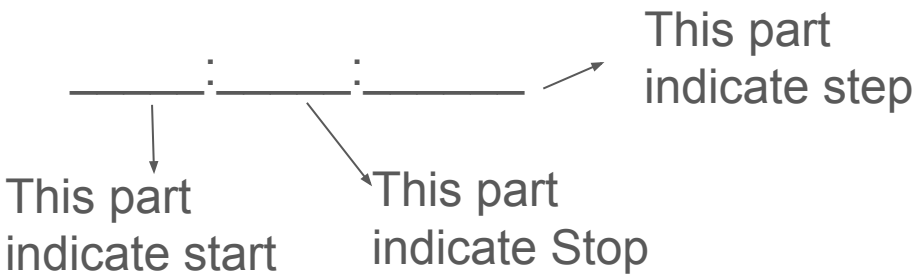


Remember :
start specifies the starting index of the slice (inclusive).
stop specifies the stopping index of the slice (exclusive).
If
start is omitted, it defaults to the beginning of the string.
stop is omitted, it defaults to the end of the string.

Let’s study clearly

Here, we haven’t included step so, it defaults to 1

If we write s[::] then,



Remember :
start specifies the starting index of the slice (inclusive).
stop specifies the stopping index of the slice (exclusive).
step specifies the step size or the increment between characters in the slice.
If
start is omitted, it defaults to the beginning of the string.
stop is omitted, it defaults to the end of the string.
step is omitted, it defaults to 1, meaning every character is included.

What will be the output

```
s = 'HelloWorld'

print(s[:5])

print(s[2:5])

print(s[2:])
```

Hello

llo

lloWorld

What will be the output

```
s = 'HelloWorld'

print(s[::-1])
```

dlroWolleH

Note: We can reverse a string using slicing by providing the step value as -1.

The negative step size indicates that the slicing will occur in reverse order, effectively reversing the string.

What will be the output

```
s = 'HelloWorld'

s1 = s[2:8:2]

print(s1)
```

loo

Here we have define our step as two so every other character is taken

What will be the output

```
s = 'HelloWorld'

s1 = s[8:1:-1]

print(s1)
```

Output

lroWoll

Here the index values are taken from end to start. The substring is made from indexes 1 to 7 from end to start.

	H	e	l	l	o	W	o	r	l	D
Start to End ->	0	1	2	3	4	5	6	7	8	9
	9	8	7	6	5	4	3	2	1	0

<- End to Start

String Concatenation

You can concatenate strings using the + operator.

```
first_name = "NCCS"

last_name = "COLLEGE"

full_name = first_name + " " + last_name

print(full_name)
```

String Formatting

Strings in Python or string data type in Python can be formatted with the use of format() method which is a very versatile and powerful tool for formatting Strings. Format method in String contains curly braces {} as placeholders which can hold arguments according to position or keyword to specify the order.

String formatting in Python can be done using various methods, including old-style formatting with %, the str.format() method, and f-strings (formatted string literals).

Old-style formatting with %

This method is similar to the C language's printf function. You use % to indicate a placeholder in the string and provide the values to replace the placeholders.

```
name = "John"

age = 30

print("My name is %s and I am %d years old." % (name, age))
```

str.format() method

This method provides more flexibility and readability compared to old-style formatting. You can use {} as placeholders and pass values to format() method.

```
name = "John"

age = 30

print("My name is {} and I am {} years old.".format(name, age))
```

f-strings (formatted string literals)

Introduced in Python 3.6, f-strings provide a concise and readable way to format strings. You prefix the string with f or F and use {} to insert variables directly into the string.

```
name = "John"

age = 30

print(f"My name is {name} and I am {age} years old.")
```

Escape Sequencing in Python

Escape sequences in Python are special characters preceded by a backslash (\) that are used to represent characters that are difficult or impossible to represent directly in a string.

\n: Newline - Inserts a newline character.

\t: Tab - Inserts a tab character.

\r: Carriage Return - Moves the cursor to the beginning of the line.

\\: Backslash - Inserts a literal backslash.

\': Single Quote - Inserts a single quote character.

\": Double Quote - Inserts a double quote character.

\b: Backspace - Deletes the previous character.

\f: Formfeed - Inserts a formfeed character.

\ooo: Octal value - Inserts the ASCII character represented by the octal number.

\xhh: Hex value - Inserts the ASCII character represented by the hexadecimal number.

```
print("Hello\nWorld")   # Output: Hello
                        #         World
print("Hello\tWorld")   # Output: Hello    World
print("Hello\\World")   # Output: Hello\World
print(\'Single quotes\')  # Output: 'Single quotes'
print(\"Double quotes\"")  # Output: "Double quotes"
print("Hello\bWorld")   # Output: HellWorld
print("Hello\fWorld")   # Output: Hello
                        #         World
print("\101")   # Output: A (ASCII character represented by octal number)
print("\x41")   # Output: A (ASCII character represented by hexadecimal number)
```

Python String Reversed

We can also reverse a string by using built-in join and reversed functions, and passing the string as the parameter to the reversed() function.

```
# Program to reverse a string


a = "NCCSCOLLEGE"

# Reverse the string using reversed and join function

a = "".join(reversed(a))


print(a)
```

Boolean

Python boolean type is one of the built-in data types provided by Python, which represents one of the two values i.e. True or False.

Generally, it is used to represent the truth values of the expressions.

A = True

B = False

C = (1==3)

The bool() in-built Function

The bool() method in Python returns a boolean value and can be used to cast a variable to the type Boolean. It takes one parameter on which you want to apply the procedure. However, passing a parameter to the bool() method is optional, and if not passed one, it simply returns False.

```
A = 1.123

print(bool(A))

A = 23

B = 23.01

print(bool(A==B))
```

List

Introduction

- In Python, lists are ordered collections of items that allow for easy use of a set of data.
- List values are placed in between square brackets [ ], separated by commas. It is good practice to put a space between the comma and the next value. The values in a list do not need to be unique (the same value can be repeated).
- Empty lists do not contain any values within the square brackets.
- They are used to store collections of items, which can be of any data type, and are mutable, meaning they can be changed after they are created.

Example

```
fruits = ["apple", "banana", "cherry"]

print(fruits)
```

Output: ['apple', 'banana', 'cherry']

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

eg:

```
fruits = ["apple", "banana", "cherry"]

print(fruits[0])
```

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Allow Duplicates

Since lists are indexed, lists can have items with the same value:

```
fruits = ["apple", "banana", "cherry", "apple", "cherry"]

print(fruits)
```

List Length

To determine how many items a list has, use the len() function:

```
fruits = ["apple", "banana", "cherry", "apple", "cherry"]

print(len(fruits))
```

List Items - Data Types

List items can be of any data type:

list1 = ["apple", "banana", "cherry"]

list2 = [1, 5, 7, 9, 3]

list3 = [True, False, False]

A list can contain different data types:

```
list1 = ["abc", 34, True, 40, "male"]
```

type()

the type() function is used to determine the data type of a given object. It returns the type of the object.

Eg:

```
fruits = ["apple", "banana", "cherry"]

print(type(fruits))
```

Output:

```
<class 'list'>
```

The list() Constructor

It is also possible to use the list() constructor when creating a new list.

```
fruits = list(("apple", "banana", "cherry"))

print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
```

Indexing and Slicing

Indexing refers to accessing individual elements of a list by their position. In Python, indexing starts at 0 for the first element, 1 for the second element, and so on. You can also use negative indices to access elements from the end of the list.

```
fruits = ["apple", "banana", "cherry"]

print(fruits[0])

print(fruits[-1])
```

Slicing allows you to extract a sublist (or a portion) of a list by specifying a range of indices.

The syntax for slicing is list[start:end:step], where start is the starting index (inclusive), end is the ending index (exclusive), and step is the step size.

```
my_list = ['a', 'b', 'c', 'd', 'e']

print(my_list[1:4])

print(my_list[:3])

print(my_list[2:])

print(my_list[::2])

print(my_list[::-1])
```

Output

['b', 'c', 'd'] (from index 1 to index 3)

['a', 'b', 'c'] (from index 0 to index 2)

['c', 'd', 'e'] (from index 2 to the end)

['a', 'c', 'e'] (every second element)

['e', 'd', 'c', 'b', 'a'] (reverse the list)

Changing Items

We can change the value of specific items in a list by assigning a new value to the corresponding index.

Lists are mutable, meaning you can modify them after they are created.

Eg:

```
my_list = [1, 2, 3, 4, 5]

# Change the value of the item at index 2

my_list[2] = 10

print(my_list)  # Output: [1, 2, 10, 4, 5]
```

The original list my_list contains the integers from 1 to 5.

We use indexing (my_list[2]) to access the item at index 2, which is originally 3.

We assign a new value 10 to the item at index 2 using the assignment operator (=).

After the assignment, the item at index 2 in my_list is changed from 3 to 10.

Notes:

We can change the value of any item in a list by assigning a new value to its corresponding index.

The new value can be of the same or different data type as the original value.

Adding and Removing Items

In Python, you can add and remove items from a list dynamically. Lists provide several methods for adding and removing items, allowing you to modify the contents of the list as needed.

Adding Items

append(): The append() method adds an item to the end of the list.

insert(): The insert() method inserts an item at a specified position in the list.

Eg

```
my_list = [1, 2, 3]

my_list.append(4)

print(my_list)
```

Output: [1, 2, 3, 4]

```
my_list = [1, 2, 3]

my_list.insert(1, 5)

print(my_list)
```

Output: [1, 5, 2, 3]

We cannot add item in list like this:

```
my_list = [1, 2, 3]

my_list=[4]

print(my_list)
```

This will simply replace previous item with 4 and output will be

```
[4]
```

Removing Items

remove(): The remove() method removes the first occurrence of a specified value from the list.

pop(): The pop() method removes the item at the specified index (or the last item if index is not provided) and returns it.

del keyword: The del keyword can be used to remove an item at a specified index or delete the entire list.

eg:

```
my_list = [1, 2, 3, 4, 3]

my_list.remove(3)

print(my_list)
```

Output: [1, 2, 4, 3]

```
my_list = [1, 2, 3, 4]
popped_item = my_list.pop(1)
print(popped_item)
print(my_list)
```

Output:

```
2
[1, 3, 4]
```

```
my_list = [1, 2, 3, 4]
del my_list[1]
print(my_list)
del my_list
```

Output: [1, 3, 4]

Loop Through a List

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  print(x)
```

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the range() and len() functions to create a suitable iterable.

```
fruits = ["apple", "banana", "cherry"]

for i in range(len(fruits)):

print(thislist[i])
```

Using a While Loop

```
fruits = ["apple", "banana", "cherry"]

i = 0

while i < len(fruits):

  print(thislist[i])

  i = i + 1
```

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

A short hand for loop that will print all items in a list

```
fruits = ["apple", "banana", "cherry"]

[print(x) for x in fruits]
```

Copying item of list

In Python, when you need to make a copy of a list, you have to be careful because lists are mutable objects.

If you simply assign a list to a new variable, changes made to one list may affect the other.

Therefore, you need to create a copy of the list if you want to modify one list without affecting the other.

Using the copy() method

You can use the copy() method to create a shallow copy of the list.

```
fruits = ["apple", "banana", "cherry"]

copied_fruits= fruits.copy()

print(copied_fruits)
```

Using list slicing

You can use list slicing to create a copy of the list.

```
fruits = ["apple", "banana", "cherry"]

copied_fruits= fruits[:]

print(copied_fruits)
```

Using the list() constructor:

You can use the list() constructor to create a new list from the elements of the original list.

```
fruits = ["apple", "banana", "cherry"]

copied_fruits= list(fruits)

print(copied_fruits)
```

Using the copy module

You can use the copy module's copy() function to create a shallow copy of the list.

```
import copy
fruits = ["apple", "banana", "cherry"]
copied_fruits = copy.copy(fruits)
print(fruits)
```

List Comprehension

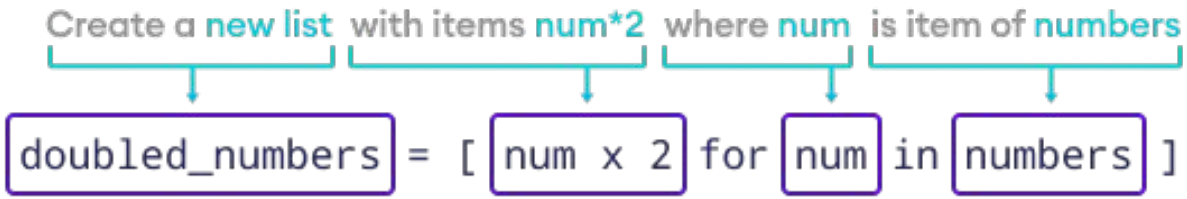
List comprehension offers a concise way to create a new list based on the values of an existing list.

```
numbers = [1, 2, 3, 4]

# list comprehension to create new list

doubled_numbers = [num * 2 for num in numbers]

print(doubled_numbers)
```

Output

[2, 4, 6, 8]

Here is how the list comprehension works:



Syntax of List Comprehension

[expression for item in list if condition == True]

Here,

for every item in list, execute the expression if the condition is True.

Note: The if statement in list comprehension is optional.

for Loop vs. List Comprehension

```
numbers = [1, 2, 3, 4, 5]

# for loop example

for num in numbers:

    print(num)
```

```
numbers = [1, 2, 3, 4, 5]

# list comprehension example

[print(num) for num in numbers]
```

Conditionals in List Comprehension

```
even_numbers = [num for num in range(1, 10) if num % 2 == 0 ]

print(even_numbers)


# Output: [2, 4, 6, 8]
```

Sorting

Sorting is a common operation in programming where the elements of a collection are arranged in a specific order.

In Python, you can sort lists using the built-in sorted() function or the sort() method of lists.

Using sorted() function

The sorted() function returns a new sorted list from the elements of any iterable object.

Syntax:

sorted_list = sorted(iterable, key=None, reverse=False)

iterable: The iterable object (e.g., list, tuple, string) to be sorted.

key (optional): A function that defines the sorting criteria.

reverse (optional): A boolean value indicating whether to sort in reverse order (default is False).

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]

sorted_numbers = sorted(numbers)

print(sorted_numbers)
```

[1, 1, 2, 3, 4, 5, 6, 9]

Using sort() method:

The sort() method sorts the elements of a list in place.

Syntax:

list.sort(key=None, reverse=False)

key (optional): A function that defines the sorting criteria.

reverse (optional): A boolean value indicating whether to sort in reverse order (default is False).

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]

numbers.sort()

print(numbers)
```

Output

[1, 1, 2, 3, 4, 5, 6, 9]

Sorting in reverse order

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]

sorted_numbers = sorted(numbers, reverse=True)

print(sorted_numbers)
```

Output:

[9, 6, 5, 4, 3, 2, 1, 1]

Sorting with a custom key function

```
words = ['banana', 'apple', 'orange', 'grape']

sorted_words = sorted(words, key=len)

print(sorted_words)
```

Output

['apple', 'grape', 'banana', 'orange']

Joining Lists

In Python, you can concatenate two or more lists to create a single list.

There are multiple ways to achieve this.

Using the + operator

```
list1 = [1, 2, 3]

list2 = [4, 5, 6]

joined_list = list1 + list2
```

Using the extend() method

The extend() method appends all the elements from another list to the end of the current list.

```
list1 = [1, 2, 3]

list2 = [4, 5, 6]

list1.extend(list2)
```

Tuple

Introduction

Python tuples are a type of data structure that is very similar to lists.

The main difference between the two is that tuples are immutable, meaning they cannot be changed once they are created.

This makes them ideal for storing data that should not be modified, such as database records.

A tuple can have any number of items, which may be of different types, such as a string, integer, float, list, etc.

Creating a Tuple

Tuples can be created in a number of ways. The most common way is to wrap them in parentheses. Both single and multiple tuples must always be followed by a comma.

my_tuple = (item1,) // single tuple

my_tuple = (item1, item2, item3) // multiple tuple

You can also create a tuple without using parentheses, by separating the values with commas:

my_tuple = 1, 2, 3

Python Tuple Types

There are two main types of tuples: named tuples and unnamed tuples.

- **Named Tuples:**

```
my_tuple = MyTuple(("one", "two", "three"))
print(my_tuple.one) // "one"
```

- **Unnamed Tuples:**

```
my_tuple = (1, 2, 3)
print(my_tuple) // (1, 2, 3)
```

Python Tuple Operations

Accessing Tuples

Tuples can be accessed just like lists, using square brackets and the index of the element you want to access. For example:

```
my_tuple = ("a", "b", "c")

print(my_tuple[1])
```

Output: b

You can also access elements from the end of the tuple using negative indices:

```
my_tuple = ("a", "b", "c")

print(my_tuple[-3])
```

Output:

```
a
```

Updating Tuple value

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

Since tuples are immutable, they do not have a built-in append() method, but there are other ways to add items to a tuple.

Convert into a list: Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

```
fruits = ("apple", "banana", "cherry")

new_list = list(fruits)

new_list.append("orange")

fruits = tuple(new_list)

print(fruits)
```

Output: ('apple', 'banana', 'cherry', 'orange')

Add tuple to a tuple: we are allowed to add tuples to tuples, so if we want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

```
fruits = ("apple", "banana", "cherry")

new_tuple = ("orange",)

fruits += new_tuple

print(fruits)
```

Output: ('apple', 'banana', 'cherry', 'orange')

Remove Items

Tuples are unchangeable, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
fruits = ("apple", "banana", "cherry")

new_list = list(fruits)

new_list.remove("apple")

fruits = tuple(new_list)

print(fruits)
```

Output: ('banana', 'cherry')

We can delete the tuple completely:

```
fruits = ("apple", "banana", "cherry")

del fruits

print(fruits) #this will raise an error because the tuple no longer exists
```

Slicing

Slicing tuples in Python allows you to extract specific portions or subsets of elements from a tuple.

Since tuples are immutable, slicing creates a new tuple containing the selected elements.

The general syntax for slicing a tuple is similar to slicing a list or a string:

new_tuple = old_tuple[start:end:step]

Eg

```
my_tuple = (1, 2, 3, 4, 5)

slice_tuple = my_tuple[1:4]

print(slice_tuple)
```

Output:

(2, 3, 4)

```
my_tuple = (1, 2, 3, 4, 5)


slice_tuple = my_tuple[-3:-1]

print(slice_tuple)
```

Output

(3, 4)

What will be the outpur

```
my_tuple = (1, 2, 3, 4, 5)

slice_tuple = my_tuple[:3]

print(slice_tuple)


slice_tuple = my_tuple[2:]

print(slice_tuple)
```

Output:

(1, 2, 3)

(3, 4, 5)

```
my_tuple = (1, 2, 3, 4, 5)

slice_tuple = my_tuple[::2]

print(slice_tuple)  slice_tuple = my_tuple[::-1]

print(slice_tuple)
```

Output:

(1, 3, 5)

(5, 4, 3, 2, 1)

Unpacking tuples

Unpacking tuples in Python refers to the process of assigning the individual elements of a tuple to separate variables.

This allows you to quickly and conveniently access the elements of a tuple without having to index each element explicitly.

Basic Unpacking

You can unpack a tuple by assigning its elements to separate variables in a single statement.

```
my_tuple = (1, 2, 3)

a, b, c = my_tuple

print(a)  # Output: 1

print(b)  # Output: 2

print(c)  # Output: 3
```

Extended Unpacking

Python also supports extended unpacking, where you can assign a variable to collect multiple elements of a tuple as a single entity.

```
my_tuple = (1, 2, 3, 4, 5)

first, *middle, last = my_tuple

print(first)    # Output: 1

print(middle)   # Output: [2, 3, 4]

print(last)     # Output: 5
```

What will be the output

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, *yellow) = fruits

print(green)

print(yellow)
```

Output

apple

['banana', 'cherry', 'strawberry', 'raspberry']

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
```

Output

apple

banana

['cherry', 'strawberry', 'raspberry']

Swapping Values:

Unpacking tuples is commonly used for swapping the values of variables.

```
a = 1

b = 2

a, b = b, a

print(a, b)  # Output: 2 1
```

Tuple Looping

Looping through tuples in Python allows you to iterate over each element in the tuple and perform operations or access its values. Since tuples are iterable objects, you can use various looping constructs such as for loops to iterate through them.

Using a for Loop

```
my_tuple = (1, 2, 3, 4, 5)

for item in my_tuple:

    print(item)
```

Using a While Loop

```
my_tuple = (1, 2, 3, 4, 5)

index = 0

while index < len(my_tuple):
    print(my_tuple[index])
    index += 1
```

Unpacking Tuples in a Loop

```
list_of_tuples = [(1, 'a'), (2, 'b'), (3, 'c')]

for number, letter in list_of_tuples:

    print(f"Number: {number}, Letter: {letter}")
```

Output

Number: 1, Letter: a

Number: 2, Letter: b

Number: 3, Letter: c

What will be the output

```
list_of_tuples = [('a',1), ('b',2), (3, 'c'),(8,'nccs')]

for number, letter in list_of_tuples:

    print(f"Number: {number}, Letter: {letter}")
```

Number: a, Letter: 1

Number: b, Letter: 2

Number: 3, Letter: c

Number: 8, Letter: nccs

Tuple Joining

In Python, joining tuples typically refers to concatenating multiple tuples together to create a single tuple.

Since tuples are immutable, you cannot directly append or modify them.

However, you can create a new tuple by combining the elements of multiple tuples.

Using the + Operator

You can use the + operator to concatenate two or more tuples into a single tuple.

```
tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

joined_tuple = tuple1 + tuple2

print(joined_tuple)
```

Output: (1, 2, 3, 4, 5, 6)

Using Tuple Unpacking and the + Operator

```
tuple1 = (1, 2)

tuple2 = (3, 4)

tuple3 = (5, 6)

joined_tuple = tuple1 + tuple2 + tuple3

print(joined_tuple)
```

Output: (1, 2, 3, 4, 5, 6)

Using the * Operator:

You can use the * operator to repeat a tuple and then concatenate them.

tuple1 = (1, 2)

repeated_tuple = tuple1 * 3

print(repeated_tuple)

Output: (1, 2, 1, 2, 1, 2)

Set

A set is an unordered collection of unique elements.

Sets are mutable, meaning you can add or remove elements, but each element must be unique.

Sets are useful for storing elements where duplicates are not allowed and for performing mathematical set operations such as union, intersection, and difference.

Creating a Set

```
fruits = {"apple", "banana", "cherry"}

print(fruits)
```

Duplicates Not Allowed

```
fruits = {"apple", "banana", "cherry", "apple"}


print(fruits)
```

Note:

The values True and 1 are considered the same value in sets, and are treated as duplicates

The values False and 0 are considered the same value in sets, and are treated as duplicates:

```
set = {"apple", "banana", "cherry", True, 1, 2}

print(set)
```

```
{'apple', True, 2, 'cherry', 'banana'}
```

Accessing element of set

Accessing elements in a set in Python is different from accessing elements in a list or a tuple because sets are unordered collections.

This means you cannot access elements by their index since sets are unordered the items has no index.

Iterating Through a Set

The most common way to access elements in a set is by iterating through it using a loop.

```
my_set = {1, 2, 3, 4, 5}

for element in my_set:
    print(element)
```

Checking Membership

You can check if an element is present in a set using the in keyword.

```
my_set = {1, 2, 3, 4, 5}

print(3 in my_set)

print(6 in my_set)
```

Output:

```
True

False
```

Adding and Removing Items

You can add elements to a set using the add() method for single elements or the update() method for multiple elements.

Adding a Single Element

Use the add() method to add a single element to a set.

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}

# Adding an existing element has no effect
my_set.add(3)
print(my_set)  # Output: {1, 2, 3, 4}
```

Adding Multiple Elements:

Use the update() method to add multiple elements to a set. The update() method can take any iterable (e.g., list, tuple, set).

```
my_set = {1, 2, 3}

my_set.update([4, 5, 6])

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


# Adding elements from another set

my_set.update({7, 8})

print(my_set)  # Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

remove() Parameters

The remove() takes only one parameter as the input from the user. This is the element that must be removed from the set.

```
my_set = {1, 2, 3, 4, 5}

my_set.remove(3)

print(my_set)

Output: {1, 2, 4, 5}

# Trying to remove an element that doesn't exist raises a KeyError

my_set.remove(10)  # Raises KeyError
```

Clearing All Elements:

Use the clear() method to remove all elements from the set, leaving it empty.

```
my_set = {1, 2, 3, 4, 5}

my_set.clear()

print(my_set)
```

Output: set()

Discarding a Specific Element:

Use the discard() method to remove a specific element. If the element is not present, discard() does nothing (no error is raised).

```
my_set = {1, 2, 3, 4, 5}

my_set.discard(3)

print(my_set)
```

Output: {1, 2, 4, 5}

```
# Discarding an element that doesn't exist does nothing

my_set.discard(10)

print(my_set)
```

Output: {1, 2, 4, 5}

Set Operations

Set operations in Python allow you to perform various mathematical operations on sets, such as union, intersection, difference, and symmetric difference.

These operations are useful for comparing sets and manipulating the elements they contain.

Here's a list of set operations:

- Union
- Intersection
- Difference
- Symmetric Difference
- Subset and Superset
- Disjoint Sets

Union

The union of two sets contains all the elements from both sets, without duplicates. You can use the ‘|’ operator or the union() method.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
# Using the | operator
union_set = set1 | set2
print(union_set)  # Output: {1, 2, 3, 4, 5}

# Using the union() method
union_set = set1.union(set2)
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

Intersection

The intersection of two sets contains only the elements that are present in both sets. You can use the ‘&’ operator or the intersection() method.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Using the & operator
intersection_set = set1 & set2
print(intersection_set)  # Output: {3}

# Using the intersection() method
intersection_set = set1.intersection(set2)
print(intersection_set)  # Output: {3}
```

Difference

The difference of two sets contains the elements that are in the first set but not in the second set. You can use the ‘-’ operator or the difference() method.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Using the - operator
difference_set = set1 - set2
print(difference_set)  # Output: {1, 2}

# Using the difference() method
difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2}
```

Symmetric Difference

The symmetric difference of two sets contains the elements that are in either of the sets but not in both. You can use the “^” operator or the symmetric_difference() method.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Using the ^ operator
symmetric_difference_set = set1 ^ set2
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}

# Using the symmetric_difference() method
symmetric_difference_set = set1.symmetric_difference(set2)
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

Subset and Superset

You can check if a set is a subset or superset of another set using the issubset() and issuperset() methods.

Example

```
set1 = {1, 2, 3}
set2 = {1, 2}

# Checking if set2 is a subset of set1
print(set2.issubset(set1))  # Output: True

# Checking if set1 is a superset of set2
print(set1.issuperset(set2))  # Output: True
```

Disjoint Sets

You can check if two sets are disjoint (i.e., they have no elements in common) using the isdisjoint() method.

Example

```
set1 = {1, 2, 3}
set2 = {4, 5, 6}

# Checking if set1 and set2 are disjoint
print(set1.isdisjoint(set2))  # Output: True

set3 = {3, 4, 5}

# Checking if set1 and set3 are disjoint
print(set1.isdisjoint(set3))  # Output: False
```

Frozenset

A frozenset in Python is an immutable version of a set.

While a normal set is mutable and allows adding and removing elements, a frozenset is immutable and does not support methods that modify the set, such as add(), remove(), and pop().

However, frozenset supports all other set operations that do not involve modifying the set, such as union, intersection, difference, and symmetric difference.

Creating a Frozenset

You can create a frozenset by passing an iterable to the frozenset() constructor.

```
# Creating a frozenset from a list
frozen_set = frozenset([1, 2, 3, 4, 5])
print(frozen_set)  # Output: frozenset({1, 2, 3, 4, 5})

# Creating a frozenset from a set
normal_set = {1, 2, 3}
frozen_set = frozenset(normal_set)
print(frozen_set)  # Output: frozenset({1, 2, 3})
```

```
# tuple of vowels
vowels = ('a', 'e', 'i', 'o', 'u')

fSet = frozenset(vowels)
print('The frozen set is:', fSet)
print('The empty frozen set is:', frozenset())

# frozensets are immutable
fSet.add('v')
```

```
The frozen set is: frozenset({'a', 'o', 'u', 'i', 'e'})
The empty frozen set is: frozenset()
Traceback (most recent call last):
  File "<string>, line 8, in <module>
    fSet.add('v')
AttributeError: 'frozenset' object has no attribute 'add'
```

frozenset() for Dictionary

When you use a dictionary as an iterable for a frozen set, it only takes keys of the dictionary to create the set.

```
# random dictionary

person = {"name": "John", "age": 23, "sex": "male"}

fSet = frozenset(person)

print('The frozen set is:', fSet)
```

Output

```
The frozen set is: frozenset({'name', 'sex', 'age'})
```

Frozenset operations

Like normal sets, frozenset can also perform different operations like copy(), difference(), intersection(), symmetric_difference(), and union().

```
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])

# copying a frozenset
C = A.copy()
print(C) # Output: frozenset({1, 2, 3, 4})
# union
print(A.union(B))  # Output: frozenset({1, 2, 3, 4, 5, 6})
# intersection
print(A.intersection(B))  # Output: frozenset({3, 4})
# difference
print(A.difference(B))  # Output: frozenset({1, 2})
# symmetric_difference
print(A.symmetric_difference(B))  # Output: frozenset({1, 2, 5, 6})
```

Similarly, other set methods like isdisjoint(), issubset(), and issuperset() are also available.

```
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])
C = frozenset([5, 6])
# isdisjoint() method
print(A.isdisjoint(C))  # Output: True
# issubset() method
print(C.issubset(B))  # Output: True
# issuperset() method
print(B.issuperset(C))  # Output: True
```

Range

- In Python, range is a built-in function used to generate a sequence of numbers.
- It's commonly used for looping a specific number of times in for-loops.
- The range function can be used with one, two, or three arguments, and it produces an immutable sequence of numbers.

Syntax

range(stop)

range(start, stop)

range(start, stop, step)

- start: The starting point of the sequence (inclusive). The default is 0.
- stop: The ending point of the sequence (exclusive).
- step: The difference between each number in the sequence. The default is 1.

```
for i in range(5):
    print(i)

for i in range(2, 7):
    print(i)

for i in range(1, 10, 2):
    print(i)
```

Dictionary

In Python, a dictionary is a collection of key-value pairs where each key is unique and maps to a value.

Dictionaries are unordered (prior to Python 3.7), mutable, and indexed by keys, which can be of any immutable data type (like strings, numbers, or tuples).

The values can be of any data type and can be duplicated.

Creating Dictionaries

You can create dictionaries in several ways:
Using Curly Braces {}:

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

print(my_dict)
```

Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

Using the dict() Constructor:

```
my_dict = dict(name='Alice', age=25, city='New York')

print(my_dict)
```

Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

Accessing Values

You can access dictionary values using their keys.

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

print(my_dict['name'])  # Output: Alice

print(my_dict['age'])   # Output: 25
```

Adding and Modifying Items

You can add or modify dictionary items by assigning a value to a key.

```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
my_dict['email'] = 'alice@example.com'  # Adding a new key-value pair
print(my_dict)
```

- Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'email': 'alice@example.com'}

```
my_dict['age'] = 26  # Modifying an existing key-value pair
print(my_dict)
```

- Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'email': 'alice@example.com'}

Removing Items

You can remove items from a dictionary using the del statement, pop() method, or popitem() method.

Using del:

```
my_dict = {'name': 'Alice', 'age': 26, 'city': 'New York', 'email': 'alice@example.com'}
del my_dict['city']
print(my_dict)
```

Output: {'name': 'Alice', 'age': 26, 'email': 'alice@example.com'}

Using pop():

{'name': 'Alice', 'age': 26, 'city': 'New York', 'email': 'alice@example.com'}

email = my_dict.pop('email')

print(email) # Output: alice@example.com

print(my_dict) # Output: {'name': 'Alice', 'age': 26,'city': 'New York'}

Using popitem():

popitem() removes and returns the last inserted key-value pair as a tuple.

This method is useful in versions of Python where dictionaries maintain insertion order (Python 3.7+).

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Removing and getting the last inserted key-value pair

last_item = my_dict.popitem()

print(last_item)  # Output: ('c', 3)

print(my_dict)    # Output: {'a': 1, 'b': 2}
```

Looping Through Dictionaries

You can loop through dictionaries to access keys, values, or both.

Looping through Keys:

```
for key in my_dict:
    print(key)
```

Looping through Values:

```
for value in my_dict.values():
    print(value)
```

Looping through Key-Value Pairs:

```
for key, value in my_dict.items():
    print(f'{key}: {value}')
```

Dictionary Methods

- keys(): Returns a view object with the dictionary's keys.

```
print(my_dict.keys())
```

- values(): Returns a view object with the dictionary's values.

```
print(my_dict.values())
```

- items(): Returns a view object with the dictionary's key-value pairs.

```
print(my_dict.items())
```

- get(key, default): Returns the value for key if key is in the dictionary, otherwise default value is printed.

```
age = my_dict.get('age', 'Unknown')

print(age)
```

- update(other_dict): Updates the dictionary with key-value pairs from another dictionary or iterable of key-value pairs.

```
my_dict.update({'age': 27, 'city': 'Boston'})

print(my_dict)
```

Binary Types

Binary types in Python refer to the data types used to represent binary data, such as bytes and bytearrays.

These data types are particularly useful when working with binary data, such as reading and writing files in binary mode, handling network protocols, or cryptographic operations.

Bytes

bytes is an immutable sequence of bytes, representing binary data.

Bytes literals are prefixed with a b or B.

Each byte in a bytes object represents an integer value between 0 and 255.

Bytes objects are immutable, meaning their values cannot be changed after creation.

Example

```
# Creating a bytes object from a byte literal
data = b'hello'


# Accessing individual bytes
print(data[0])  # Output: 104 (ASCII value of 'h')


# Bytes objects are immutable
# data[0] = 65  # Raises TypeError: 'bytes' object does not support item assignment
```

Bytearray

bytearray is a mutable sequence of bytes, representing binary data.

Bytearray literals are prefixed with bytearray().

Each byte in a bytearray object represents an integer value between 0 and 255.

Bytearray objects are mutable, allowing you to modify their values after creation.

Example

```
# Creating a bytearray object from a bytearray literal
data = bytearray(b'hello')

# Accessing individual bytes
print(data[0])  # Output: 104 (ASCII value of 'h')

# Modifying individual bytes
data[0] = 65
print(data)  # Output: bytearray(b'Aello')
```

None Type

In Python, NoneType is a special data type that represents the absence of a value or a null value. It is used to denote that a variable or an expression has no value or has not been assigned any value.

Characteristics of NoneType:

- None is a singleton object of type NoneType. There is only one None object in Python.
- It is often used as a placeholder or default value.
- Functions or methods that do not explicitly return a value implicitly return None.
- It is falsy in boolean contexts.(This means that when None is used in an expression that expects a boolean value, it will be treated as False.)

Example

```
x = None
# Checking if a variable is None
if x is None:
    print("x is None")
# Function returning None implicitly
def my_function():
    pass
result = my_function()
print(result)  # Output: None
# Explicitly returning None from a function
def another_function():
    return None
result = another_function()
print(result)  # Output: None
```