1- PYTHON INTRODUCTION - (TASK - 1)

- I hope everyone has installed the Anaconda software that I shared with you.
- I would like to know how many of you are familiar with any programming languages.
- If you don't know any programming language, then you are the ideal candidate to learn Python.
- Python is a very easy language to learn.
- What is Python? Answer: Python is a highly recommended programming language that is also object-oriented.
- The father of Python is Guido van Rossum.
- Python got its name from a fun TV show called "Monty Python's Flying Circus," which was broadcast on the BBC.
- Python borrows concepts from C, C++, Java, and Unix, making it a very powerful tool for various applications.
- Python was developed in the Netherlands, and many people consider it to be a new language.
- Java was released in 1995, while Python was officially released in 1989, with the first version made available on February 20, 1991.
- It has a large and comprehensive standard library.

- Python has gained immense popularity in the software industry because it allows developers to write concise code with minimal effort.
- The current market trends include machine learning, artificial intelligence, data science, and the Internet of Things (IoT).
- Many prominent companies use Python, including Google, NASA, Uber, Netflix, Reddit, Facebook (now Meta), and numerous others, as it is widely applicable across various sectors.
- Python code is easy to understand, and it is a dynamic programming language, which means that data types do not need to be declared explicitly.
- In Python, all operations are managed by the Python Virtual Machine (PVM).
- Python is accessible on any platform—Windows, Linux, and macOS—allowing code to run seamlessly across all these systems without the need for separate programs. Once code is written, it can be executed on any platform.
- As a dynamic programming language, Python does not require the declaration of data types.
- Python is free and open-source, making it simple to move code between platforms without modification.
- Python includes a rich set of libraries, such as NumPy and Pandas, making it an ideal choice for data science applications.
- However, Python is not suitable for mobile application development, such as Android apps.
- There are various flavors of Python: CPython (based on C), Jython (based on Java), IronPython (based on C#.NET), RubyPython (used for Ruby applications), and Anaconda (designed for big data and data science).
- Python 1.0 was introduced in January 1994, but it is no longer actively maintained.
- Python 2.0 was released in October 2000, and it is also no longer supported.
- Python 3.0 was launched in December 2008, with subsequent versions being released in 2016, 2017, and 2024, including the latest versions: 3.6, 3.7, 3.8, 3.9, 3.10, and 3.11.

In [11]:
```
import sys
sys.version
```

Out[11]:
```
'3.13.5 | packaged by Anaconda, Inc. | (main, Jun 12 2025, 16:37:03) [MSC v.1929 64 bit (AMD64)]'
```

In [1]:
```
x = 3
x
#id(x)
```

Out[1]:
```
3
```

In [2]:
```
x = 4
x
```

Out[2]:
```
4
```

In [3]:
```
y = 3
y
```

Out[3]:
```
3
```

In [4]:
```
x, y
```

Out[4]:
```
(4, 3)
```

In [5]:
```
x, y = 4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 x, y = 4

TypeError: cannot unpack non-iterable int object
```

In [6]:
```
type(x)
```

Out[6]:
```
int
```

In [7]:
```
y = 3
id(y)
```

Out[7]:
```
140714132091880
```

In [8]:
```
x1 = 4
id(y)
```

Out[8]:
```
140714132091880
```

In [9]:
```
y = False
```

In [10]:
```
type(y)
```

Out[10]:
```
bool
```

In [12]:
```
import sys
sys.version
```

Out[12]:
```
'3.13.5 | packaged by Anaconda, Inc. | (main, Jun 12 2025, 16:37:03) [MSC v.1929 64 bit (AMD64)]'
```

2 - Getting started with Python Language

2- Getting started with Python Language

Python 3.x

Version Release Date

- **Version** - **Date**
- 3.13 - 2022-25-08
- 3.10 - 2021-10-04
- 3.9 - 2020-10-05
- 3.8 - 2020-04-29
- 3.7 - 2018-06-27
- 3.6 - 2016-12-23
- 3.5 - 2015-09-13
- 3.4 - 2014-03-17
- 3.3 - 2012-09-29
- 3.2 - 2011-02-20
- 3.1 - 2009-06-26
- 3.0 - 2008-12-03

Python 2.x

Version Release Date

- **Version** - **Date**
- 2.7 - 2010-07-03
- 2.6 - 2008-10-02
- 2.5 - 2006-09-19
- 2.4 - 2004-11-30
- 2.3 - 2003-07-29
- 2.2 - 2001-12-21
- 2.1 - 2001-04-15
- 2.0 - 2000-10-16

In [13]:
```
import sys
sys.version
```

Out[13]:
```
'3.13.5 | packaged by Anaconda, Inc. | (main, Jun 12 2025, 16:37:03) [MSC v.1929 64 bit (AMD64)]'
```

In [14]:
```
a = 5
print(a)
type(a)
```

```
5
```

Out[14]:
```
int
```

In [15]:
```
a@ = 6
a@
```

```
  Cell In[15], line 1
    a@ = 6
       ^
SyntaxError: invalid syntax
```

In [16]:
```
6 = b
```

```
  Cell In[16], line 1
    6 = b
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

Verify if Python is installed

In [18]:
```
import sys
sys.version
```

Out[18]:
```
'3.13.5 | packaged by Anaconda, Inc. | (main, Jun 12 2025, 16:37:03) [MSC v.1929 64 bit (AMD64)]'
```

3 - Creating Variables & Assigning Values

To create a variable in Python, all you need to do is specify the variable name and then assign a value to it.

- The syntax is: **variable_name = value**
- In Python, the equals sign (=) is used to assign values to variables.
- There is no requirement to declare a variable in advance or to specify a data type for it.
- When you assign a value to a variable, you are both declaring and initializing it with that value.
- It is not possible to declare a variable without initially assigning a value to it. value.
- There's no way to declare a variable without assigning it an initial value.

In [19]:
```
a = 5
```

In [20]:
```
a
```

Out[20]:
```
5
```

In [21]:
```
type(a)
```

Out[21]:
```
int
```

In [22]:
```
b = 5.5
b
```

Out[22]:
```
5.5
```

In [23]:
```
type(b)
```

Out[23]:
```
float
```

In [24]:
```
c = 'hi'
```

In [25]:
```
c
```

Out[25]:
```
'hi'
```

In [26]:
```
type(str)
```

Out[26]:
```
type
```

In [27]:
```
x = 2
x
type(x)
```

Out[27]:
```
int
```

In [28]:
```
x = 2
print(x)
print(type(x))
```

```
2
<class 'int'>
```

In [29]:
```
x@ = 3
x@
```

```
  Cell In[29], line 1
    x@ = 3
       ^
SyntaxError: invalid syntax
```

In [30]:
```
# Integer
a = 2
type(a)
#a
```

Out[30]: int

In [31]:
```
# Integer
b = 7007522512
print(b)
```

```
7007522512
```

In [32]:
```
#Floating Point
pi = 3.14
```

In [33]:
```
print(pi)
```

```
3.14
```

In [34]:
```
type(pi)
```

Out[34]: float

In [35]:
```
#String
c = 'A'
print(c)
```

```
A
```

In [36]:
```
type(c)
```

Out[36]: str

In [37]:
```
#String
name = 'John Doe'
print(name)
type(name)
```

```
John Doe
```

Out[37]: str

In [38]:
```
# Boolean
q = True
print(q)
```

```
True
```

In [39]:
```
type(q)
```

Out[39]: bool

In [40]:
```
# Empty value or Null Type
x = None
print(x)
```

```
None
```

In [41]:
```
#Variable assignment functions from left to right, so the following will cause a syntax error.
5 = x
```

```
  Cell In[41], line 2
    5 = x
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

In [42]:
```
x = 5
x
```

Out[42]: 5

4 - Python: Identifiers, Variables, and Objects

- In programming, a person's name serves as an identifier, and similarly, Python uses names to identify various elements in code.
- The name used in a Python program for this purpose is called an IDENTIFIER (e.g., in the expression `x = 10`, `x` is the identifier).
- Just as there are rules for naming a child—such as choosing a name from the gods or ancestors, and conducting some research before making a decision—there are also rules for naming Python identifiers.

- For example, you wouldn't name a child "Cat" or "Dog," as parents must adhere to certain guidelines when selecting a child's name.
- **Here are the rules for defining a Python identifier:**

1. It can start with an alphabet letter (either uppercase or lowercase).
2. It may contain digits (0-9), but cannot begin with a digit.
3. It can include underscores (_).

In [43]:
```
ABC = 50
```

In [44]:
```
ABC
```

Out[44]: 50

In [45]:
```
abc = 60
abc
```

Out[45]: 60

In [47]:
```
Abc = 70
ABCD
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[47], line 2
      1 Abc = 70
----> 2 ABCD

NameError: name 'ABCD' is not defined
```

In [48]:
```
xyz = 20000
xyz
```

Out[48]: 20000

In [49]:
```
ABCD = 70
ABCD
```

Out[49]: 70

In [50]:
```
NIT = 15000
nit1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[50], line 2
      1 NIT = 15000
----> 2 nit1

NameError: name 'nit1' is not defined
```

In [51]:
```
nIT = 20
nIT
```

Out[51]: 20

In [52]:
```
montycorps = 78
montcorP
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[52], line 2
      1 montycorps = 78
----> 2 montcorP

NameError: name 'montcorP' is not defined
```

In [53]:
```
cash123 = 10
cash123
```

Out[53]: 10

In [54]:
```
123cash = 20
123cash
```

```
  Cell In[54], line 1
    123cash = 20
      ^
SyntaxError: invalid decimal literal
```

In [55]:
```
1A = 5
```

```
  Cell In[55], line 1
    1A = 5
    ^
SyntaxError: invalid decimal literal
```

In [56]:
```
A1 = 5
A1
```

Out[56]: 5

In [57]:
```
# x = 10, where x is the variable and 10 is the value.
cash = 10  # Identifier rules: alphabet
# ca$h = 10, the $ symbol is not allowed in Python identifiers but is allowed in Java.
# ca$h
cash
```

Out[57]: 10

In [58]:
```
ca$h = 20
ca$sh
```

```
  Cell In[58], line 1
    ca$h = 20
      ^
SyntaxError: invalid syntax
```

In [59]:
```
ca*h =20
ca*h
```

```
  Cell In[59], line 1
    ca*h =20
    ^
SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of '='?
```

In [60]:
```
CASH = 20
#Cash
CASH
```

Out[60]: 20

In [ ]:
```
# Identifiers should not start with a digit.
# sum123 = 20, digit case identifier.
123total = 30
# sum123,
```

In [ ]:
```
# Python identifiers are case sensitive
Abcde = 20
# total
type(Abcde)
```

In [62]:
```
new = 30
NEW
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[62], line 2
      1 new = 30
----> 2 NEW

NameError: name 'NEW' is not defined
```

In [63]:
```
Total5 = 30
#TOTAL
Total5
```

Out[63]: 30

In [64]:
```
def = 4.6
def
```

```
  Cell In[64], line 1
    def = 4.6
        ^
SyntaxError: invalid syntax
```

In [65]:
```
del = 9
```

```
  Cell In[65], line 1
    del = 9
        ^
SyntaxError: invalid syntax
```

In [66]:
```
import keyword
keyword.kwlist
```

Out[66]:
```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

In [67]:
```
len(keyword.kwlist)
```

Out[67]:
```
35
```

In [68]:
```
DEF = 4
DEF
```

Out[68]:
```
4
```

In [69]:
```
if = 780
if
```

```
  Cell In[69], line 1
    if = 780
       ^
SyntaxError: invalid syntax
```

In [70]:
```
IF = 780
IF
```

Out[70]:
```
780
```

In [71]:
```
DEF = 5.6
DEF
#Please note that "def" is a keyword in Python
```

Out[71]:
```
5.6
```

In [1]:
```
def = 7
def
```

```
  Cell In[1], line 1
    def = 7
        ^
SyntaxError: invalid syntax
```

In [3]:
```
import keyword
keyword.kwlist
```

```
Out[3]: ['False',
         'None',
         'True',
         'and',
         'as',
         'assert',
         'async',
         'await',
         'break',
         'class',
         'continue',
         'def',
         'del',
         'elif',
         'else',
         'except',
         'finally',
         'for',
         'from',
         'global',
         'if',
         'import',
         'in',
         'is',
         'lambda',
         'nonlocal',
         'not',
         'or',
         'pass',
         'raise',
         'return',
         'try',
         'while',
         'with',
         'yield']
```

```
In [4]: len(keyword.kwlist)
```

```
Out[4]: 35
```

4.1 - Keywords Cannot Be Used as Identifiers

```
In [5]: if = 10  # 'if' is a keyword
        def = 20  # 'def' is a keyword
        for = 50  # 'for' is a keyword
```

```
  Cell In[5], line 1
    if = 10  # 'if' is a keyword
       ^
SyntaxError: invalid syntax
```

```
In [6]: FOR = 58
```

```
In [7]: FOR
```

```
Out[7]: 58
```

```
In [8]: def = 30
        def
```

```
  Cell In[8], line 1
    def = 30
        ^
SyntaxError: invalid syntax
```

Rules for Python Identifiers

- Identifiers can include the letters A to Z (both uppercase and lowercase) and the digits 0 to 9.
- Identifiers cannot start with a digit.
- Identifiers are case sensitive.
- Reserved words or keywords in Python cannot be used as identifiers.
- There is no limit to the length of an identifier.
- The underscore ('_') is the only special character allowed.
- No other special characters are permitted.

```
In [9]: print('Hello')
```

```
Hello
```

Python Reserved Words

- When a child goes to school, they learn the alphabet from A to Z, followed by words that start with those letters, such as A for Apple, B for Ball, and C for Cat. In this context, Apple, Ball, and Cat can be considered reserved words in English. An apple is associated with the fruit, a ball is related to play, and a cat refers to an animal. There are many such words in the dictionary, commonly referred to as reserved words.
- In programming languages, reserved words exist as well. Today, we will focus on Python reserved words.
- Python has 35 reserved words. If you learn these 35 reserved words, you will have a strong foundation in Python. Each reserved word has its meaning and functionality. Therefore, learning Python effectively involves understanding the functionality of these reserved words.

35 Reserved Words in Python

- **True, False, None:** Represent Boolean data types.
- **and, or, not, is:** Represent logical operators.
- **if, else, elif:** Represent conditional statements (Note: Python does not have a switch or do...while statement).
- **while, for, break, continue, return, in, yield:** Represent looping constructs.
- **try, except, finally, raise, assert:** Used for exception handling and functionality.
- **import, from, as, class, def, pass, global, nonlocal, lambda, del, with:** Represent classes, methods, and functions.

Notes:

- The 35 reserved words are alphabetical, except for True, False, and None.

In [ ]:
```
# a = True  # hash is used for comments
```

In [11]:
```
a = True
a
```

Out[11]: True

In [13]:
```
a1 = true
a1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 a1 = true
      2 a1

NameError: name 'true' is not defined
```

True = a

In [14]:
```
b = None
#b = none
b
```

In [16]:
```
b = none
b
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[16], line 1
----> 1 b = none
      2 b

NameError: name 'none' is not defined
```

In [17]:
```
c = False
#c = false
c
```

Out[17]: False

In [18]:
```
true + true
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[18], line 1
----> 1 true + true

NameError: name 'true' is not defined
```

In [19]:
```
True + true
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[19], line 1
----> 1 True + true

NameError: name 'true' is not defined
```

In [20]:
```
True * True
```

Out[20]: 1

In [21]:
```
True / True #Float division
```

Out[21]: 1.0

In [22]:
```
True // True #Integer division
```

Out[22]: 1

In [23]:
```
False / True
```

Out[23]: 0.0

In [24]:
```
pi = 3.14
```

In [25]:
```
pi
```

Out[25]: 3.14

In [26]:
```
pi = 3.17
pi
```

Out[26]: 3.17

How to remember keywords for interview questions.

- A keyword is a module that can be run using the **import class**. Import the keyword module to access the list of keywords with `keyword.kwlist`.

In [27]:
```
kw = keyword.kwlist
kw
```

Out[27]:
```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

In [28]:
```
#Please write a code snippet to create an index for the following keywords.
import pandas as pd  # pandas is the module to create a DataFrame
df = pd.DataFrame(keyword.kwlist)  # Correct spelling of DataFrame
df  # Display the DataFrame
# Always remember Python indexing begins with '0'
```

Out[28]:

	0
0	False
1	None
2	True
3	and
4	as
5	assert
6	async
7	await
8	break
9	class
10	continue
11	def
12	del
13	elif
14	else
15	except
16	finally
17	for
18	from
19	global
20	if
21	import
22	in
23	is
24	lambda
25	nonlocal
26	not
27	or
28	pass
29	raise
30	return
31	try
32	while
33	with
34	yield

In [29]:
```
a = 10
id(a)
```

Out[29]:
```
140714119574728
```

Python Data Types

(14) Built-in Data Types

1. **int**
2. **float**
3. **complex**
4. **bool**
5. **str**
6. **bytes**
7. **bytearray**

Data Structures

8. **range**
9. **list**
10. **tuple**
11. **set**
12. **frozenset**
13. **dict**
14. **None**

- Python provides some built-in functions such as:

  1. `print()`
  2. `type()`
  3. `id()`
- The data types **int**, **float**, **complex**, and **bool** do not represent objects.

- Everything else is considered an object.

Note:
In Python, all 14 data types are objects.
This is why we refer to Python as an object-oriented programming language.

In [31]:
```
# What are the other built-in data types available besides the 14 standard data types?

a = 10
print(a)       # To display the variable
print(a)       # To display the value of the variable
print(type(a)) # To find the data type
print(id(a))   # To find the address of the object
print(type(a)) # Display the data type again
```

```
10
10
<class 'int'>
140714119574728
<class 'int'>
```

In [32]:
```
id(a)
```

Out[32]: 140714119574728

In [33]:
```
b = 10
b
```

Out[33]: 10

In [34]:
```
id(b)
```

Out[34]: 140714119574728

In [35]:
```
a = 10
b = 10
id(a)
```

Out[35]: 140714119574728

In [36]:
```
a = 10
a
id(a)
```

Out[36]: 140714119574728

Integer Data Types

- **Integer Data Types**: Numbers without decimal points are referred to as integral data types.

- **Representation of Integer Values**: There are three ways to represent integer values:

1. **Binary Form**: (Base-2) - uses digits 0 and 1
2. **Octal Form**: (Base-8) - uses digits 0 to 7
3. **Decimal Form**: (Base-10) - uses digits 0 to 9

This structure makes the information clearer.

In [37]:
```
a = 1111
a
```

Out[37]: 1111

In [38]:
```
type(a)
```

Out[38]:
```
int
```

In [39]:
```
id(a)
```

Out[39]:
```
2516865220752
```

In [41]:
```
#2. Binary Form (Base 2)
a = 1111  # Value is declared
b = 0b1111  # Now PVM(Python Virtual Machine) converts the value to binary
b
#a
```

Out[41]:
```
15
```

In [42]:
```
bin(15)
```

Out[42]:
```
'0b1111'
```

In [43]:
```
b_1 = 0b11
b_1
```

Out[43]:
```
3
```

In [46]:
```
bin(b_1)
```

Out[46]:
```
'0b11'
```

In [48]:
```
b_ = 0b22
b_
```

```
  Cell In[48], line 1
    b_ = 0b22
           ^
SyntaxError: invalid digit '2' in binary literal
```

In [49]:
```
b1 = 111
b1
```

Out[49]:
```
111
```

In [50]:
```
c = 0b111
c
```

Out[50]:
```
7
```

In [51]:
```
b3 = 0b2
b3
```

```
  Cell In[51], line 1
    b3 = 0b2
           ^
SyntaxError: invalid digit '2' in binary literal
```

In [52]:
```
True/False
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[52], line 1
----> 1 True/False

ZeroDivisionError: division by zero
```

In [53]:
```
b = 0b10 #The variable `b` is assigned the value `0b10`, which represents the number 2 in binary format.
b
```

Out[53]:
```
2
```

In [54]:
```
c = 0b100
c
```

Out[54]:
```
4
```

In [55]:
```
#3. Octal Form (Base 8)
a = 111  # Value is declared
b1 = 0o11  # Now Python converts the value to octal
b1
# a
```

Out[55]:
```
9
```

In [56]:
```
i = 0b22
```

```
  Cell In[56], line 1
    i = 0b22
          ^
SyntaxError: invalid digit '2' in binary literal
```

In [57]:
```
i1 = 0o22
i1
```

Out[57]: 18

In [58]:
```
b2 = 0o27
b2
```

Out[58]: 23

In [59]:
```
# Final summary of integral data types
a = 10
b = 0b10
c = 0o100
a
b
c
```

Out[59]: 64

In [61]:
```
c = 0O33
c
```

Out[61]: 27

In [62]:
```
b
```

Out[62]: 2

In [63]:
```
c
```

Out[63]: 27

In [64]:
```
A = 78
type(A)
```

Out[64]: int

Float Datatypes

- Employee salary: 5676.76
- Diesel price: 67.25
- These values are not whole numbers; they are known as decimal values.
- In Python, you cannot declare floating-point numbers in binary, octal, or hexadecimal formats, as the Python interpreter does not accept those.
- In school, we learn about exponential notation, such as (1.2e3), which is applicable to float datatypes. Only the letter **'e'** is allowed in this context.

In [65]:
```
b = 67.9
print(b)
```

67.9

In [66]:
```
type(b)
```

Out[66]: float

In [67]:
```
b1 = 0b1.1
b1
```

```
  Cell In[67], line 1
    b1 = 0b1.1
           ^
SyntaxError: invalid syntax
```

In [68]:
```
c = 0o11.6
c
```

```
  Cell In[68], line 1
    c = 0o11.6
           ^
SyntaxError: invalid syntax
```

In [69]:
```
# Float datatypes -
# b2 = 0b1111.22
# b2
# print(type(b2))
# type(b)
```

```
d = 0o4567.67  # This is octal
# e = 0b456789  # This is binary
d
# e
# b
```

```
  Cell In[69], line 6
    d = 0o4567.67  # This is octal
              ^
SyntaxError: invalid syntax
```

In [70]:
```
f1 = 1e4
f1
type(f1)
```

Out[70]:
```
float
```

In [71]:
```
f1
```

Out[71]:
```
10000.0
```

In [72]:
```
f = 1e3
f
```

Out[72]:
```
1000.0
```

In [73]:
```
f = 1e4  # only the letter 'e' is allowed.
g = 2.4E3  # except for 'E', you can't run any program.
g
f
type(g)
```

Out[73]:
```
float
```

In [74]:
```
g1 = 23e3
g1
```

Out[74]:
```
23000.0
```

In [75]:
```
e = 5.e3
e
```

Out[75]:
```
5000.0
```

In [76]:
```
type(e)
```

Out[76]:
```
float
```

Complex Datatypes

- The format for a complex datatype is: (a + bj) where (a) is the real part and (b) is the imaginary part. Here, (j^2 = -1).
- The value (j) is mandatory, and no other values can be used in the complex type.
- Since (j^2 = -1), it follows that (j = sqrt{-1}). This relationship is pure mathematics, illustrating why Python is an excellent choice for developing mathematical or scientific applications.
- For the real part, any numerical type can be used, but the imaginary part must be an integer.

In [77]:
```
x = 30 + 40j # Assigned integer values for the real part and the imaginary part.
x
```

Out[77]:
```
(30+40j)
```

In [78]:
```
type(x)
```

Out[78]:
```
complex
```

In [80]:
```
y = 1+2j  #A float value has been assigned to both the real and imaginary parts.
z = 3+2j  #A float value is assigned to the real part, while a real value is assigned to the imaginary part.
print(y + z)
print(y-z)
print(y*z)
print(y/z)
```

```
(4+4j)
(-2+0j)
(-1+8j)
(0.5384615384615384+0.30769230769230776j)
```

In [81]:
```
c = 15 + 0b111j  # Imaginary part cannot be in binary or octal.
d = 0b11 + 15j  # The real part can be in binary or octal.
print(c)
print(d)
```

```
  Cell In[81], line 1
    c = 15 + 0b111j  # Imaginary part cannot be in binary or octal.
                  ^
SyntaxError: invalid binary literal
```

In [82]:
```
c = 1 + 0b10j
c
```

```
  Cell In[82], line 1
    c = 1 + 0b10j
              ^
SyntaxError: invalid binary literal
```

In [83]:
```
d2 = 0b111+15j
d2
```

Out[83]: (7+15j)

In [84]:
```
e1 = 4 + 15j
e1
```

Out[84]: (4+15j)

In [85]:
```
a1 = 20+30j
b1 = 40+50j
print(a1+b1)
print(a1-b1)
print(a1*b1)
print(a1/b1)
```

```
(60+80j)
(-20-20j)
(-700+2200j)
(0.5609756097560976+0.04878048780487805j)
```

In [86]:
```
print(a1*b1)
```

```
(-700+2200j)
```

In [87]:
```
a = 2+3j
type(a)
```

Out[87]: complex

In [91]:
```
a1 = 10 + 20j  # I want to know the value of the real part and imaginary part
print(a1.real)  # The complex data type will be used in mathematical concepts, not just for programming languages.
#print(a1.imaginary)
print(a1.imag)
```

```
10.0
20.0
```

In [92]:
```
a1.real
```

Out[92]: 10.0

In [94]:
```
help()
```

```
Welcome to Python 3.13's help utility! If this is your first time using
Python, you should definitely check out the tutorial at
https://docs.python.org/3.13/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To get a list of available
modules, keywords, symbols, or topics, enter "modules", "keywords",
"symbols", or "topics".

Each module also comes with a one-line summary of what it does; to list
the modules whose name or summary contain a given string such as "spam",
enter "modules spam".

To quit this help utility and return to the interpreter,
enter "q", "quit" or "exit".

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

In [95]:
```
com = 10 + 15j
type(com)
```

Out[95]: complex

In [96]:
```
com.real
```

Out[96]: 10.0

In [98]:
```
com.imag
```

Out[98]:
```
15.0
```

Boolean Data Types -

- Allowed values: True/False
- False value -- 0 (internal memory conversion)
- True value -- 1

In [100…
```
a = 10
b = 20
c = a>b
c
```

Out[100…
```
False
```

In [101…
```
type(c)
```

Out[101…
```
bool
```

In [106…
```
print(True+True)
print(True*True)
print(True-True)
print(True/True)
print(True//True)
print(False+False)
print(False+True)
```

```
2
1
0
1.0
1
0
1
```

In [107…
```
print(True/False)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[107], line 1
----> 1 print(True/False)

ZeroDivisionError: division by zero
```

String Data Types

- Strings are enclosed in single quotes (' ') or double quotes (" ").
- For a single line, you can use either single quotes or double quotes.
- To define a multiline string, use triple quotes (''' ''' or """ """).
- Both single and double quotes can be used for single-line strings.
- Triple quotes are also allowed for multi-line comments and can be used in a single line as well.

In [108…
```
ABC = ''''Good for Data Science'''
ABC
```

Out[108…
```
"'Good for Data Science"
```

In [109…
```
type(ABC)
```

Out[109…
```
str
```

In [110…
```
DEF = '''Good for Data Science'''
#'Single Quotes'
print(DEF)
type(DEF)
```

```
Good for Data Science
```

Out[110…
```
str
```

In [111…
```
w = ''' Good
                for Data Science'''
print(w)
type(w)
```

```
Good
               for Data Science
```

Out[111…
```
str
```

In [112…
```
ts = '''The most common cause of the Python `SyntaxError: EOL(End Of Line) while scanning string literal` is a missing closing
```

```
        This error occurs when a string is initiated using single quotes ('), double quotes ("), or triple quotes (""") but is
```

In [113…
```
ts
```

Out[113…
```
'The most common cause of the Python `SyntaxError: EOL(End Of Line) while scanning string literal` is a missing closing quote at the end of a string. \n        This error occurs when a string is initiated using single quotes (\'), double quotes ("), or triple quotes (""") but is not properly closed.'
```

In [114…
```
y = 'good for datascience' #Single Quotes
y
```

Out[114…
```
'good for datascience'
```

In [115…
```
y = "good for datascience" #Double Quotes
y
```

Out[115…
```
'good for datascience'
```

In [117…
```
a = ''' hallo
how
are
you''' #Triple Quotes
a
```

Out[117…
```
' hallo\nhow \nare  \nyou'
```

In [118…
```
type(a)
```

Out[118…
```
str
```

In [119…
```
# Single quotes and double quotes are both equal.
# ''' This is a multiline comment '''
```

In [120…
```
b = '''('hello'
  'how'
    'are you')'''

    # ''' ''' indicates multiline comments
b
    # is used for commenting
```

Out[120…
```
"('hello' \n  'how'\n    'are you')"
```

In [121…
```
x, y, z, m, n = 10, True, 10.9, 1 + 10j, 'hi'
```

Out[121…
```
str
```

In [123…
```
type(x)
```

Out[123…
```
int
```

In [124…
```
type(y)
```

Out[124…
```
bool
```

In [125…
```
type(z)
```

Out[125…
```
float
```

In [126…
```
type(m)
```

Out[126…
```
complex
```

In [127…
```
type(n)
```

Out[127…
```
str
```

Type Conversion Functions:

- `int()`
- `float()`
- `complex()`
- `bool()`
- `str()`

int()

The `int()` function can convert other types to an integer, except for complex numbers.

Examples:

- `int(10.123)` # Converts float to int

- `int(10 + 20j)` # Cannot convert complex to int
- `int(True)` # Converts boolean True to int (1)
- `int(False)` # Converts boolean False to int (0)
- `int('10')` # Converts string '10' to int
- `int('ten')` # This will raise an error because 'ten' is not a valid integer representation

float()

The `float()` function can convert any type to a float, except for complex numbers.

Examples:

- `float(10)` # Converts int to float
- `float(10 + 20j)` # Cannot convert complex to float
- `float(False)` # Converts boolean False to float (0.0)
- `float('11')` # Converts string '11' to float

In [128…
```
float(10, 20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[128], line 1
----> 1 float(10, 20)

TypeError: float expected at most 1 argument, got 2
```

Using the `complex()` Function

The `complex()` function is used to convert various data types to a complex type. Below is a breakdown of how it works with different types of arguments.

Single Argument Conversion

The following examples demonstrate how to convert a single argument to a complex number:

- `complex(10)` # Converts an integer to a complex number
- `complex(10.5)` # Converts a float to a complex number
- `complex(True)` # Converts a boolean (True) to a complex number
- `complex(False)` # Converts a boolean (False) to a complex number
- `complex('10')` # Converts a string to a complex number

Two-Argument Conversion

The following examples show the behavior of the `complex()` function when provided with two arguments:

- `complex(10, 20)` # Converts integers to a complex number (10 + 20j)
- `complex(10, 20.5)` # Converts a mix of an integer and a float to a complex number (10 + 20.5j)
- `complex('10')` # Only provides one argument; two arguments cannot be assigned as a string

Note: When using two arguments, the first argument represents the real part and the second argument represents the imaginary part of the complex number. However, passing a string as a single argument will raise an error for the second argument.

The `bool()` function converts values to a boolean type. Here's how it interprets different types of input:

- `bool(0)` # Converts integer 0 to boolean (results in `False`)
- `bool(-10)` # Converts integer -10 to boolean (results in `True`)
- `bool(0.0)` # Converts float 0.0 to boolean (results in `False`)
- `bool(0.01)` # Converts float 0.01 to boolean (results in `True`)
- `bool(10 + 20j)` # Converts complex number 10 + 20j to boolean (results in `True`)
- `bool(0 + 1j)` # Converts complex number 0 + 1j to boolean (results in `True`)
- `bool(" ")` # Converts a space string to boolean (results in `True` since it's not empty)
- `bool("abc")` # Converts a non-empty string "abc" to boolean (results in `True`)
- `bool("")` # Converts an empty string to boolean (results in `False`)

Note that any non-zero number (integer, float, or complex) and any non-empty string will evaluate to `True`, while zero and empty strings evaluate to `False`.

In [130…
```
bool(-10)
```

Out[130… `True`

In [131…
```
bool(0+1j)
```

Out[131…
```
True
```

In [133…
```
# str(): --- Any type can be converted to a string

print(str(10))     # Convert int to string
print(str(10.50))  # Convert float to string
print(str(True))   # Convert bool to string
print(str(10+20j)) # Convert complex to string
```

```
10
10.5
True
(10+20j)
```

Fundamental data types are those we have covered so far, and we also explored how to perform type casting from one data type to another.

It's important to note that we cannot convert complex data types into integers or floats.

The primary data types we discussed include:

- `int()`
- `float()`
- `complex()`
- `bool()`
- `str()`

Fundamental Datatypes vs. Immutability

- All fundamental datatypes are immutable. What does immutability mean? It means that once we create an object, we are not allowed to perform any changes to that object. In other words, it exhibits non-changeable behavior.
- Why is the concept of immutability important? Consider the following example: although we create only one object with the value of 10, we can assign three different references to that same object.
- The biggest advantage of this approach is improved memory utilization and performance, as the processing virtual machine (PVM) does not want to waste memory.
- While you can create objects with different names, you cannot create multiple objects with the same name.

In [136…
```
x2 = 10
y2 = 10
z2 = 20
print(id(x2))
print(id(y2))
print(id(z2))
```

```
140714119574728
140714119574728
140714119575048
```

- Mutable: Changeable—once you create an object, it can be modified.
- Immutable: Non-changeable—once you create an object, it cannot be modified.
- Fundamental data types in Python are immutable, but lists are mutable.
- Everything in Python is an object.

The concept of reusing the same object applies to the following ranges:

1. int: from 0 to 256
2. bool: always reusable
3. str: always reusable
4. float and complex: cannot be reused in this context.

In [138…
```
x = 10  # Memory address of the variable x
y = 10  # Memory address of the variable y
print(id(x))
print(id(y))
```

```
140714119574728
140714119574728
```

In [139…
```
id(y)
```

Out[139…
```
140714119574728
```

In [141…
```
# 'is' operator
x = 20
```

```
y = 20
print(x is y)
print(y is x)
```

```
True
True
```

In [142…
```
x = True
y = True
z = False
print(x is y)
print(y is z)
print(z is x)
print(z is y)
```

```
True
False
False
False
```

Python Data Structure

LIST DATA STRUCTURE

- Insertion order is preserved and duplicates are allowed.
- If you want to represent a group of values as a single entity, where insertion order is preserved and duplicates are allowed, then you can use the LIST data type.
- What does it mean for insertion order to be preserved?
- You can increase or decrease the size of a list object.
- You can use both the index operator and the slice operator with lists.

In [144…
```
l = []
type(l)
```

Out[144… `list`

In [145…
```
#I want to add some objects to the list. I will append the following values: 10, 20, 30, and another 10.
#Here's how it looks in code:

l.append(10)
l.append(20)
l.append(30)
l.append(10)
```

In [146…
```
l
```

Out[146… `[10, 20, 30, 10]`

In [148…
```
l.add(50)
l
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[148], line 1
----> 1 l.add(50)
      2 l

AttributeError: 'list' object has no attribute 'add'
```

In [149…
```
print(l)
```

```
[10, 20, 30, 10]
```

In [150…
```
# Heterogeneous objects are allowed.
# Java developers have inquired about the availability of null; examples of such types are not included here.
# List objects are expandable. This session serves as an introduction, and we will explore the topic in detail later.

l.append('amx')
l.append(8.0)
l.append(None)
l.append(1 + 2j)
l.append(True)
```

In [152…
```
l
```

Out[152… `[10, 20, 30, 10, 'amx', 8.0, None, (1+2j), True]`

In [153…
```
l.remove('amx') #Delete the object from list
```

In [154…
```
l
```

Out[154… `[10, 20, 30, 10, 8.0, None, (1+2j), True]`

In [155…
```
l[-4]
```
Out[155… 8.0

In [156…
```
l[-7]
```
Out[156… 20

In [157…
```
l
```
Out[157… [10, 20, 30, 10, 8.0, None, (1+2j), True]

In [158…
```
l[4]
```
Out[158… 8.0

In [159…
```
l[6]
```
Out[159… (1+2j)

In [160…
```
l[12]
```
```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[160], line 1
----> 1 l[12]

IndexError: list index out of range
```

In [161…
```
l
```
Out[161… [10, 20, 30, 10, 8.0, None, (1+2j), True]

In [169…
```
l[5]
```

In [167…
```
l[3]
```
Out[167… 10

In [170…
```
l[8]
```
```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[170], line 1
----> 1 l[8]

IndexError: list index out of range
```

In [171…
```
l
```
Out[171… [10, 20, 30, 10, 8.0, None, (1+2j), True]

In [174…
```
l[-1]
```
Out[174… True

In [175…
```
l[-2]
```
Out[175… (1+2j)

Let's introduce the Slicing Operator

In [176…
```
l[:] # ':' Slicing Operator
```
Out[176… [10, 20, 30, 10, 8.0, None, (1+2j), True]

In [177…
```
l
```
Out[177… [10, 20, 30, 10, 8.0, None, (1+2j), True]

In [178…
```
l[0:8]
```
Out[178… [10, 20, 30, 10, 8.0, None, (1+2j), True]

In [179…
```
l[14]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[179], line 1
----> 1 l[14]

IndexError: list index out of range
```

In [180…
```
l
```
Out[180… [10, 20, 30, 10, 8.0, None, (1+2j), True]

In [181…
```
l[3:5]
```
Out[181… [10, 8.0]

In [182…
```
l
```
Out[182… [10, 20, 30, 10, 8.0, None, (1+2j), True]

In [183…
```
l[2:4]
```
Out[183… [30, 10]

In [184…
```
l[0]
```
Out[184… 10

In [185…
```
l[1]
```
Out[185… 20

In [186…
```
l[0] = 100
```
In [187…
```
l
```
Out[187… [100, 20, 30, 10, 8.0, None, (1+2j), True]

In [189…
```
l[1] = 'NIT'
```
In [190…
```
l
```
Out[190… [100, 'NIT', 30, 10, 8.0, None, (1+2j), True]

In [191…
```
l[:]
```
Out[191… [100, 'NIT', 30, 10, 8.0, None, (1+2j), True]

In [192…
```
l[0] = 1000
```
In [193…
```
l
```
Out[193… [1000, 'NIT', 30, 10, 8.0, None, (1+2j), True]

In [194…
```
l[1] = 'AMX'
```
In [195…
```
l[-2] = 'monty'
```
In [196…
```
l
```
Out[196… [1000, 'AMX', 30, 10, 8.0, None, 'monty', True]

In [197…
```
l[:]
```
Out[197… [1000, 'AMX', 30, 10, 8.0, None, 'monty', True]

In [199…
```
l[2]
```
Out[199… 30

In [200…
```
l[2:5]
```
Out[200… [30, 10, 8.0]

In [201…
```
l
```
Out[201… [1000, 'AMX', 30, 10, 8.0, None, 'monty', True]

In [202…
```
l[:4]
```
Out[202… [1000, 'AMX', 30, 10]

In [203…
```
l[2] = 'hi'
```

In [204…
```
l
```

Out[204… `[1000, 'AMX', 'hi', 10, 8.0, None, 'monty', True]`

In [205…
```
l[2:-2]
```

Out[205… `['hi', 10, 8.0, None]`

In [206…
```
l
```

Out[206… `[1000, 'AMX', 'hi', 10, 8.0, None, 'monty', True]`

In [207…
```
l[0:-1]
```

Out[207… `[1000, 'AMX', 'hi', 10, 8.0, None, 'monty']`

In [208…
```
l[:]
```

Out[208… `[1000, 'AMX', 'hi', 10, 8.0, None, 'monty', True]`

In [209…
```
l[-1]
```

Out[209… `True`

In [210…
```
l[-2]
```

Out[210… `'monty'`

In [211…
```
l
```

Out[211… `[1000, 'AMX', 'hi', 10, 8.0, None, 'monty', True]`

In [212…
```
l[:] # This will give us total values which we passed in the object
```

Out[212… `[1000, 'AMX', 'hi', 10, 8.0, None, 'monty', True]`

In [213…
```
l.pop(6)
```

Out[213… `'monty'`

In [214…
```
l
```

Out[214… `[1000, 'AMX', 'hi', 10, 8.0, None, True]`

In [215…
```
del l[-2]
```

In [216…
```
l
```

Out[216… `[1000, 'AMX', 'hi', 10, 8.0, True]`

In [217…
```
l[:-1]
```

Out[217… `[1000, 'AMX', 'hi', 10, 8.0]`

In [218…
```
a, b, c = [1, 'nit', [1, 2, 3,]]
```

In [219…
```
a
```

Out[219… `1`

In [220…
```
b
```

Out[220… `'nit'`

In [221…
```
c
```

Out[221… `[1, 2, 3]`

In [222…
```
a1, b1, c1 = [1, 'nit', [1,2,3]]
```

In [223…
```
a1
```

Out[223… `1`

In [224…
```
b1
```

Out[224… `'nit'`

In [225…
```
c1
```
Out[225…
```
[1, 2, 3]
```
In [226…
```
l2 = [a1, b1, c1]
```
In [227…
```
l2
```
Out[227…
```
[1, 'nit', [1, 2, 3]]
```
In [228…
```
l
```
Out[228…
```
[1000, 'AMX', 'hi', 10, 8.0, True]
```
In [229…
```
l[::-1] #Reverse order
```
Out[229…
```
[True, 8.0, 10, 'hi', 'AMX', 1000]
```
In [230…
```
l
```
Out[230…
```
[1000, 'AMX', 'hi', 10, 8.0, True]
```
In [231…
```
l[::-3]
```
Out[231…
```
[True, 'hi']
```
In [232…
```
l
```
Out[232…
```
[1000, 'AMX', 'hi', 10, 8.0, True]
```
In [234…
```
l[::-1]
```
Out[234…
```
[True, 8.0, 10, 'hi', 'AMX', 1000]
```
In [235…
```
l
```
Out[235…
```
[1000, 'AMX', 'hi', 10, 8.0, True]
```
In [236…
```
l[::-2]
```
Out[236…
```
[True, 10, 'AMX']
```
In [237…
```
l
```
Out[237…
```
[1000, 'AMX', 'hi', 10, 8.0, True]
```
In [238…
```
l[:-2]
```
Out[238…
```
[1000, 'AMX', 'hi', 10]
```
In [239…
```
l
```
Out[239…
```
[1000, 'AMX', 'hi', 10, 8.0, True]
```
In [240…
```
l[-4:]
```
Out[240…
```
['hi', 10, 8.0, True]
```
In [241…
```
l
```
Out[241…
```
[1000, 'AMX', 'hi', 10, 8.0, True]
```
In [242…
```
l.remove(8.0)
```
In [243…
```
l
```
Out[243…
```
[1000, 'AMX', 'hi', 10, True]
```
In [244…
```
l
```
Out[244…
```
[1000, 'AMX', 'hi', 10, True]
```
In [245…
```
l.remove('7')
```
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[245], line 1
----> 1 l.remove('7')

ValueError: list.remove(x): x not in list
```

In [246…
```
l
```
Out[246… [1000, 'AMX', 'hi', 10, True]

In [247…
```
l[-1]
```
Out[247… True

In [248…
```
l
```
Out[248… [1000, 'AMX', 'hi', 10, True]

In [249…
```
l[:-3]
```
Out[249… [1000, 'AMX']

In [250…
```
l
```
Out[250… [1000, 'AMX', 'hi', 10, True]

In [251…
```
l[2]
```
Out[251… 'hi'

In [252…
```
l
```
Out[252… [1000, 'AMX', 'hi', 10, True]

In [253…
```
l[:-1]
```
Out[253… [1000, 'AMX', 'hi', 10]

Final Summary of List Data Types:

1. Order is important.
2. Duplicates are allowed.
3. Heterogeneous data is permitted, meaning different types can be included.
4. Lists are growable in nature; you can delete or insert objects in the list.
5. A list is a data structure.
6. Values are enclosed in square brackets [].
7. Lists are mutable.
8. The concept of indexing is applicable in list data types: index +ve goes from left to right, while negative indexing goes from right to left.
9. Slicing is also available.

- So far, we have covered fundamental data types: int, float, complex, str, and bool.
- We have also discussed type casting functions and how to convert from one type to another.
- Additionally, we have covered one data structure called a list.

Tuple Concept

- A tuple is immutable, while a list is mutable.
- A tuple is represented using parentheses: ().
- Once a tuple is created, it cannot be modified.
- Tuples allow for ordered elements and duplicates.
- They can contain heterogeneous data types, meaning different types of elements are allowed.
- Indexing applies to both tuples and lists, where indexing starts from the left for positive indices and proceeds to the right with negative indices.
- The slicing concept is also applicable to tuples.

In [257…
```
l = [10,20,30,40] #list datastructure
t = (10,20,30,40) #tuple datastructure
```

In [259…
```
type(t)
```
Out[259… tuple

In [260…
```
type(l)
```
Out[260… list

In [261…
```
l
```
Out[261… [10, 20, 30, 40]

In [262…
```
t
```
Out[262… (10, 20, 30, 40)

In [263…
```
l[:]
```
Out[263… [10, 20, 30, 40]

In [264…
```
t[:]
```
Out[264… (10, 20, 30, 40)

In [265…
```
t1 = (10,'amx',True, 5.8, 10)
t1
```
Out[265… (10, 'amx', True, 5.8, 10)

In [266…
```
t1[0]
```
Out[266… 10

In [267…
```
t1[0] = 100
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[267], line 1
----> 1 t1[0] = 100

TypeError: 'tuple' object does not support item assignment
```

In [273…
```
icici = (1234, 'Akash',37, 200000)
type(icici)
```
Out[273… tuple

In [274…
```
icici[1234] = 2345
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[274], line 1
----> 1 icici[1234] = 2345

TypeError: 'tuple' object does not support item assignment
```

In [275…
```
icici.append(45)
```
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[275], line 1
----> 1 icici.append(45)

AttributeError: 'tuple' object has no attribute 'append'
```

In [276…
```
icici
```
Out[276… (1234, 'Akash', 37, 200000)

In [277…
```
icici.remove(1234)
```
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[277], line 1
----> 1 icici.remove(1234)

AttributeError: 'tuple' object has no attribute 'remove'
```

In [278…
```
len(icici)
```
Out[278… 4

In [279…
```
len(t1)
```
Out[279… 5

In [280…
```
t1
```
Out[280… (10, 'amx', True, 5.8, 10)

In [281…
```
t1[0]
```
Out[281… 10

In [282…
```
type(t1)
```

Out[282… tuple

In [283…
```
t1
```

Out[283… (10, 'amx', True, 5.8, 10)

In [284…
```
t1[1] = 20 # tuple immutable (not changeable) e.g - Kyc/Aadhar
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[284], line 1
----> 1 t1[1] = 20

TypeError: 'tuple' object does not support item assignment
```

In [285…
```
t1
```

Out[285… (10, 'amx', True, 5.8, 10)

In [286…
```
t1[0:3]  # [[IF YOU RIGHT SIDE THE CALCULATION WILL (Nth INDEX-1) (3-1) == UPTO 2ND INDEX]]
```

Out[286… (10, 'amx', True)

In [287…
```
t1
```

Out[287… (10, 'amx', True, 5.8, 10)

In [288…
```
t1[0:4]
```

Out[288… (10, 'amx', True, 5.8)

In [289…
```
t1
```

Out[289… (10, 'amx', True, 5.8, 10)

In [290…
```
t
```

Out[290… (10, 20, 30, 40)

In [291…
```
t[0] # Oth index
```

Out[291… 10

In [292…
```
#difference between list and tuple
l
```

Out[292… [10, 20, 30, 40]

In [293…
```
l.append(50)
```

In [294…
```
l
```

Out[294… [10, 20, 30, 40, 50]

In [295…
```
l[0]
```

Out[295… 10

In [296…
```
l[0] = 30 #mutable (change)
```

In [297…
```
l
```

Out[297… [30, 20, 30, 40, 50]

In [298…
```
l.append(60)
```

In [299…
```
l
```

Out[299… [30, 20, 30, 40, 50, 60]

In [300…
```
l[:10]
```

Out[300… [30, 20, 30, 40, 50, 60]

In [301…
```
l[10:]
```

Out[301… []

In [302…
```
type(l)
```

Out[302… list

In [303…
```
l
```
Out[303… [30, 20, 30, 40, 50, 60]

In [304…
```
l[2:]
```
Out[304… [30, 40, 50, 60]

In [305…
```
l
```
Out[305… [30, 20, 30, 40, 50, 60]

In [306…
```
l[:2]
```
Out[306… [30, 20]

In [307…
```
t
```
Out[307… (10, 20, 30, 40)

In [308…
```
# cannot change any value once you decleare cuz tuple is immutable
```

In [309…
```
t[0]= 20
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[309], line 1
----> 1 t[0]= 20

TypeError: 'tuple' object does not support item assignment
```

In [310…
```
t1
```
Out[310… (10, 'amx', True, 5.8, 10)

In [311…
```
t
```
Out[311… (10, 20, 30, 40)

In [312…
```
t.append(50)
```
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[312], line 1
----> 1 t.append(50)

AttributeError: 'tuple' object has no attribute 'append'
```

In [313…
```
t.add(50)
```
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[313], line 1
----> 1 t.add(50)

AttributeError: 'tuple' object has no attribute 'add'
```

In [314…
```
t
```
Out[314… (10, 20, 30, 40)

In [315…
```
t.remove(30)
```
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[315], line 1
----> 1 t.remove(30)

AttributeError: 'tuple' object has no attribute 'remove'
```

In [316…
```
t
```
Out[316… (10, 20, 30, 40)

In [317…
```
t = t*3
t
```
Out[317… (10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40)

In [318…
```
t[0] = 20
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[318], line 1
----> 1 t[0] = 20

TypeError: 'tuple' object does not support item assignment
```

In [319…
```
t1
```
Out[319… (10, 'amx', True, 5.8, 10)

In [320…
```
t2 = t1 * 2 #in this case content has not changed, but the same t content is repeated twice
```

In [321…
```
t2
```
Out[321… (10, 'amx', True, 5.8, 10, 10, 'amx', True, 5.8, 10)

In [322…
```
t3 = (10,20,(2,6))   #is this valid one & u can declare list inside the tuple
```

In [323…
```
t3
```
Out[323… (10, 20, (2, 6))

In [324…
```
type(t3)
```
Out[324… tuple

In [325…
```
colors = "red", "green", "blue"
colors
```
Out[325… ('red', 'green', 'blue')

In [327…
```
colors = "red", "green", "blue"
print(colors)
rev = colors[::-1]
print(rev)
```
```
('red', 'green', 'blue')
('blue', 'green', 'red')
```

In [328…
```
rev = colors[:-1]
rev
```
Out[328… ('red', 'green')

In [329…
```
rev
```
Out[329… ('red', 'green')

In [330…
```
colors
```
Out[330… ('red', 'green', 'blue')

In [331…
```
rev1 = colors[::-2]
rev1
```
Out[331… ('blue', 'red')

In [332…
```
colors = "red", "green", "blue"
print(colors)
rev = colors[:-1]
print(rev)
```
```
('red', 'green', 'blue')
('red', 'green')
```

In [333…
```
colors = "red", "green", "blue"
print(colors)
rev = colors[:-2]
print(rev)
```
```
('red', 'green', 'blue')
('red',)
```

In [334…
```
colors
```
Out[334… ('red', 'green', 'blue')

In [335…
```
colors[-1]
```
Out[335… 'blue'

In [336…
```
colors
```

Out[336… ('red', 'green', 'blue')

In [337…
```
rev = colors[::-1] # reversing order is allowed
rev
```

Out[337… ('blue', 'green', 'red')

In [338…
```
colors
```

Out[338… ('red', 'green', 'blue')

In [339…
```
rev = colors[::-1]
rev
```

Out[339… ('blue', 'green', 'red')

In [340…
```
type(colors)
```

Out[340… tuple

In [341…
```
rev1 = colors[:-1]
rev1
```

Out[341… ('red', 'green')

In [342…
```
rev = colors[::-1] # reversing order is allowed
rev
```

Out[342… ('blue', 'green', 'red')

In [343…
```
colors
```

Out[343… ('red', 'green', 'blue')

In [344…
```
rev2 = colors[::-2] # reversing order is allowed
rev2
```

Out[344… ('blue', 'red')

Range()

In Python, one of the most common data structures is the range() function.

- The range() function generates a sequence of values.
- It is always immutable, meaning that once created, the sequence cannot be modified.
- There are multiple forms of the range() function; let's explore them one by one.

Additionally, it's important to note that while lists preserve their insertion order, sets do not preserve the order of insertion.

In [346…
```
r = range(5) # 1 arguement
r
```

Out[346… range(0, 5)

In [347…
```
r1 = range(10)
r1
```

Out[347… range(0, 10)

In [348…
```
range(10,20)
```

Out[348… range(10, 20)

In [349…
```
r2 = list(range(10)) # we declare 1 argument
print(r2)
r_ = list(range(3))
print(r_)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2]
```

In [350…
```
list(range(5,20))
```

Out[350… [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

In [351…
```
list(range(10,100,10))
```

Out[351… [10, 20, 30, 40, 50, 60, 70, 80, 90]

In [352…
```
list(range(10,100,5))
```

Out[352… [10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]

In [353…
```
list(range(5,20,5,2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[353], line 1
----> 1 list(range(5,20,5,2))

TypeError: range expected at most 3 arguments, got 4
```

In [356…
```
# FORM-1: range(10) -- represents values from 0 to 9 (Python index starts from 0)
r = list(range(10))
print(type(r))
print(r)
```

```
<class 'list'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [357…
```
list(r)
```

Out[357… [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [358…
```
for i in r:
    print('yes')
    print(i)
```

```
yes
0
yes
1
yes
2
yes
3
yes
4
yes
5
yes
6
yes
7
yes
8
yes
9
```

In [359…
```
r
```

Out[359… [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [360…
```
range(10.0, 11.5) # you cannot declare float argument
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[360], line 1
----> 1 range(10.0, 11.5)

TypeError: 'float' object cannot be interpreted as an integer
```

In [363…
```
w1 = list(range(10,20))
w1
```

Out[363… [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

In [362…
```
for i in w1:
    print(i)
```

```
10
11
12
13
14
15
16
17
18
19
```

In [364…
```
r
```

Out[364… [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [365…
```
r[4]
```

Out[365… 4

In [366… `r[0]`

Out[366… 0

In [367… `r[5]`

Out[367… 5

In [368… `r[0:3]`

Out[368… [0, 1, 2]

FORM-1: When you pass only one argument to range(n), it generates numbers from 0 to n-1.

In [1]:
```
r = range(10)  # Generates numbers from 0 to 9
print(list(r)) # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

FORM-2 (if we pass 2 arguments): When you use the range(start, stop) function, it generates a sequence of numbers starting from start up to stop - 1.

In [2]:
```
r = range(10, 20)
print(list(r))  # Output: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

FORM-3 (if we pass 3 arguments): When you use `range(start, stop, step)`, it generates a sequence of numbers starting from `start` up to `stop - 1`, incrementing by `step` each time.

In [3]:
```
r = range(10, 50, 5)
print(list(r))  # Output: [10, 15, 20, 25, 30, 35, 40, 45]
```

```
[10, 15, 20, 25, 30, 35, 40, 45]
```

In [4]:
```
# You cannot declare 4 arguments in range(), because the maximum allowed is 3 (start, stop, step).
# Example of correct usage (FORM-3):
r = range(10, 20, 5)  # start=10, stop=20, step=5
print(list(r))

# If you try to use 4 arguments, Python will raise a TypeError:
# range(10, 20, 5, 6)  # TypeError: range expected at most 3 arguments, got 4
```

```
[10, 15]
```

In [5]:
```
range(10, 20, 5, 6)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 range(10, 20, 5, 6)

TypeError: range expected at most 3 arguments, got 4
```

Set Datatype

Difference between List & Set:

- Use a **List** (`[]`) when you need to represent a group of objects as a single entity, and **duplicates are allowed** and **order is important**.
- Use a **Set** (`{}`) when you do **not want duplicates** and **order is not important**.

Key Points:

- **List:** Ordered, allows duplicates, supports indexing and slicing, mutable, uses `append()` to add elements.
- **Set:** Unordered, does not allow duplicates, does **not** support indexing or slicing, mutable, uses `add()` and `remove()` to modify elements.
- **Set objects:** Insertion order is not preserved, index concept is not allowed, heterogeneous objects are allowed.

In [6]:
```
s = {10,20,30,10,20,30} # is it allowed or not, we will check
print(s)  # Output: {10, 20, 30}
```

```
{10, 20, 30}
```

Explanation:

Yes, this is allowed. When you create a set in Python, any duplicate values are automatically removed. So, s will only contain {10, 20, 30}. Sets do not allow duplicates and do not preserve order.

In [7]:
```
s_ = {56,30,75,109 }
```

```
s_
```

Out[7]: {30, 56, 75, 109}

In [8]:
```
s1 = {30,10,20,10,'abc',5.0,True}
s1
```

Out[8]: {10, 20, 30, 5.0, True, 'abc'}

In [9]:
```
s1[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 s1[0]

TypeError: 'set' object is not subscriptable
```

In [10]:
```
s1[1:4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 s1[1:4]

TypeError: 'set' object is not subscriptable
```

In [11]:
```
s
```

Out[11]: {10, 20, 30}

In [13]:
```
s[:]   #set object does not support indexing, slicing, or subscripting
# This will raise: TypeError: 'set' object is not subscriptable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 s[:]

TypeError: 'set' object is not subscriptable
```

In [14]:
```
s
```

Out[14]: {10, 20, 30}

In [15]:
```
s[1:]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 s[1:]

TypeError: 'set' object is not subscriptable
```

In [16]:
```
s
```

Out[16]: {10, 20, 30}

In [17]:
```
s.append(True)  #mutable
s
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[17], line 1
----> 1 s.append(True)  #mutable
      2 s

AttributeError: 'set' object has no attribute 'append'
```

In [18]:
```
s.add(True)  #mutable
s
```

Out[18]: {True, 10, 20, 30}

In [19]:
```
s.add(300)
s
```

Out[19]: {True, 10, 20, 30, 300}

In [20]:
```
s.add('b')
s
```

Out[20]: {10, 20, 30, 300, True, 'b'}

In [22]:
```
s.add('c')
```

In [23]:
```
s
```

Out[23]:
```
{10, 20, 30, 300, True, 'b', 'c'}
```

In [24]:
```
s.add('c','d','d')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 1
----> 1 s.add('c','d','d')

TypeError: set.add() takes exactly one argument (3 given)
```

In [25]:
```
s
```

Out[25]:
```
{10, 20, 30, 300, True, 'b', 'c'}
```

In [26]:
```
s[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[26], line 1
----> 1 s[1]

TypeError: 'set' object is not subscriptable
```

In [28]:
```
s.remove(300) #remove function is applicable for lists, not for sets

# The remove() method **is** applicable for sets in Python, but it works differently than with lists.
# For sets, remove() deletes the specified element. If the element is not present, it raises a KeyError.
# For lists, remove() deletes the first occurrence of the specified value.
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[28], line 1
----> 1 s.remove(300)

KeyError: 300
```

In [29]:
```
s.remove(300)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[29], line 1
----> 1 s.remove(300)

KeyError: 300
```

In [30]:
```
s.add('d')
s
```

Out[30]:
```
{10, 20, 30, True, 'b', 'c', 'd'}
```

In [33]:
```
s3 = {[10,20,30], 40, True}
s3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 1
----> 1 s3 = {[10,20,30], 40, True}
      2 s3

TypeError: unhashable type: 'list'
```

In [34]:
```
# Sets in Python can only contain hashable (immutable) elements.
# Lists are not hashable, so you cannot put a list inside a set.
```

In [35]:
```
s1
```

Out[35]:
```
{10, 20, 30, 5.0, True, 'abc'}
```

In [36]:
```
s[0] = 50
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[36], line 1
----> 1 s[0] = 50

TypeError: 'set' object does not support item assignment
```

In [37]:
```
myset = {'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight'}

for i in myset:
    print(i)
```

```
three
seven
one
eight
six
four
five
two
```

In [38]:
```
myset.add('nine')
```

In [39]:
```
myset
```

Out[39]:
```
{'eight', 'five', 'four', 'nine', 'one', 'seven', 'six', 'three', 'two'}
```

In [40]:
```
for i in enumerate(myset):
    print(i)
```

```
(0, 'three')
(1, 'seven')
(2, 'one')
(3, 'eight')
(4, 'six')
(5, 'four')
(6, 'nine')
(7, 'five')
(8, 'two')
```

In [41]:
```
#enumerate(myset) returns pairs of (index, value) for each element in myset.
```

In [42]:
```
'nine' in myset
```

Out[42]:
```
True
```

In [43]:
```
myset.add([10,20])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[43], line 1
----> 1 myset.add([10,20])

TypeError: unhashable type: 'list'
```

In [44]:
```
printmyset.update([10,20])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[44], line 1
----> 1 printmyset.update([10,20])

NameError: name 'printmyset' is not defined
```

In [45]:
```
myset.add((30,40,50,50))
myset.update(('ab', 56,[1,2,3]))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[45], line 2
      1 myset.add((30,40,50,50))
----> 2 myset.update(('ab', 56,[1,2,3]))

TypeError: unhashable type: 'list'
```

In [46]:
```
myset
```

Out[46]:
```
{(30, 40, 50, 50),
 56,
 'ab',
 'eight',
 'five',
 'four',
 'nine',
 'one',
 'seven',
 'six',
 'three',
 'two'}
```

In [47]:
```
myset.discard('nine')
```

In [48]:
```
myset
```

Out[48]:
```
{(30, 40, 50, 50),
 56,
 'ab',
 'eight',
 'five',
 'four',
 'one',
 'seven',
 'six',
 'three',
 'two'}
```

Set operations in Python allow you to perform mathematical set operations like union, intersection, difference, and symmetric difference.

In [49]:
```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# Union: combines all elements from both sets
print(A | B)           # {1, 2, 3, 4, 5, 6, 7, 8}
print(A.union(B))      # {1, 2, 3, 4, 5, 6, 7, 8}

# Intersection: elements common to both sets
print(A & B)           # {4, 5}
print(A.intersection(B)) # {4, 5}

# Difference: elements in A but not in B
print(A - B)           # {1, 2, 3}
print(A.difference(B)) # {1, 2, 3}

# Symmetric Difference: elements in either set, but not both
print(A ^ B)           # {1, 2, 3, 6, 7, 8}
print(A.symmetric_difference(B)) # {1, 2, 3, 6, 7, 8}
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 4, 5, 6, 7, 8}
{4, 5}
{4, 5}
{1, 2, 3}
{1, 2, 3}
{1, 2, 3, 6, 7, 8}
{1, 2, 3, 6, 7, 8}
```

In [50]:
```
A = {1,2,3,4,5}
B = {4,5,6,7,8}
C = {8,9,10}
```

In [51]:
```
A | B
```

Out[51]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

In [52]:
```
A.union(B)
```

Out[52]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

In [53]:
```
A.union(C)
```

Out[53]:
```
{1, 2, 3, 4, 5, 8, 9, 10}
```

In [54]:
```
A.union(C)
```

Out[54]:
```
{1, 2, 3, 4, 5, 8, 9, 10}
```

In [55]:
```
A | C | B
```

Out[55]:
```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

In [56]:
```
A, B, C
```

Out[56]:
```
({1, 2, 3, 4, 5}, {4, 5, 6, 7, 8}, {8, 9, 10})
```

In [57]:
```
A & B
```

Out[57]:
```
{4, 5}
```

In [59]:
```
A & C
```

Out[59]:
```
set()
```

In [60]:
```
A = {1,2,3,4,5}
C = {8,9,10}
A & C  # Intersection: returns elements common to both sets
# Output: set()
```

Out[60]:
```
set()
```

In [61]:
```
B & C
```

Out[61]:
```
{8}
```

In [62]:
```
A.intersection(B)
```

Out[62]:
```
{4, 5}
```

In [65]:
```
A.intersection_update(B) ## This will update set A to contain only the elements found in both A and B.
```

In [67]:
```
A.intersection(B)
```

Out[67]:
```
{4, 5}
```

In [68]:
```
A
```

Out[68]:
```
{4, 5}
```

In [69]:
```
B
```

Out[69]:
```
{4, 5, 6, 7, 8}
```

In [70]:
```
D = {10, 11, 12, 13, 14, 15}
len(D)
```

Out[70]:
```
6
```

In [71]:
```
A - B
```

Out[71]:
```
set()
```

In [72]:
```
len(D)
```

Out[72]:
```
6
```

In [73]:
```
A | D
```

Out[73]:
```
{4, 5, 10, 11, 12, 13, 14, 15}
```

In [74]:
```
A & D
```

Out[74]:
```
set()
```

In [75]:
```
A - D
```

Out[75]:
```
{4, 5}
```

In [76]:
```
C | A
```

Out[76]:
```
{4, 5, 8, 9, 10}
```

In [77]:
```
myset
```

Out[77]:
```
{(30, 40, 50, 50),
 56,
 'ab',
 'eight',
 'five',
 'four',
 'one',
 'seven',
 'six',
 'three',
 'two'}
```

In [78]:
```
myset.discard('eleven')
```

In [79]:
```
myset
```

Out[79]:
```
{(30, 40, 50, 50),
 56,
 'ab',
 'eight',
 'five',
 'four',
 'one',
 'seven',
 'six',
 'three',
 'two'}
```

In [80]:
```
myset.remove('eleven')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[80], line 1
----> 1 myset.remove('eleven')

KeyError: 'eleven'
```

In [81]:
```
myset1 = myset.copy()
```

In [82]:
```
myset1
```

Out[82]:
```
{(30, 40, 50, 50),
 56,
 'ab',
 'eight',
 'five',
 'four',
 'one',
 'seven',
 'six',
 'three',
 'two'}
```

In [83]:
```
myset
```

Out[83]:
```
{(30, 40, 50, 50),
 56,
 'ab',
 'eight',
 'five',
 'four',
 'one',
 'seven',
 'six',
 'three',
 'two'}
```

In [84]:
```
for i in enumerate(myset):
    print(i)
```

```
(0, 'three')
(1, 'seven')
(2, 'one')
(3, 'eight')
(4, 'ab')
(5, 'six')
(6, 'four')
(7, 56)
(8, (30, 40, 50, 50))
(9, 'five')
(10, 'two')
```

In [85]:
```
'nine' in myset
```

Out[85]:
```
False
```

In [86]:
```
myset.add([10,20])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[86], line 1
----> 1 myset.add([10,20])

TypeError: unhashable type: 'list'
```

In [87]:
```
print(myset.update([10,20])) # .update() adds elements to the set. It does not return anything, so printing its result will sh
```

```
None
```

In [89]:
```
print(myset.add((30,40,50,50))) # Adds the tuple as a single element to the set

#The following line will raise a TypeError because [1,2,3] is a list (unhashable type)
print(myset.update(('ab', 56,[1,2,3]))) # To avoid the error, only use hashable (immutable) types in update: (1,2,3) is a tupl
```

```
None
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[89], line 4
      1 print(myset.add((30,40,50,50))) # Adds the tuple as a single element to the set
      3 #The following line will raise a TypeError because [1,2,3] is a list (unhashable type)
----> 4 print(myset.update(('ab', 56,[1,2,3])))

TypeError: unhashable type: 'list'
```

In [90]:
```
#.add() adds a single element (here, a tuple).
#.update() adds each element from the iterable to the set.
```

Python has several built-in data structures:

- List: Ordered, mutable, allows duplicates. Example: [1, 2, 3]
- Tuple: Ordered, immutable, allows duplicates. Example: (1, 2, 3)
- Set: Unordered, mutable, no duplicates. Example: {1, 2, 3}
- Frozenset: Unordered, immutable set. Example: frozenset([1, 2, 3])
- Dict (Dictionary): Key-value pairs, mutable, keys must be unique. Example: {'a': 1, 'b': 2}
- Range: Immutable sequence of numbers. Example: range(5)

In [91]:
```
myset
```

Out[91]:
```
{(30, 40, 50, 50),
 10,
 20,
 56,
 'ab',
 'eight',
 'five',
 'four',
 'one',
 'seven',
 'six',
 'three',
 'two'}
```

In [92]:
```
myset.discard('nine')
```

In [93]:
```
myset
```

Out[93]:
```
{(30, 40, 50, 50),
 10,
 20,
 56,
 'ab',
 'eight',
 'five',
 'four',
 'one',
 'seven',
 'six',
 'three',
 'two'}
```

In [94]:
```
#### frozenset is the same as set, but it is immutable (cannot be changed after creation).
fs = frozenset([1, 2, 3, 2, 1])
print(fs)  # Output: frozenset({1, 2, 3})

# You cannot add or remove elements from a frozenset.
# fs.add(4)      # This will raise an AttributeError
# fs.remove(1)   # This will raise an AttributeError
```

```
frozenset({1, 2, 3})
```

In [95]:
```
fs.add(4)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[95], line 1
----> 1 fs.add(4)

AttributeError: 'frozenset' object has no attribute 'add'
```

In [96]:
```
fs.remove(1)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[96], line 1
----> 1 fs.remove(1)

AttributeError: 'frozenset' object has no attribute 'remove'
```

In [97]:
```
myset = {'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight'}
for a in myset:
    print(a)
```

```
three
seven
one
eight
six
four
five
two
```

DICTIONARY DATATYPES (`dict`)

- A dictionary in Python stores data as key-value pairs, similar to how words and meanings are stored in an Oxford dictionary.
- In lists, tuples, sets, range, and frozenset, each element is an individual object. In a dictionary, objects are grouped as pairs (e.g., `rollno:name`, `fruit:price`, `mobileno:name`).
- Dictionaries are very important and unique compared to other data types.
- **Duplicate keys are not allowed**, but values can be duplicated.
- Dictionaries are represented using curly braces `{}` and key-value pairs separated by colons (`:`).
- Both keys and values can be heterogeneous (any data type).
- There is no restriction that all keys must be integers or all values must be strings.
- Keys and values can be any type of object.

In [98]:
```python
# Creating a dictionary
student = {
    101: "Alice",
    102: "Bob",
    103: "Charlie",
    104: "Alice"  # Duplicate value is allowed
}
print(student)
```

```
{101: 'Alice', 102: 'Bob', 103: 'Charlie', 104: 'Alice'}
```

In [99]:
```python
type(student)
```

Out[99]: dict

In [100…
```python
student = {}
type(student)
```

Out[100… dict

In [132…
```python
# Functions and methods commonly used with Python dictionaries:

d = {100: 'amx', 200: 'shiv', 300: 'nar'}
```

In [133…
```python
# 1. Get all keys
print(d.keys())  # dict_keys([100, 200, 300])
```

```
dict_keys([100, 200, 300])
```

In [134…
```python
# 2. Get all values
print(d.values())  # dict_values(['amx', 'shiv', 'nar'])
```

```
dict_values(['amx', 'shiv', 'nar'])
```

In [135…
```python
# 3. Get all key-value pairs
print(d.items())  # dict_items([(100, 'amx'), (200, 'shiv'), (300, 'nar')])
```

```
dict_items([(100, 'amx'), (200, 'shiv'), (300, 'nar')])
```

In [144…
```python
# 4. Get value for a key (returns None if key not found)
print(d.get(100))  # 'amx'
```

```
None
```

In [137…
```python
# 5. Add or update a key-value pair
d[400] = 'new'
print(d)
```

```
{100: 'amx', 200: 'shiv', 300: 'nar', 400: 'new'}
```

In [ ]:

In [138…
```python
# 6. Remove a key-value pair by key
d.pop(200)
print(d)
```

```
{100: 'amx', 300: 'nar', 400: 'new'}
```

In [139…
```python
# 6. Remove a key-value pair by value
d.pop('shiv')
print(d)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[139], line 2
      1 # 6. Remove a key-value pair by value
----> 2 d.pop('shiv')
      3 print(d)

KeyError: 'shiv'
```

In [140…
```
# 7. Remove all items
d.clear()
print(d)
```

```
{}
```

In [141…
```
# 8. Copy dictionary
d2 = {1: 'a', 2: 'b'}
d3 = d2.copy()
print(d3)
```

```
{1: 'a', 2: 'b'}
```

In [142…
```
# 9. Check if key exists
print(1 in d2)  # True
```

```
True
```

In [143…
```
# 10. Delete a key-value pair using del
del d2[1]
print(d2)
```

```
{2: 'b'}
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: