UNIT-1: Basics – Data Types, Operators

Data Types, Escape Sequences, Variables and Basic Input/Output; Assignment Statements, Operators; Arithmetic Expressions, Operator precedence, Type Casting, Program Comments and Docstrings; Program Format and Structure, REPL, IDLE, Running a Script from a Terminal Command Prompt;

1.0 Introduction to Python Programming:

1.0.1 What is Python?

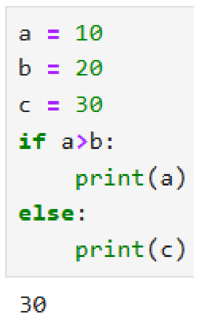
- It is a Programming Language.
- We can develop applications by using this programming language.
- We can say that, Python is a High-Level Programming Language. Immediately you may get doubt that,

1.0.2 What is the meaning of High-Level Programming Language?

High-Level means Programmer friendly Programming Language. This means we are not required to worry about Low-level things (i.e., Memory management, security, destroying the objects and so on).

- By simply seeing the code programmer can understand the program. We can write the code very easily.
- High level languages are Programmer friendly languages but not machine friendly languages.

Example:



```
a = 10
b = 20
c = 30
if a>b:
    print(a)
else:
    print(c)
```

```
30
```

Examples of High-Level Programming Languages:

- C
- C++
- Java
- C#
- Python

1.0.3 Python is a General-Purpose High Level Programming Language.

Here, **General Purpose** means Python is not specific to a particular area; happily, we can use Python for any type of application areas. For example,

- Desktop Applications
- Web Applications
- Data Science Applications
- Machine learning applications and so on.

1.0.4 Who Developed Python?

- **Guido Van Rossum** developed Python language while working in National Research Institute (NRI) in Netherland.

1.0.5 When he developed this Language?

Most of the people may think that so far we are heard about Java, C and C++ and we are recently knowing about python (especially in our INDIA) and they may assume that Python is a new programming language and Java is a old programming language.

- Java came in 1995 and officially released in 1996.
- Python came in 1989; this means that Python is older programming language than Java. Even it is developed in 1989, but it is not released to the public immediately.
- In 1991, Python made available to the public. Officially Python released into the market on 21-02-1991 (i.e., First version).

Then, immediately you may have a doubt that, **Why Python suddenly (in 2019) became Popular?**

- Generally, Market requirements are keeps on changing from time to time.
- Current market situation is, everyone talks about
    - I need Simple Language (i.e., Easy to understandable)
    - I have to write very less (or) concise code to fulfill my requirement.
    - In these days, everyone talks about AI, Machine Learning, Deep Learning, Neural networks, Data Science, IOT. For these trending requirements, best suitable programming language is Python.

That's why in these days, Python becomes more popular programming language.

1.0.6 Easiness of Python compared to other programming languages

In Python

```
In [1]:

print("Hello World");

Hello World
```

```
In [3]:

print("Hello World")          # ';' is also optional

Hello World
```

Just to print 'Hello World',

- C language takes 5 lines of code
- Java takes 7 lines of code

When compared with any other programming language (C, C++, C## or Java), the easiest programming language is Python.

Key Points:

1. Python is a general-purpose high-level programming language.
2. Python was developed by Guido Van Rossum in 1989, while working at National Research Institute at Netherlands.
3. Officially Python was made available to public in 1991. The official Date of Birth for Python is: Feb 20th 1991.
4. Python is recommended as first programming language for beginners.
5. Python is an Example of Dynamically typed programming language

1.0.7 Features of Python:

1. Simple and easy to learn
2. Freeware and Open Source
3. High Level Programming Language
4. Platform Independent
5. Portability
6. Dynamically Typed
7. Python is both Procedure oriented and Object oriented
8. Interpreted
9. Extensible
10. Embedded
11. Extensive Library

Summary:

- These are the 11 key features of Python programming language.
- Among various features of Python discussed above, the following 3 features are specific to Python
    a. Dynamically Typed
    b. Both Procedural Oriented and Object Oriented
    c. Extensive Library

These 3 Features aren't supported by any other programming languages like C, C++ & Java etc.,

1.0.8 Reserved Words (Or) Keywords in Python:

- In Python some words are reserved to represent some meaning or functionality. Such types of words are called **reserved words**.
- There are **33 reserved words** available in Python.
    True, False, None
    and, or, not, is
    if, elif, else
    while, for, break, continue, return, in, yield
    try, except, finally, raise, assert
    import, from, as, class, def, pass, global, nonlocal, lambda, del, with
- All 33 keywords contain only alphabets symbols.
- Except the following 3 reserved words, all contain only lower case alphabet symbols.
    True
    False
    None

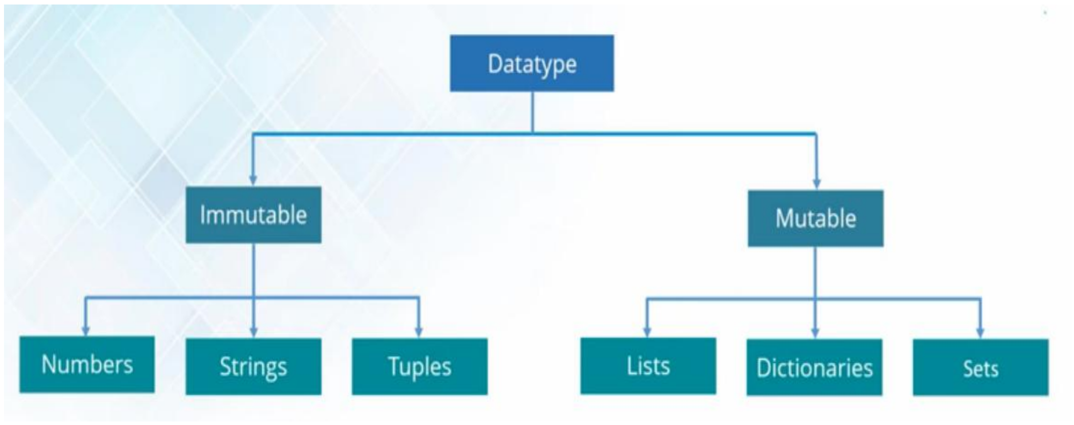
1.1 Data Types in Python:

- Data Type represents the type of data present inside a variable.
- In Python we are not required to specify the type explicitly. Based on value provided, the type will be assigned automatically. Hence Python is **Dynamically Typed Language**.
- A **literal** is the way a value of a data type looks to a programmer. The programmer can use a literal in a program to mention a data value. When the Python interpreter evaluates a literal, the value it returns is simply that literal.

The following shows example literals of several Python data types.

Type of Data	**Python Type Name**	**Example Literals**
Integers	int	–1, 0, 1, 2
Real numbers	float	–0.55, .3333, 3.14, 6.0
Character Strings	str	"Hi", "", 'A', "66"

Python provides built-in data types. Every value in Python has a data type. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.



Data type is classified into Immutable and Mutable. In general, data types in Python can be distinguished based on whether objects of the type are mutable or immutable.

1) Immutable:

The content of objects of immutable types cannot be changed after they are created.

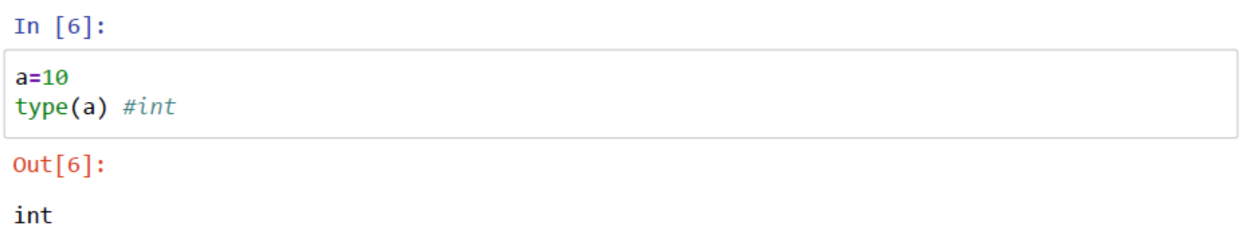
Example:

a) int, float, long, complex
b) str
c) bytes
d) tuple
e) frozen set

a. Integer type

- We can use 'int' data type to represent whole numbers (integral values).

Example:



```
In [6]:
a=10
type(a) #int
Out[6]:
int
```

By using an in-built function **type()**, we can find the type of any variable.

```
In [9]:
```

```python
a = 10
print(type(a))b
```

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

b. Float type

- We can use float data type to represent floating point values (Number with decimal values)

Example:

```
In [12]:
```

```python
f=1.234
type(f)
```

```
Out[12]:

float
```

We can also represent floating point values by using exponential form (scientific notation) as shown in the given example,

```
In [17]:
```

```python
f = 1.2e3
print(f)
```

```
1200.0
```

c. Boolean type

- We can use this data type to represent boolean values.
- The only allowed values for this data type are: True and False (true & false are not allowed in Python)
- Internally Python represents True as 1 and False as 0

Example:

```
In [13]:
```

```python
a = True
print(type(a))
```

```
<class 'bool'>
```

```
In [42]:

a = 10
b = 20
c = a>b
print(c)
type(c)

False

Out[42]:

bool
```

```
In [43]:

print(True + True)

2
```

```
In [44]:

print(True - False)

1
```

d. String Literals: str data type representations by using **single, double and triple quotes**.

- In Python, a string literal is a sequence of characters enclosed within single quotes or double quotes.
- str represents String data type. It is the most commonly used data type in Python.
- In Python to represent a string, can use either single quotes ("), double quotes ("") or triple quotes.

Example1:

```
x = 'welcome'                     #in single quote
y = "Welcome"                     #in double quotes
z = '''how
            are
                  you'''          #multi-line string
print(x)
print(y)
print(z)
```

```
welcome
Welcome
how
            are
                  you
```

Example2:

```
In [46]:

s = 'Karthi'
print(type(s))

<class 'str'>

In [47]:

s = "Karthi"
print(type(s))

<class 'str'>

In [48]:

s = 'a'
print(type(s))    # in Python there is no 'char'data type

<class 'str'>
```

The two string literals (' ' and " ") represent the empty string. Although it contains no characters, the empty string is a string nonetheless. Note that the empty string is different from a string that contains a single blank space character, " ".

In Python, we can use triple quotes also in the following cases.

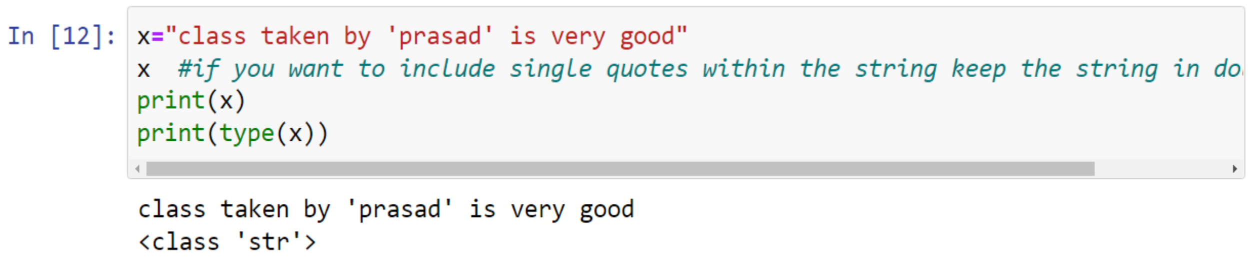
1. By using single quotes or double quotes we cannot represent multi line string literals.

- For this requirement we should go for triple single quotes(''') or triple double quotes(""").

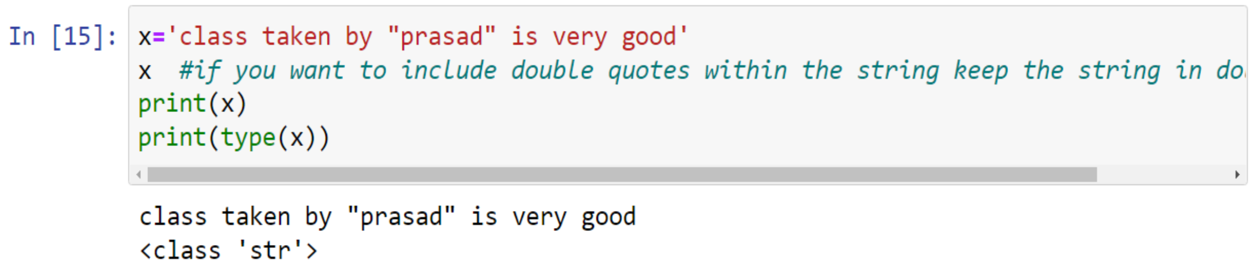
Example:

```
In [54]:

s = '''Karthi
sahasra'''
print(s)

Karthi
sahasra

In [57]:

s = """Karthi
sahasra"""
print(s)

Karthi
sahasra
```

2. We can also use triple quotes, to use single quotes or double quotes as normal characters in our String.

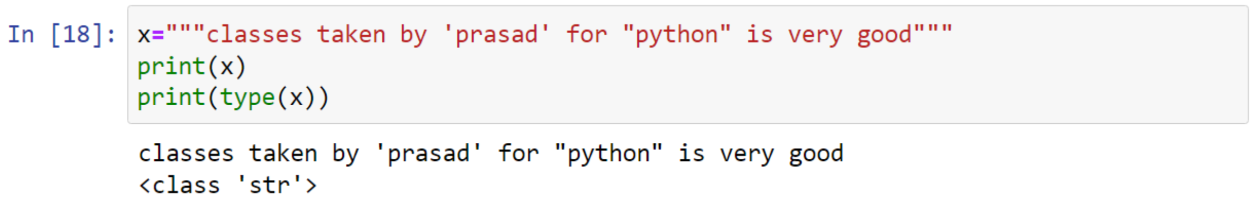


```
In [12]: x="class taken by 'prasad' is very good"
         x  #if you want to include single quotes within the string keep the string in do
         print(x)
         print(type(x))

         class taken by 'prasad' is very good
         <class 'str'>
```



```
In [15]: x='class taken by "prasad" is very good'
         x  #if you want to include double quotes within the string keep the string in do
         print(x)
         print(type(x))

         class taken by "prasad" is very good
         <class 'str'>
```

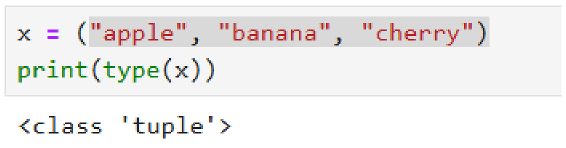
Now, if we want to use both single quotes and double quotes as the normal characters in the string, then you need to enclose the string in triple quotes.



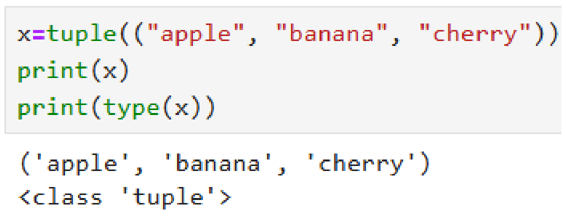
```
In [18]: x="""classes taken by 'prasad' for "python" is very good"""
         print(x)
         print(type(x))

         classes taken by 'prasad' for "python" is very good
         <class 'str'>
```

e. Tuple:

- A tuple is a sequence of immutable objects; therefore, tuple cannot be changed. The objects are enclosed within parenthesis and separated by comma.
- **Example:**

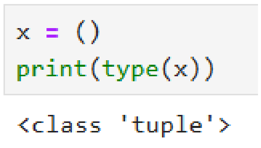


```
x = ("apple", "banana", "cherry")
print(type(x))

<class 'tuple'>
```



```
x=tuple(("apple", "banana", "cherry"))
print(x)
print(type(x))

('apple', 'banana', 'cherry')
<class 'tuple'>
```

- The empty tuple is written as two parentheses containing nothing –
- **Example:**

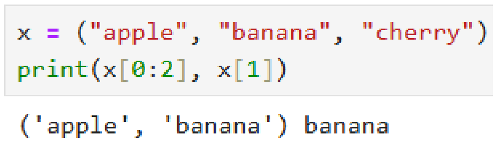


```
x = ()
print(type(x))
```

```
<class 'tuple'>
```

Accessing Values in Tuples:

- To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index.



```
x = ("apple", "banana", "cherry")
print(x[0:2], x[1])
```

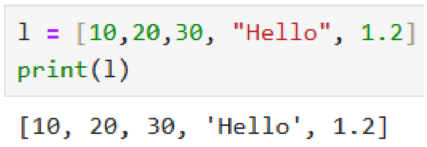
```
('apple', 'banana') banana
```

2) Mutable:

- Only mutable objects support methods that change the object in place, such as reassignment of a sequence slice, which will work for lists, but raise an error for tuples and strings.
- These are of type list, dict, set. Custom classes are generally mutable.

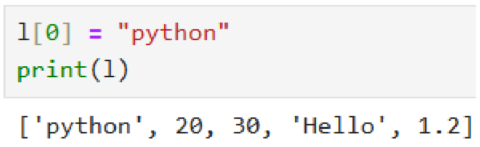
a. List:

- List is mutable, that means, in existing object only you can perform any modifications. This changeable behavior is known as Mutability.
- A list can be created by putting the value inside the square bracket and separated by comma. Important thing about a list is that items in a list need not be of the same type.
- **Syntax:**
  list_name = [value$_1$, value$_2$, value$_3$, ……,value$_n$];
- **For accessing list:**
  Listname[index]
- **Example:**



```
l = [10,20,30, "Hello", 1.2]
print(l)
```

```
[10, 20, 30, 'Hello', 1.2]
```

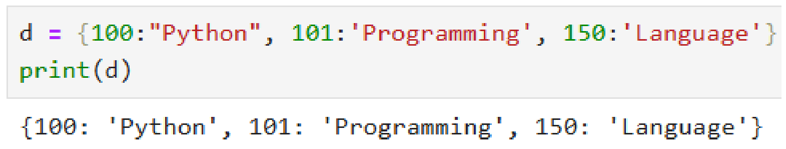


```
l[0] = "python"
print(l)
```

```
['python', 20, 30, 'Hello', 1.2]
```

b) Dictionary:

- Dictionary is an unordered set of key and value pair. It is an container that contains data, enclosed within curly braces.
- The pair i.e., key and value is known as item.
- The key passed in the item must be unique. The key and the value are separated by a colon (:). This pair is known as item.
- Items are separated from each other by a comma (,). Different items are enclosed within a curly brace and this forms Dictionary.
- **Example:**

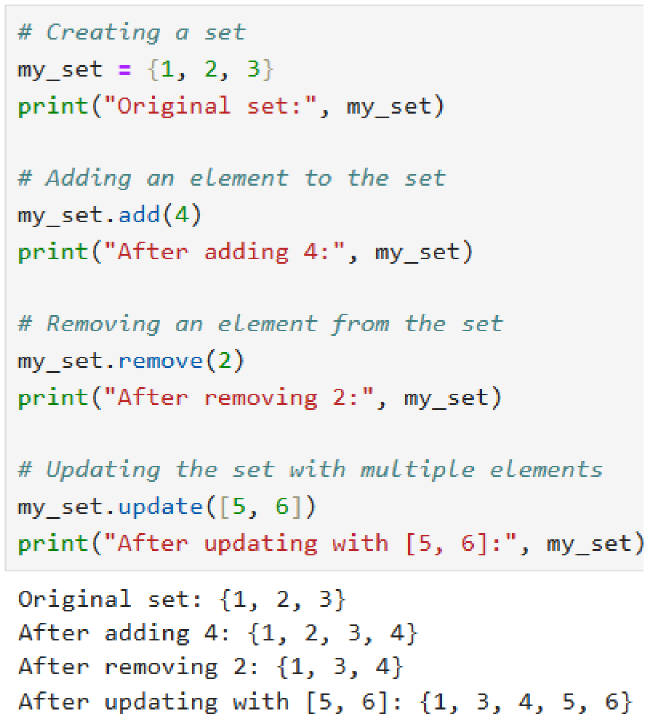


```
d = {100:"Python", 101:'Programming', 150:'Language'}
print(d)
```

```
{100: 'Python', 101: 'Programming', 150: 'Language'}
```

c) Sets:

- Sets are mutable.
- This means that the contents of a set can be changed after the set has been created.
- You can add, remove, and update elements in a set.
- **Example:**



```
# Creating a set
my_set = {1, 2, 3}
print("Original set:", my_set)

# Adding an element to the set
my_set.add(4)
print("After adding 4:", my_set)

# Removing an element from the set
my_set.remove(2)
print("After removing 2:", my_set)

# Updating the set with multiple elements
my_set.update([5, 6])
print("After updating with [5, 6]:", my_set)
```

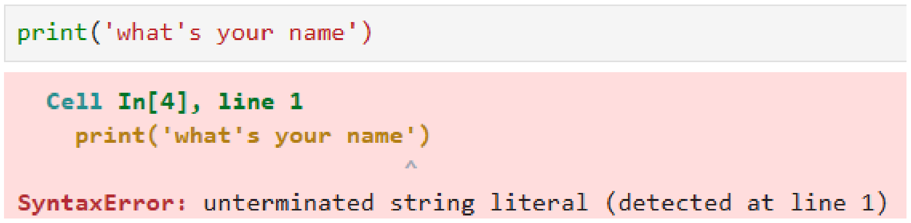
```
Original set: {1, 2, 3}
After adding 4: {1, 2, 3, 4}
After removing 2: {1, 3, 4}
After updating with [5, 6]: {1, 3, 4, 5, 6}
```

1.2 Escape Sequences:

1.2.1 Introduction to Escape Sequence: Some characters (like”, \) cannot be directly included in a string. Such characters must be escaped by placing a backslash before them.

Example:

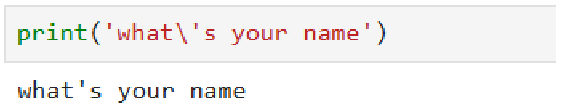
What will happen if you try to print **what’s your name.**



```
print('what's your name')
```

```
  Cell In[4], line 1
    print('what's your name')
                           ^
SyntaxError: unterminated string literal (detected at line 1)
```

To clearly specify that this single quote does not indicate the end of the string. This indication can be given with the help of an escape sequence. You can specify the single quote as \’ (single quote preceded by a backslash).



```
print('what\'s your name')
```

```
what's your name
```

1.2.2 Definition: An **escape sequence** is a combination of characters that is translated into another character or a sequence of characters that may be difficult or impossible to represent directly.

Escape Sequence	Meaning
\n	Newline
\b	Backspace
\f	Form feed
\t	Horizontal tab
\r	Carriage Return
\\	The \ character
\’	Single quotation mark
\”	Double quotation mark

Table2: Some escape sequences in Python

Because the backslash is used for escape sequences, it must be escaped to appear as a literal character in a string. Thus, print('\\') would display a single \ character.

Example1:

```
print("Maharaj Vijayaram Gajapathi Raj College of Engineering(Autonomous)")
print("Maharaj Vijayaram Gajapathi Raj College of Engineering\t(Autonomous)")
print("Maharaj Vijayaram Gajapathi Raj College of Engineering\n(Autonomous)")
```

```
Maharaj Vijayaram Gajapathi Raj College of Engineering(Autonomous)
Maharaj Vijayaram Gajapathi Raj College of Engineering  (Autonomous)
Maharaj Vijayaram Gajapathi Raj College of Engineering
(Autonomous)
```

Example2:

```
print('This is \' symbol')
print('This is \" symbol')
print('This is \\ symbol')
```

```
This is ' symbol
This is " symbol
This is \ symbol
```

Example3:

```
print("mvgr\bcollege")
```

```
mvgcollege
```

In this example, the \b character moves the cursor back one space, overwriting the 'r' with 'c'.

1.3 Variables and Basic Input/Output:

1.3.1 Variables:

Variables in Python are used to store data that can be referenced and manipulated in a program. They act as labels for the data stored in the memory and can hold different types of values, such as numbers, strings, lists, and more.

Characteristics of Python Variables:

1. **Dynamic Typing:** In Python, you don't need to declare a variable's type. The type is inferred based on the value assigned to the variable.
2. **Case-Sensitive:** Variable names are case-sensitive. For example, myVar and myvar are two different variables.
3. **Naming Rules:**
   a) Must start with a letter (a-z, A-Z) or an underscore (_).
   b) Can be followed by letters, digits (0-9), or underscores.

c) Cannot use Python reserved keywords (e.g., if, for, while).

Example of Variable Assignment:

- **# Assigning integer value to a variable**
  x = 10
- **# Assigning floating-point value to a variable**
  pi = 3.14
- **# Assigning string value to a variable**
  name = "Alice"
- **# Assigning boolean value to a variable**
  is_active = True
- **# Assigning list value to a variable**
  numbers = [1, 2, 3, 4, 5]

Example of Multiple Assignments:

- You can assign values to multiple variables in a single statement.

  a, b, c = 1, 2, 3

1.3.2 Basic Input/Output:

In Python, input and output (I/O) are essential for interacting with users and other systems.

1. **Input:**
   - **Using input() Function:** The input() function is used to take input from the user. By default, it takes the input as a string.
   - **Example 1:** Basic Input

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

```
Enter your name:  K. Leela Prasad
Hello, K. Leela Prasad!
```

   - **Example 2:** Input with Type Conversion

```
age = int(input("Enter your age: "))
print("You are " + str(age) + " years old.")
```

```
Enter your age:  33
You are 33 years old.
```

   - In this example, the input is converted from a string to an integer using the int() function.

2. **Output:**
   - **Using print() Function:** The print() function is used to output data to the standard output, typically the console.
   - **Example 1:** Basic Output

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

```
Hello, World!
```

   - **Example 2:** Output with Variables

```
name = "K. Leela Prasad"
age = 33
print("Name:", name)
print("Age:", age)
```

```
Name: K. Leela Prasad
Age: 33
```

3. **Combining Input and Output:**
   - **Example:** Simple Interaction

```
# Getting user input
name = input("Enter your name: ")
age = int(input("Enter your age: "))

# Displaying output
print("Hello, " + name + "!")
print("You are " + str(age) + " years old.")
```

```
Enter your name:  K. Leela Prasad
Enter your age:  33
Hello, K. Leela Prasad!
You are 33 years old.
```

a) **Using format() Method:** The format() method or f-strings can be used to format output in a more readable way.
   - **Example:** Using format() Method

```
name = "K. Leela Prasad"
age = 33
print("Name: {}, Age: {}".format(name, age))
```

```
Name: K. Leela Prasad, Age: 33
```

   - **Example:** Using f-strings (Python 3.6+)

```
name = "K. Leela Prasad"
age = 33
print(f"Name: {name}, Age: {age}")
```

```
Name: K. Leela Prasad, Age: 33
```

1.4 Assignment Statements:

Assignment statements are used to assign values to variables. They are a fundamental aspect of programming, allowing you to store data that can be referenced and manipulated later in your code.

The basic **syntax of an assignment statement is:** variable_name = value

Examples:

1. **Simple Assignment:**

```
x = 10
name = "Alice"
is_active = True
```

2. **Multiple Assignments:**

```
a, b, c = 1, 2, 3
```

3. **Chained Assignment:**

```
x = y = z = 100
```

All three variables x, y, and z are assigned the value 100.

4. **Assignment with Expressions:**

```
total = 5 + 3
```

The expression 5 + 3 is evaluated, and the result 8 is assigned to the variable total.

5. **Updating Variables:**

```
count = 10
count += 5  # Equivalent to count = count + 5
```

6. **Swapping Variables:**

```
x, y = 1, 2
x, y = y, x
```

This swaps the values of x and y. After execution, x will be 2 and y will be 1.

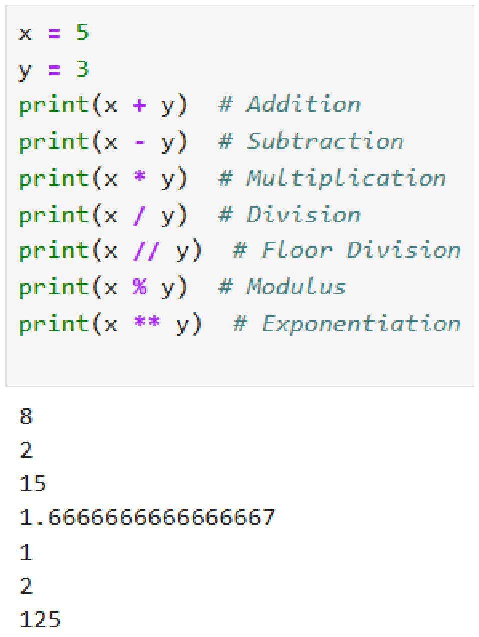
1.5 Operators:

Operators in Python are special symbols that perform specific operations on one or more operands (values). They are used to manipulate variables and values. Python supports different types of operators which are as follows –

1. Arithmetic operators
2. Comparison operators
3. Assignment operators
4. Logical operators
5. Unary operators
6. Bitwise operators
7. Membership operators
8. Identity operators

1. Arithmetic operators:

These operators are used to perform mathematical operations.



```
x = 5
y = 3
print(x + y)  # Addition
print(x - y)  # Subtraction
print(x * y)  # Multiplication
print(x / y)  # Division
print(x // y)  # Floor Division
print(x % y)  # Modulus
print(x ** y)  # Exponentiation
```

```
8
2
15
1.6666666666666667
1
2
125
```

2. Comparison (Relational) Operators:

These operators compare the values of operands and return a Boolean value.

```
print(x == y)  # Equal(==)
print(x != y)  # Not Equal(!=)
print(x > y)  # Greater than (>)
print(x < y)  # Less than (<)
print(x >= y)  # Greater than or Equal To (>=)
print(x <= y)  # Less than or Equal To (<=)
```

```
False
True
True
False
True
False
```

3. Assignment Operators:

These operators are used to assign values to variables.

```
x = 5                 # Assign(=)
x += 3  # x = x + 3   # Add AND (+=)
print(x)
x -= 3  # x = x - 3   # Subtract AND (-=)
print(x)
x *= 3  # x = x * 3   # Multiply AND (*=)
print(x)
x /= 3  # x = x / 3   # Divide AND (/=)
print(x)
x %= 3  # x = x % 3   # Modulus AND (%=)
print(x)
x **= 3  # x = x ** 3 # Exponent AND (**=)
print(x)
x //= 3  # x = x // 3 # Floor Division AND (//=)
print(x)
```

```
8
5
15
5.0
2.0
8.0
2.0
```

4. Logical Operators

These operators are used to combine conditional statements.

```
print(x > 3 and y < 5)  # AND (and):
print(x > 3 or y > 5)  # OR (or):
print(not(x > 3 and y < 5))  # NOT (not):
```

```
False
False
True
```

5. Unary Operators

Unary operators in Python are operators that operate on a single operand. The primary unary operators in python are:

1. **Unary Positive (+):**
    - The unary positive operator returns the value of the operand. It doesn't change the value.
    - **Example:**

    ```
    x = 5
    positive_x = +x
    print(positive_x)  # Output: 5
    ```

    ```
    5
    ```
2. **Unary Negative (-):**
    - The unary negative operator returns the negation of the operand's value.
    - **Example:**

    ```
    x = 5
    negative_x = -x
    print(negative_x)  # Output: -5
    ```

    ```
    -5
    ```
3. **Unary NOT (not):**
    - The unary not operator returns the inverse of the operand's boolean value.
    - **Example:**

    ```
    x = True
    not_x = not x
    print(not_x)  # Output: False
    ```

    ```
    False
    ```

6. Bitwise Operators

These operators perform bit-level operations on binary numbers.

```
x = 5
y = 3
print(x & y)  # AND (&)
print(x | y)  # OR (|)
print(x ^ y)  # XOR (^)
print(~x)  # NOT (~)
print(x << 2)  # Left Shift (<<)
print(x >> 2)  # Right Shift (>>)
```

```
1
7
6
-6
20
1
```

7. Membership Operators:

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

```
x = ["apple", "banana"]
print("banana" in x)  # in:
print("pineapple" not in x)  # not in:
```

```
True
True
```

8. Identity Operators:

These operators compare the memory locations of two objects.

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x is z)  # is:
print(x is y)  # is:
```

```
True
False
```

1.6 Arithmetic Expressions:

An arithmetic expression consists of operands and operators combined. The following arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Operator	Meaning	Example
+	Add two operands or unary plus	x+y+2
-	Subtract right operand from the left or unary minus	x-y-2
*	Multiply two operands	x*y
/	Divide left operand by the right one (always results in float)	x/y
%	Modulus – remainder of the division of left operand by the right	x%y (remainder of x/y)
//	Floor division – division that results into whole number adjusted to the left in the number line.	x//y
**	Exponent – left operand raised to the power of right operand.	x**y (x to the power y)

Example:

```
x = 15
y = 4
print('x + y =',x+y)
print('x - y =',x-y)
print('x * y =',x*y)
print('x / y =',x/y)
print('x // y =',x//y)
print('x ** y =',x**y)
```

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
```

1.8 Operator Precedence:

In Python, operator precedence determines the order in which operations are performed in an expression. If multiple operators present then which operator will be evaluated first is decided by operator precedence. Operators with higher precedence are evaluated before those with lower precedence. If operators have the same precedence, they are evaluated based on their associativity (left-to-right or right-to-left).

Example:

```
print(3+10*2)
print((3+10)*2)
```

```
23
26
```

Operator Precedence Table (From Highest to Lowest)

1. Parentheses ( )
2. Exponentiation **
3. Unary Plus, Minus, and Bitwise NOT +x, -x, ~x
4. Multiplication, Division, Floor Division, Modulus *, /, //, %
5. Addition, Subtraction +, -
6. Bitwise Shift Operators <<, >>
7. Bitwise AND &
8. Bitwise XOR ^
9. Bitwise OR |
10. Comparison Operators ==, !=, >, >=, <, <=
11. Identity Operators is, is not
12. Membership Operators in, not in
13. Logical NOT not
14. Logical AND and
15. Logical OR or
16. Assignment Operators =, +=, -=, *=, /=, //=, %=, **=

Examples:

1. **Parentheses:**

   ```
   result = (2 + 3) * 4
   print(result)  # Output: 20
   ```

   Parentheses have the highest precedence and ensure that 2 + 3 is evaluated first.
2. **Exponentiation:**

   ```
   result = 2 ** 3 ** 2
   print(result)  # Output: 512
   ```

   Exponentiation is right-associative, so 3 ** 2 is evaluated first, then 2 ** 9.
3. **Unary Operators:**

   ```
   result = -3 + 2
   print(result)  # Output: -1
   ```

   ```
   -1
   ```

   Unary minus has higher precedence than addition, so -3 is evaluated first.

4. **Multiplication and Division:**

```
result = 10 / 2 * 3
print(result)  # Output: 15.0
```

```
15.0
```

Multiplication and division have the same precedence and are evaluated left-to-right.

5. **Addition and Subtraction:**

```
result = 5 + 3 - 2
print(result)  # Output: 6
```

```
6
```

Addition and subtraction have the same precedence and are evaluated left-to-right.

6. **Bitwise Operators:**

```
result = 5 & 3 | 2
print(result)  # Output: 3
```

```
3
```

Bitwise AND & is evaluated before Bitwise OR |.

7. **Logical Operators:**

```
result = not False and True or False
print(result)  # Output: True
```

```
True
```

not has higher precedence than and, which has higher precedence than or.

8. **Complex Example:**

```
result = 5 + 2 * (3 ** 2) // 4 - 1
print(result)  # Output: 7
```

```
8
```

Order of Evaluation:

- **Parentheses:** (3 ** 2) → 9
- **Exponentiation:** 3 ** 2 → 9
- **Multiplication**: 2 * 9 → 18
- **Floor Division:** 18 // 4 → 4
- **Addition:** 5 + 4 → 9
- **Subtraction:** 9 - 1 → 8

1.9 Type Casting:

Type Casting in Python refers to the process of converting one data type to another. This is useful when you need to perform operations that require a certain data type or when you want to ensure data consistency. Type casting can be implicit or explicit. Implicit type casting is performed automatically by Python, whereas explicit type casting is done manually using built-in functions.

1. **Implicit Type Casting:**
   - Python automatically converts one data type to another without explicit instruction from the user. This usually happens in expressions involving mixed data types.
   - **Example:**

```python
# Implicit Type Casting
num_int = 10    # Integer
num_float = 5.5   # Float

# Adding integer and float
result = num_int + num_float

print(result)  # Output: 15.5
print(type(result))  # Output: <class 'float'>
```

```
15.5
<class 'float'>
```

2. **Explicit Type Casting:**
   - Explicit type casting requires the use of built-in functions like int(), float(), str(), etc., to convert data types intentionally.
   - **Example 1:** Converting String to Integer

```python
# Explicit Type Casting
num_str = "25"
num_int = int(num_str)  # Converting string to integer

print(num_int)  # Output: 25
print(type(num_int))  # Output: <class 'int'>
```

```
25
<class 'int'>
```

   - **Example2:** Converting Float to Integer

```
# Explicit Type Casting
num_float = 12.7
num_int = int(num_float)  # Converting float to integer

print(num_int)  # Output: 12
print(type(num_int))  # Output: <class 'int'>
```

```
12
<class 'int'>
```

- **Example3:** Using Type Casting in Arithmetic Expressions

```
# Type Casting in Arithmetic Expressions
num1 = "50"  # String
num2 = 20    # Integer

# Converting string to integer before addition
result = int(num1) + num2

print(result)  # Output: 70
print(type(result))  # Output: <class 'int'>
```

```
70
<class 'int'>
```

1.20 Program Comments and Docstrings:

1.20.1 What is a comment?

A comment is a piece of program text that the computer ignores but that provides useful documentation to programmers.

1. **Single line comment in Python**
   - '#' used for single line comments in python programming.
2. **Multiline comments**
   - Multi line comments are not available in Python. If multiple lines are there to comment, use '#' at every line.

1.20.2 What is docstring?

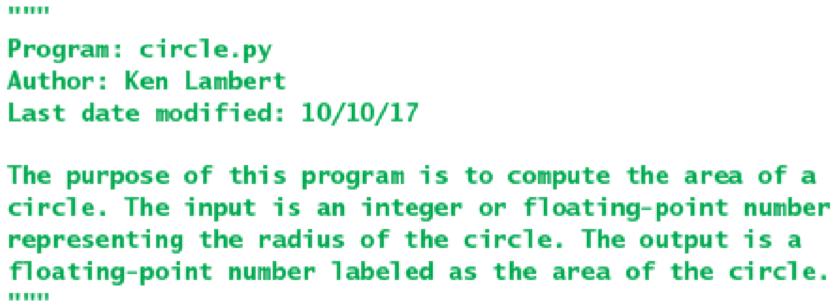
- **Docstrings** in Python are a powerful way to document your code. To know the author of a program can include his or her name and a brief statement about the program's purpose at the beginning of the program file. This type of comment, called a docstring. It is also a form of multi-line string.
- **Docstrings** are special string literals that appear right after the definition of a function, method, class, or module. Docstrings are used to describe what the function, method,

class, or module does, its parameters, return values, and other relevant information. They help developers understand and use the code more effectively.

Syntax of Docstrings:

Docstrings are enclosed in triple quotes (""" or '''), which allows them to span multiple lines.

- **Example of a program author and program’s purpose:**



```
"""
Program: circle.py
Author: Ken Lambert
Last date modified: 10/10/17

The purpose of this program is to compute the area of a
circle. The input is an integer or floating-point number
representing the radius of the circle. The output is a
floating-point number labeled as the area of the circle.
"""
```

- **Example of a Function Docstring:**

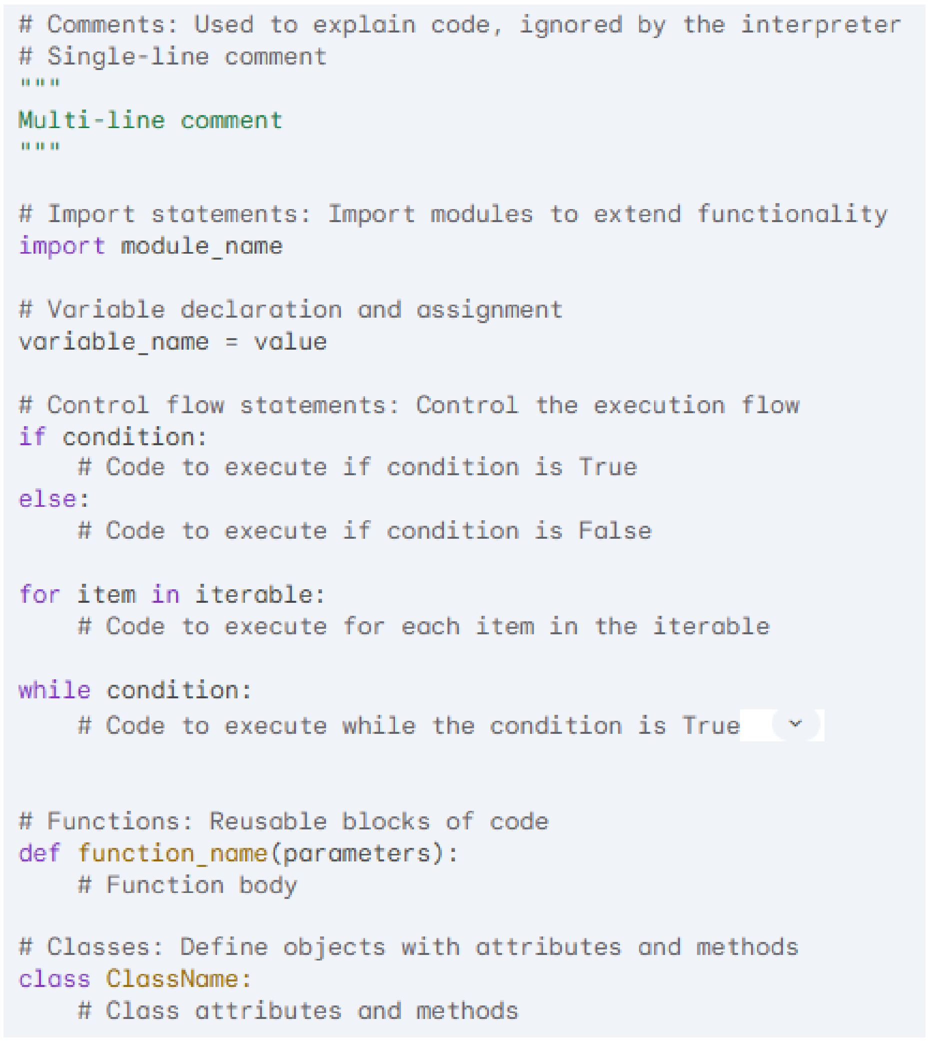
```
def add(a, b):
    """
    Add two numbers and return the result.
        Parameters:
    a (int, float): The first number.
    b (int, float): The second number.
        Returns:
    int, float: The sum of the two numbers.
    """
    return a + b
```

1.9 Program format and Structure:

1.9.1 Program format:

1. Start with an introductory comment stating the author's name, the purpose of the program, and other relevant information. This information should be in the form of a docstring.
2. Then, include statements that do the following:
   - Import any modules needed by the program.
   - Initialize important variables, suitably commented.
   - Prompt the user for input data and save the input data in variables.
   - Process the inputs to produce the results.
   - Display the results.

1.9.2 Basic Structure:



```
# Comments: Used to explain code, ignored by the interpreter
# Single-line comment
"""
Multi-line comment
"""

# Import statements: Import modules to extend functionality
import module_name

# Variable declaration and assignment
variable_name = value

# Control flow statements: Control the execution flow
if condition:
    # Code to execute if condition is True
else:
    # Code to execute if condition is False

for item in iterable:
    # Code to execute for each item in the iterable

while condition:
    # Code to execute while the condition is True


# Functions: Reusable blocks of code
def function_name(parameters):
    # Function body

# Classes: Define objects with attributes and methods
class ClassName:
    # Class attributes and methods
```

1.10 REPL, IDLE:

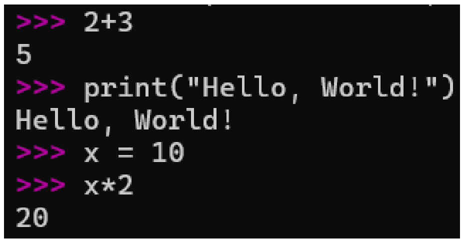
1.10.1 REPL (Read-Eval-Print Loop)

REPL stands for Read-Eval-Print Loop. It's an interactive programming environment that takes single user inputs (one expression), evaluates them, and returns the result to the user. This process repeats, allowing the user to test code snippets in a quick and interactive manner. REPL is an essential tool for learning and experimenting with Python code.

Features of REPL:

- Read: The user inputs a single expression.
- Eval: The interpreter evaluates the expression.
- Print: The result of the evaluation is printed to the console.
- Loop: The process repeats, allowing for continuous interaction.

Example: In the Python shell (REPL environment), you can enter code like:



```
>>> 2+3
5
>>> print("Hello, World!")
Hello, World!
>>> x = 10
>>> x*2
20
```

1.10.2 IDLE (Integrated Development and Learning Environment)

IDLE is an integrated development environment for Python. It comes bundled with Python and provides a simple and user-friendly interface for writing and executing Python code. IDLE is especially useful for beginners as it includes features that make coding in Python more accessible.

Features of IDLE:

- **Interactive Shell:** Allows for interactive code testing (REPL).
- **Editor:** A text editor for writing and saving Python scripts (.py files).
- **Debugger:** Basic debugging capabilities, including setting breakpoints and stepping through code.
- **Introspection:** Helps in understanding objects and their properties.

Using IDLE: When you open IDLE, you are presented with an interactive shell where you can type Python commands and see the results immediately. You can also create a new file (script) and write more extensive programs.

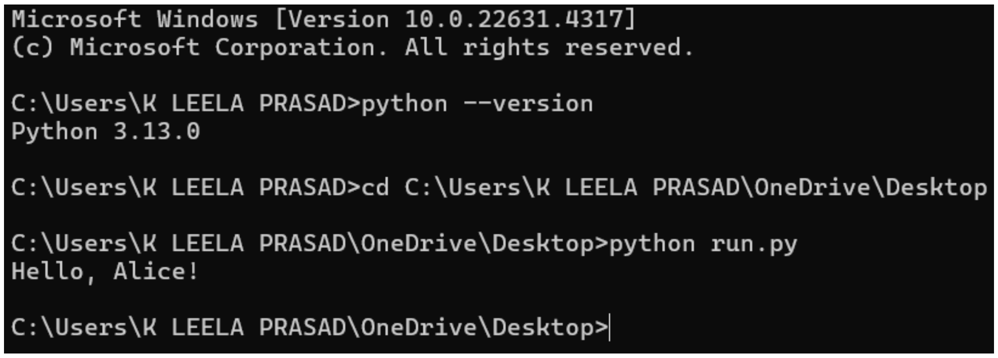
Example: In IDLE, you can interact with the shell similar to REPL or write scripts:

```
>>> # In the IDLE editor, you might write:
... def greet(name):
...     return f"Hello, {name}!"
...
... print(greet("Alice"))
...
... # Save the file and run it to see the output in the interactive shell:
... # Hello, Alice!
>>> [DEBUG ON]
>>> [DEBUG OFF]
>>>
    =========== RESTART: C:/Users/K LEELA PRASAD/OneDrive/Desktop/run.py ===========
    Hello, Alice!
>>>
```

1.10.3 Running a Script from a Terminal Command Prompt:

Steps to Run a Python Script from a Terminal Command Prompt

1. Open Terminal or Command Prompt:
   - Windows: Press Win + R, type cmd, and press Enter.
2. Navigate to the Script's Directory:
   - Use the cd (change directory) command to navigate to the directory where your Python script is located.
   - **Example:**
     cd path/to/your/script
3. Run the Python Script:
   - Type python (or python3 if you have both Python 2 and Python 3 installed) followed by the script's filename.
   - **Example:**
     python script_name.py



```
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Users\K LEELA PRASAD>python --version
Python 3.13.0

C:\Users\K LEELA PRASAD>cd C:\Users\K LEELA PRASAD\OneDrive\Desktop

C:\Users\K LEELA PRASAD\OneDrive\Desktop>python run.py
Hello, Alice!

C:\Users\K LEELA PRASAD\OneDrive\Desktop>
```

UNIT-1: Built – In Functions and Modules

Built-In Functions and Modules; User Defined modules creation and importing a user defined module; NumPy – Functions on 1D arrays, Functions on 2D arrays; Pandas Module-Creation of Series, DataFrames, indexing objects;

1.11 Built-In Functions and Modules:

1.11.1 Built-In Functions:

Python provides many built-in functions that are always available. These functions are powerful tools for performing common operations without the need for additional code.

Some commonly used built-in functions:

1. **print(): Outputs text to the console.**

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

```
Hello, World!
```

2. **len(): Returns the length (number of items) of an object.**

```
my_list = [1, 2, 3, 4]
print(len(my_list))  # Output: 4
```

```
4
```

3. **type(): Returns the type of an object.**

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

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

4. **int(), float(), str(): Converts a value to an integer, float, or string.**

```
num = "100"
print(int(num))    # Output: 100
print(float(num))  # Output: 100.0
print(str(100))    # Output: "100"
```

```
100
100.0
100
```

5. **sum(): Returns the sum of all items in an iterable.**

```
numbers = [1, 2, 3, 4]
print(sum(numbers))  # Output: 10
```

```
10
```

6. **max(), min(): Returns the largest or smallest item in an iterable.**

```
print(max(numbers))
print(min(numbers))
```

```
4
1
```

7. **range(): Generates a sequence of numbers.**

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

```
0
1
2
3
4
```

8. **abs(): Returns the absolute value of a number.**

```
print(abs(-5))
```

```
5
```

9. **round(): Rounds a floating-point number to the nearest integer or to the specified number of decimal places.**

```
print(round(3.14159, 2))
```

```
3.14
```

10. **sorted(): Returns a sorted list from the elements of an iterable.**

```
my_list = [3, 1, 4, 1, 5]
print(sorted(my_list))
```

```
[1, 1, 3, 4, 5]
```

1.11.2 Introduction to Module:

In Python, a **module** is a file containing Python code that can define functions, classes, and variables. It can also include runnable code. Modules in Python are a way to structure your code into manageable, reusable pieces. They help in organizing functions, classes, and variables into separate files, making your code more maintainable and easier to understand.

1.11.2.1 Creating and Importing a Module in Python

Step1: Creating a Module

- A module is simply a Python file with a .py extension. For example, you can create a file named mymodule.py:

```
# mymodule.py

def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b

pi = 3.14159
```

Step2: Import the User-Defined Module

- To use a module, you need to import it in another python file or interactive shell. You can import your module using the import statement.
- **Example:** Create a new Python file named main.py and import your module named mymodule:

```
# main.py

import mymodule

# Using the functions from mymodule
result = mymodule.add(5, 3)
print(result)  # Output: 8

message = mymodule.greet("Alice")
print(message)  # Output: Hello, Alice!

# Using the variable from mymodule
print(mymodule.pi)  # Output: 3.14159
```

Step3: Run your main.py script to see the output.

```
=========== RESTART: C:/Users/K LEELA PRASAD/OneDrive/Desktop/main.py ==========
8
Hello, Alice!
3.14159
```

Importing Specific Functions or Variables

- You can import specific functions, classes, or variables from a module using the from ... import ... **syntax:**

```
# main.py

from mymodule import greet, add, pi

result = add(5, 3)
print(result)  # Output: 8

message = greet("Alice")
print(message)  # Output: Hello, Alice!

print(pi)  # Output: 3.14159
```

Using Aliases

- You can use aliases to give a different name to a module or function while importing:

```
# main.py

import mymodule as mm

result = mm.add(5, 3)
print(result)  # Output: 8

message = mm.greet("Alice")
print(message)  # Output: Hello, Alice!

print(mm.pi)  # Output: 3.14159
```

1.11.2.2 Built-In Modules in Python:

Python's standard library includes many built-in modules that provide additional functionality. These modules can be imported into your programs to leverage their capabilities.

Commonly used Built-In Modules are:

1. math: The math module includes several functions that perform basic mathematical operations. The next code session imports the math module and lists a directory of its resources:

a) **Importing the math Module –**

```
import math
dir(math)
```

```
['__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

- This list of function names includes some familiar trigonometric functions as well as Python's most exact estimates of the constants π and e.
- To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (.) and the name of the resource.
- **Example:** To use the value of pi from the math module, you would write the following code:

a) **Mathematical Functions:**

```
import math
print(math.sqrt(16))
print(math.pi)
print(math.e)
print(math.floor(3.98))
print(math.ceil(3.98))
print(math.pow(3,2))
```

```
4.0
3.141592653589793
2.718281828459045
3
4
9.0
```

b) **Logarithm Functions:** math.log(x[, base]): Returns the logarithm of x to the given base. If no base is specified, returns the natural logarithm (base e).

```
print(math.log(10))
print(math.log(100, 10))
```

```
2.302585092994046
2.0
```

c) **Trigonometric Functions: math.sin(x):** Returns the sine of x (x in radians).

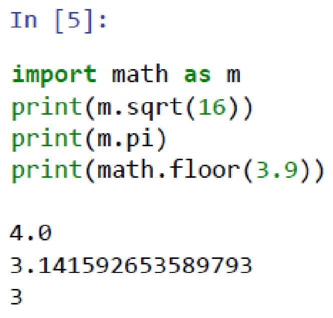
```
import math
print(math.sin(math.pi / 2))  # Output: 1.0
print(math.cos(math.pi))  # Output: -1.0
print(math.tan(math.pi / 4))  # Output: 1.0
print(math.asin(1))  # Output: 1.5707963267948966
print(math.acos(1))  # Output: 0.0
print(math.atan(1))  # Output: 0.7853981633974483
```

```
1.0
-1.0
0.9999999999999999
1.5707963267948966
0.0
0.7853981633974483
```

b) **Module Aliasing:**

- Whenever we are importing math module, so any variable or function we can use within that module by specifying with the name of that module (For example, math.sqrt and math.py etc).
- If the module name is bigger then there is a problem of every time make using of that bigger module name when you are calling the function within that module. As result of this length of the code increases. To avoid this, we can create alias name by using as keyword.

- **import math as m**
- Once we create alias name, by using that we can access functions and variables of that module.



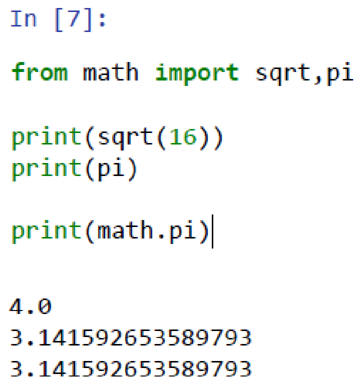
```
In [5]:

import math as m
print(m.sqrt(16))
print(m.pi)
print(math.floor(3.9))

4.0
3.141592653589793
3
```

- We can import a particular member of a module explicitly as follows

  Example1:

  from math import sqrt

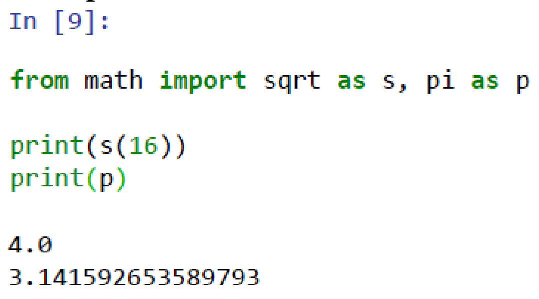
  from math import sqrt, pi
- If we import a member explicitly then it is not required to use module name while accessing.
- **Example2:**



```
In [7]:

from math import sqrt,pi

print(sqrt(16))
print(pi)

print(math.pi)

4.0
3.141592653589793
3.141592653589793
```

c) Member Aliasing:

- We can create alias name for member of module also.
- **Example:**



```
In [9]:

from math import sqrt as s, pi as p

print(s(16))
print(p)

4.0
3.141592653589793
```

2. datetime: Supplies classes for manipulating dates and times.

```python
from datetime import datetime
now = datetime.now()
print(now)  # Output: current date and time
```

```
2024-11-12 09:59:03.221965
```

3. os: Provides functions for interacting with the operating system.

- Knowing the Current Working Directory, **getcwd()** function from the os module to get the current working directory.
- Setting a New Working Directory, **chdir()** function from the os module to change the working directory.

```python
import os

# Get the current working directory
current_directory = os.getcwd()
print(f"Current working directory: {current_directory}")

# Set a new working directory
new_directory = 'D:/Work'
os.chdir(new_directory)

# Verify that the working directory has been changed
current_directory = os.getcwd()
print(f"New working directory: {current_directory}")
```

```
Current working directory: D:\Work
New working directory: D:\Work
```

4. Statistics: Provides functions for statistical calculations

The statistics module in Python provides functions for performing statistical operations on numeric data. It's a part of the Python Standard Library, which means you don't need to install anything extra to use it.

a) Importing the statistics Module

- To use the functions in the statistics module, you need to import it:
- import statistics
- **Example:**

```
import statistics

data = [1, 2, 3, 4, 5, 6, 6]
print(statistics.mean(data))
print(statistics.median(data))
```

```
3.857142857142857
4
```

b) Measures of Central Tendency

- **Mean** (statistics.mean): Calculates the arithmetic mean (average) of a list of numbers.

```
data = [1, 2, 3, 4, 5]
mean_value = statistics.mean(data)
print("Mean:", mean_value)
```

```
Mean: 3
```

- **Median (**statistics.median**)**: Calculates the median (middle value) of a list of numbers.

```
median_value = statistics.median(data)
print("Median:", median_value)   # Output: Median: 3
```

```
Median: 3
```

- **Mode (**statistics.mode**)**: Calculates the mode (most common value) of a list of numbers.

```
data_with_mode = [1, 1, 2, 3, 4]
mode_value = statistics.mode(data_with_mode)
print("Mode:", mode_value)  # Output: Mode: 1
```

```
Mode: 1
```

c) Measures of Variability

- **Variance** (statistics.variance): Calculates the variance of a list of numbers.
- **Example:**

```
variance_value = statistics.variance(data)
print("Variance:", variance_value)  # Output: Variance: 2.5
```

```
Variance: 2.5
```

- **Standard Deviation** (statistics.stdev): Calculates the standard deviation of a list of numbers.

```
stdev_value = statistics.stdev(data)
print("Standard Deviation:", stdev_value)
```

```
Standard Deviation: 1.5811388300841898
```

5. random: Implements pseudo-random number generators.

The random module in Python is a standard library module that provides a suite of tools to generate random numbers, select random items, shuffle data, and more. Its key functions are:

a) Basic Random Number Generation

- **randint(a, b):**
  - Returns a random integer N such that a <= N <= b.
  - **Example:**

```
import random
print(random.randint(1, 10))
```

```
7
```

- **random():**
  - **Returns a random float number between 0.0 and 1.0.**
  - **Example:**

```
import random
print(random.random())
```

```
0.601793272060729
```

b) Random Selections

- **choice(seq):**
  - returns a random element from non-empty sequence seq.
  - **Example:**

```
colors = ['red', 'green', 'blue', 'yellow']
print(random.choice(colors))  # Example output: 'blue'
```

```
yellow
```

- **choices(population, weights=None, *, k=1):**
  - Returns a list with k random elements from the population, with optional weights.
  - **Example:**

```
print(random.choices(colors, k=2))
```

```
['yellow', 'red']
```

- **sample(population, k):**
  - Returns a list with k unique elements from the population.
  - **Example:**

```
print(random.sample(colors, 3))
```

```
['yellow', 'red', 'green']
```

c) Shuffling Data

- **shuffle(seq):**

- Shuffles the sequence seq in place.
- **Example:**

```
deck = [1, 2, 3, 4, 5]
random.shuffle(deck)
print(deck)  # Example output: [5, 2, 3, 1, 4]
```

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

1.12 Numpy – Functions on 1D Arrays:

- NUMPY-NumPy stands for "Numeric Python" or "Numerical python". NumPy is a package that contains several classes, functions, variables etc. to deal with scientific calculations in Python. NumPy is useful to create and process single and multi-dimensional arrays. In addition, NumPy contains a large library of mathematics like linear algebra functions and Fourier transformations.
- The arrays which are created using numpy are called n dimensional arrays where n can be any integer. If n = 1 it represents a one-dimensional array. If n= 2, it is a two-dimensional array etc.
- NumPy array can accept only one type of elements. We cannot store different data types into same arrays.

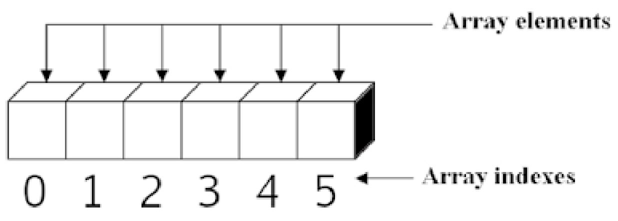
1.12.1 Types of arrays in NumPy:

An array in numpy is of the following types-

1. 1-D Array
2. 2-D Array
3. N-Dimension Array

1. 1-D Array in NumPy:

- 1D Array- One dimensional array contains elements only in one dimension. In other words, the shape of the numpy array should contain only one value in the tuple.



One-dimensional array with six elements

Example: A simple program to implement one dimensional array using numpy

```
import numpy
a = numpy.array([10,20,30,40,50])
print(a)
```

```
[10 20 30 40 50]
```

```
import numpy as np
a = np.array([10,20,30,40,50])
print(a)
```

```
[10 20 30 40 50]
```

Note: if we use the following statement then there is no need to add anything in front of array function.

```
from numpy import *
a = array([10, 20,30,40,50])
print(a)
```

```
[10 20 30 40 50]
```

1.12.2 Creation of 1D Array in Numpy:

Creating array in numpy can be done in several ways. Some of the important ways are-

1. Using array() function
2. Using linspace() function
3. Using arange() function
4. Using zeros() and ones() functions

1. Using array() function:

- Using this function, we can create array of any data type, but if not, data types is mentioned the default data type will be the "int".
- **Example:**

  ```
  from numpy import *
  Arr=array([10, 20, 30, 40, 50],int)
  print(Arr)
  ```

  ```
  [10 20 30 40 50]
  ```

- While creating array if one of the values in the specified list belongs to float then all the values will be converted to float by default.

```
from numpy import *
a = array([10,30,40.5, 50,100])
print(a)
```

```
[ 10.    30.    40.5   50.   100. ]
```

2. linspace() Function:

- The linspace() function is used to create an array with evenly spaced points between a starting and ending point. The following examples demonstrate the use of linspace() function.
- **Syntax:** linspace(start, stop, N)
- **Example:**

```
import numpy as np
a = np.linspace(1,10,6)
print(a)
```

```
[ 1.   2.8  4.6  6.4  8.2 10. ]
```

3. arange() Function:

- The arange() function in numpy is same as range() function in Python. The following format is used to create an array using the arange() function.
- **Syntax-**
  arrange (start, stop, stepsize)
- **Example:**

```
import numpy as np
a = np.arange(10)
b = np.arange(5,10)
c = np.arange(10,1,-1)
print(a)
print(b)
print(c)
```

```
[0 1 2 3 4 5 6 7 8 9]
[5 6 7 8 9]
[10  9  8  7  6  5  4  3  2]
```

4. Creating array using ones() and zeros() functions:

- We can use zeros() function to create an array with all zeros. The ones() function will is useful to create an array with all 1s.
- They are written in the following format-
  - zeros(n,datatype)

    - ones(n,datatype)

- **Note:** if datatype is missing then the default value will be float.
- **Example:**

```
import numpy as np
L = np.zeros(5)
P = np.ones(5)
print(L)
print(P)
```

```
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
```

1.12.3 Indexing and Slicing:

Indexing and slicing are fundamental operations in NumPy that allow you to access and manipulate elements of arrays efficiently. The key functions and techniques for indexing and slicing 1D arrays in NumPy:

a) Indexing

- **Basic Indexing:** You can access individual elements of a 1D array using their indices. NumPy arrays are zero-indexed, meaning the first element has an index of 0.

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Access the first element
print(arr[0])  # Output: 10

# Access the third element
print(arr[2])  # Output: 30

# Access the last element
print(arr[-1])  # Output: 50
```

```
10
30
50
```

b) Slicing

- **Basic Slicing:** Slicing allows you to access a range of elements in an array. The syntax is arr[start:stop:step], where start is the beginning index, stop is the end index (exclusive), and step is the interval between indices.

```
# Slice the first three elements
print(arr[0:3])  # Output: [10, 20, 30]

# Slice from the second to the end
print(arr[1:])  # Output: [20, 30, 40, 50]

# Slice with step
print(arr[::2])  # Output: [10, 30, 50]
```

```
[10 20 30]
[20 30 40 50]
[10 30 50]
```

1.12.4 Operations on 1-D array:

Once arrays are declared, we can access its element or perform certain operations.

1. **Arithmetic Operations on Arrays –**
    - It is possible to perform various arithmetic operations like addition, subtraction, division etc. on the elements of any arrays. The functions of math module can be applied to the elements of any array.

```
import numpy as np
K = np.array([10, 20, 30, 40,50])
K = K+5
print("Addition: ",K)
K = K-5
print("Subtraction: ",K)
K = K*5
print("Multiplication: ",K)
K = K/5
print("Division: ",K)
```

```
Addition:  [15 25 35 45 55]
Subtraction:  [10 20 30 40 50]
Multiplication:  [ 50 100 150 200 250]
Division:  [10. 20. 30. 40. 50.]
```

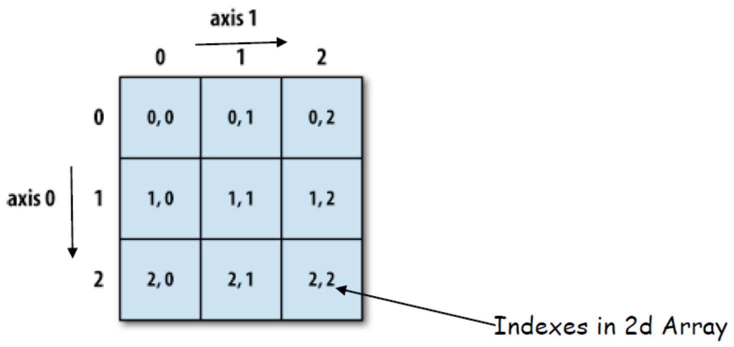
2. **Sorting -** Sorting the array

```
arr = np.array([5, 1, 3, 2, 4])
sorted_arr = np.sort(arr)
print("Sorted Array:", sorted_arr)
```

```
Sorted Array: [1 2 3 4 5]
```

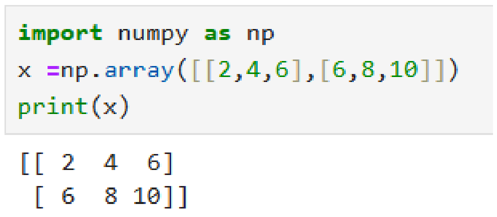
2. 2-D Array in NumPy:

The dimension of an array represents the arrangement of elements in the array. If the elements are arranged horizontally, it is called the row and if the elements are arranged vertically, then it is called the column. When they contain only one row and one column of elements, it is called the Single dimensional array or one-dimensional array. When an array contains more than one row and more than one column of elements, it is called the two-dimensional array or 2-D array. The following example is used demonstrate how to declare the two-dimensional array in using numpy.



2D Array structure

1. Creation of 2-D Array in NumPy:



```
import numpy as np
x =np.array([[2,4,6],[6,8,10]])
print(x)
```

```
[[ 2  4  6]
 [ 6  8 10]]
```

1.12.6 Attributes of 2-D Array:

a) ndim Attribute –

- ‘ndim’ attribute is used to represent the number of dimensions of axes of the array. The number of dimensions is also known as 'rank'. The following example demonstrate the use of the ndim attribute.
- **Example:**

```
import numpy as np
A = np.array([5,6,7,8])
R = np.array([[4,5,6],[7,8,9]])
print(A.ndim)
print(R.ndim)
```

```
1
2
```

b) shape attribute –

- The 'shape' attribute gives the shape of an array. The shape is tuple listing the number of elements along each dimension. A dimension is called an axis. For one dimensional array it will display a single value and for two-dimensional array it will display two values separated by commas represent rows and columns.
- **Example:**

```
import numpy as np
k = np.array([1,2,3,4,5])
print(k.shape)
d = np.array([[5,6,7],[7,8,9]])
print(d.shape)
```

```
(5,)
(2, 3)
```

c) size Attribute –

- The size attributes give the total number of elements in the array.
- **Example:**

```
import numpy as np
a1 = np.array([1,2,3,4,5])
print(a1.size)
```

```
5
```

- In case of two-dimensional array, the result will be total rows* total columns.

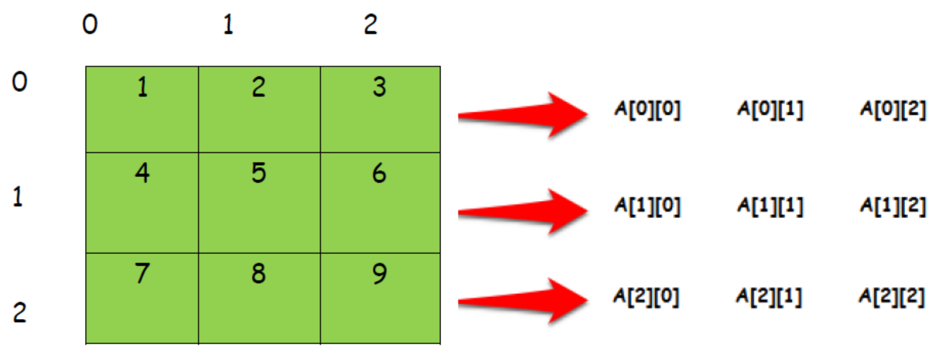
```
import numpy as np
k = np.array([[5,6,7],[7,8,9]])
print(k.size)
```

```
6
```

1.12.7 Indexing:

a) Index represents the location number. The individual elements of an array can be accessed by specifying the location number of the row and column of the array element as follow-

- A[0][0] => represents 0th row and 0th column element in array A
- A[1][3] => represents 1st row and 3rd column element in the array A

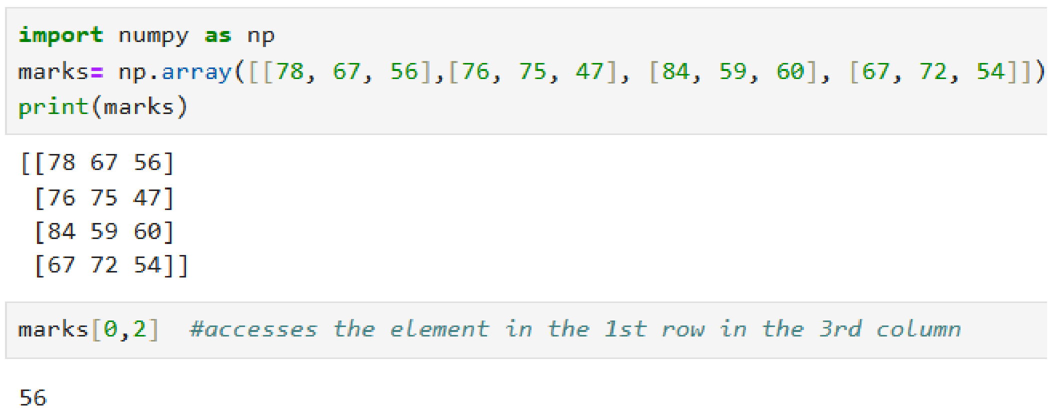


Example: Consider the table showing marks of students in different subjects

Name	Maths	English	Science
A1	78	67	56
A2	76	75	47
A3	84	59	60
A4	67	72	54

Let us create an array called marks to store marks given in three subjects for four students given in this table. As there are 4 students (i.e. 4 rows) and 3 subjects (i.e. 3 columns), the array will be called marks [4][3]. This array can store 4*3 = 12 elements.

Here, marks [i, j] refers to the element at (i+1)th row and (j+1)th column because the index values start at 0. Thus marks [3,1] is the element in 4th row and second column which is 72 (marks of A2 in English).



```python
import numpy as np
marks= np.array([[78, 67, 56],[76, 75, 47], [84, 59, 60], [67, 72, 54]])
print(marks)
```

```
[[78 67 56]
 [76 75 47]
 [84 59 60]
 [67 72 54]]
```

```python
marks[0,2]   #accesses the element in the 1st row in the 3rd column
```

```
56
```

```
marks [0,4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[118], line 1
----> 1 marks [0,4]

IndexError: index 4 is out of bounds for axis 1 with size 3
```

1.12.8 Operations on 2-D array:

Arithmetic operations on NumPy arrays are fast and simple. When we perform a basic arithmetic operation like addition, subtraction, multiplication, division etc. on two arrays, the operation is done on each corresponding pair of elements. For instance, adding two arrays will result in the first element in the first array to be added to the first element in the second array, and so on.

Consider the following element-wise operations on two arrays:

```
array1 = np.array([[3,6],[4,2]])
array2 = np.array([[10,20],[15,12]])

#Element-wise addition of two matrices.
array1 + array2
```

```
array([[13, 26],
       [19, 14]])
```

```
array1 = np.array([[3,6],[4,2]])
array2 = np.array([[10,20],[15,12]])

#Element-wise subtraction of two matrices.
array1 - array2
```

```
array([[ -7, -14],
       [-11, -10]])
```

```
#Element-wise multiplication of two matrices.
array1 * array2
```

```
array([[ 30, 120],
       [ 60,  24]])
```

```
#Matrix multiplication
array1 @ array2
```

```
array([[120, 132],
       [ 70, 104]])
```

```
#Division
array2 / array1
```

```
array([[3.33333333, 3.33333333],
       [3.75      , 6.        ]])
```

1.13 Data Handling Using Pandas:

The Pandas module is a powerful and popular Python library used for data manipulation and analysis. It provides data structures and functions needed to work with structured data seamlessly and efficiently.

Some of the key features and functionalities of Pandas:

1. **Data Structures:** Pandas provides two primary data structures:
   - **Series:** A one-dimensional labeled array capable of holding any data type.
   - **DataFrame:** A two-dimensional labeled data structure with columns of potentially different types.
2. **Data Alignment and Indexing:** Pandas allows automatic and explicit data alignment, making it easy to handle missing data and perform operations on data.
3. **Handling Missing Data:** Functions to check, fill, and drop missing data in datasets.
4. **Data Manipulation:** Functions to merge, concatenate, and reshape data.
5. **Data Aggregation and Grouping:** Tools to split data into groups, apply functions, and combine results.
6. **Input/Output Tools:** Functions to read from and write to various file formats like CSV, Excel, SQL databases, and more.

1.13.1 Installing Pandas –

Installing Pandas is very similar to installing NumPy. To install Pandas from command line, we need to type in:

pip install pandas

1.13.2 Data Structure in Pandas –

A data structure is a collection of data values and operations that can be applied to that data. It enables efficient storage, retrieval and modification to the data. Two commonly used data structures in Pandas are:

1. Series
2. DataFrame

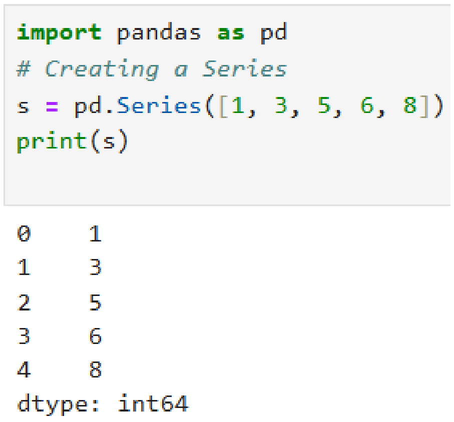
1. Series:

A **Series** is a one-dimensional array containing a sequence of values of any data type (int, float, list, string, etc) which by default have numeric data labels starting from zero. The data label associated with a particular value is called its index. We can also assign values of other data types as index. We can imagine a Pandas Series as a column in a spreadsheet. **Example of a series containing names of students is given below:**

Index	**Value**
0	Arnab
1	Samridhi
2	Ramit
3	Divyam

a) Creation of Series with lists –

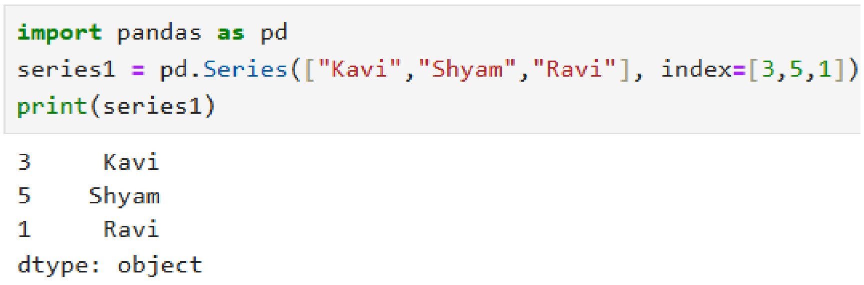
- There are different ways in which a series can be created in Pandas. To create or use series, we first need to import the Pandas library.



```python
import pandas as pd
# Creating a Series
s = pd.Series([1, 3, 5, 6, 8])
print(s)
```

```
0    1
1    3
2    5
3    6
4    8
dtype: int64
```

- We can also assign user-defined labels to the index and use them to access elements of a Series. The following example has a numeric index in random order.



```python
import pandas as pd
series1 = pd.Series(["Kavi","Shyam","Ravi"], index=[3,5,1])
print(series1)
```

```
3     Kavi
5    Shyam
1     Ravi
dtype: object
```

2. DataFrame:

- A DataFrame is a two-dimensional labelled data structure like a table of MySQL. It contains rows and columns, and therefore has both a row and column index.
- Each column can have a different type of value such as numeric, string, Boolean, etc., as in tables of a database.

a) Creation of an empty DataFrame

```
import pandas as pd
dFrameEmt = pd.DataFrame()
dFrameEmt
```

b) Creation of DataFrame from a list of lists

```
import pandas as pd

# Create a list of lists
data = [
    [1, 'Alice', 23],
    [2, 'Bob', 25],
    [3, 'Charlie', 30]
]

# Create a DataFrame
df = pd.DataFrame(data, columns=['ID', 'Name', 'Age'])

print(df)
```

```
   ID     Name  Age
0   1    Alice   23
1   2      Bob   25
2   3  Charlie   30
```

c) Creation of DataFrame from NumPy ndarrays

Create a simple DataFrame without any column labels, using a single ndarray:

```
import numpy as np
array1 = np.array([10,20,30])
array2 = np.array([100,200,300])
array3 = np.array([-10,-20,-30, -40])
dFrame4 = pd.DataFrame(array1)
dFrame4
```

```
    0
0  10
1  20
2  30
```

- We can create a DataFrame using more than one ndarrays, as shown in the following example:

```
dFrame5 = pd.DataFrame([array1, array3,
array2], columns=[ 'A', 'B', 'C', 'D'])
dFrame5
```

	A	B	C	D
0	10	20	30	NaN
1	-10	-20	-30	-40.0
2	100	200	300	NaN

1.13.3 Indexing objects:

In Pandas, indexing is a fundamental feature that allows you to access, manipulate, and analyze data efficiently. Indexing objects in Pandas, focusing on two primary data structures: Series and DataFrame.

1. Numerical Indexing
2. Categorical Indexing

1. Numerical Indexing:

a) Indexing in Pandas Series:

- Numerical indexing in a pandas Series refers to accessing elements using their integer-based positions.

```
import pandas as pd

# Create a sample Series
data = pd.Series([10, 20, 30, 40, 50])
# Numerical indexing
print(data[0])       # Output: 10
print(data[1:4])     # Output: 20, 30, 40
```

```
10
1    20
2    30
3    40
dtype: int64
```

b) Indexing in Pandas DataFrame:

- Numerical indexing in a DataFrame uses integer-based row and column positions to access data.

```python
data = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8]
})
print(data)
```

```
   A  B
0  1  5
1  2  6
2  3  7
3  4  8
```

➢ Accessing the elements of the above DataFrame.

```python
import pandas as pd

# Create a sample DataFrame
data = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8]
})
# Numerical indexing
print(data.iloc[0])       # First row
# Output:
# A    1
# B    5
# Name: 0, dtype: int64

print(data.iloc[0, 1])    # Element at first row, second column
# Output: 5

print(data.iloc[1:3])     # Rows 1 and 2
# Output:
#    A  B
# 1  2  6
# 2  3  7
```

```
A    1
B    5
Name: 0, dtype: int64
5
   A  B
1  2  6
2  3  7
```

2. Categorical Indexing:

a) Indexing in Pandas Series:

- Categorical indexing in a Series involves using labels or categories to access elements.

```
import pandas as pd

# Create a sample Series with labels
data = pd.Series([100, 200, 300], index=['a', 'b', 'c'])

# Categorical indexing
print(data['a'])            # Output: 100
print(data[['a', 'c']])     # Output: 100, 300
```

```
100
a    100
c    300
dtype: int64
```

b) Indexing in Pandas DataFrame:

- Categorical indexing in a DataFrame uses labels or categories to access rows and columns.

```
import pandas as pd

# Create a sample DataFrame with labels
data = pd.DataFrame({
    'A': [10, 20, 30, 40],
    'B': [50, 60, 70, 80]
}, index=['x', 'y', 'z', 'w'])

# Categorical indexing
print(data.loc['x'])       # Row with label 'x'

print(data.loc['x', 'B'])  # Element at row 'x', column 'B'

print(data.loc[['x', 'z']]) # Rows with labels 'x' and 'z'
```

```
A    10
B    50
Name: x, dtype: int64
50
    A   B
x  10  50
z  30  70
```

Assignment Questions:

1) Explain the various data types in Python with examples.
2) What are escape sequences in Python? Describe their importance with examples.
3) Explain assignment statements in Python with suitable examples.
4) Discuss different types of operators in Python with examples.
5) Write a detailed note on arithmetic expressions in Python and explain operator precedence with examples.
6) What is type casting in Python? Explain with examples how to convert data types.
7) Why are comments and docstrings important in Python programming? Explain with examples.
8) Describe the typical format and structure of a Python program.
9) Discuss the Read-Eval-Print Loop (REPL) environment, its benefits, and how it helps in interactive programming and debugging.
10) How do you run a Python script from the IDLE environment and from the terminal command prompt?
11) Discuss the role of built-in functions in Python with examples.
12) Explain the concept of modules in Python. How do you use built-in modules with examples?
13) Write a detailed note on creating and importing user-defined modules in Python with examples.
14) Discuss several ways for the creation and accessing elements of 1D Array in NumPy with examples.
15) Discuss several ways for the creation and accessing elements of 2D Array in NumPy with examples.
16) Describe the process of creating a Series and a DataFrame in pandas. How do you perform indexing on these objects? Provide examples.