Middle Technical University

Technical Engineering College of Artificial Intelligence

Department of Cybersecurity Technology Engineering

Third Year (2025- 2026)

PYTHON PROGRAMMING

Lecture 2: Python Basics

By: Ass. Lec. Abbas Aqeel Kareem

Python Basics

In this section, we will learn the core concepts of Python programming, including variables, data types, operators, control statements, and basic data structures, to build simple and logical Python programs.

Learning Objectives

After completing this chapter, you should be able to

- Understand Python variables, data types, and operators.
- Apply conditional statements and loops to control program flow.
- Manipulate strings and core data structures (lists, tuples, sets, dictionaries, and arrays).
- Develop simple Python programs demonstrating logical and structured problem-solving.

Python Variables

In Python, variables are used to store data that can be referenced and manipulated during program execution. A variable is essentially a name that is assigned to a value. Unlike many other programming languages, Python variables do not require explicit declaration of type. The type of the variable is inferred based on the value assigned.

Variables act as placeholders for data. They allow us to store and reuse values in our program.

Code	Output

`# Variable 'age' stores the integer value 29`
`age = 29`
`# Variable 'name' stores the string "abbas aqeel"`
`name = "abbas aqeel"`
`print(age)`
`print(name)`
	`29`
	`abbas aqeel`

1.1. Rules for Naming Variables

To use variables effectively, we must follow Python's naming rules:

- Variable names can only contain letters, digits and underscores (_).
- A variable name cannot start with a digit.
- Variable names are case-sensitive (myVar and myvar are different).
- Avoid using Python keywords (e.g., if, else, for) as variable names.

Valid	Invalid

`age = 21`
`_colour = "lilac"`
`total_score = 90`
	`1name = "Error"  # Starts with a digit`
	`class = 10  # 'class' is a reserved keyword`
	`user-name = "Doe"  # Contains a hyphen`

1.2.Assigning Values to Variables

Variables in Python are assigned values using the = operator

Code

`x = 5`
`y = 3.14`
`z = "Hi"`

Python variables are dynamically typed, meaning the same variable can hold different types of values during execution.

Code
`x = 10` `x = "Now a string"`

We can assign different values to multiple variables simultaneously, making the code concise and easier to read.

Code	Output
`x, y, z = 1, 2.5, "Python"` `print(x, y, z)`	`1 2.5 Python`

1.3. Type Casting a Variable

Type casting refers to the process of converting the value of one data type into another. Python provides several built-in functions to facilitate casting, including:

- `int()`: Converts compatible values to an integer
- `float()`: Transforms values into floating-point numbers.
- `str()`: Converts any data type into a string.

Code	Output

`# Casting variables`
`s = "10"  # Initially a string`
`n = int(s)  # Cast string to integer`
`cnt = 5`
`f = float(cnt)  # Cast integer to float`
`age = 25`
`s2 = str(age)  # Cast integer to string`
`# Display results`
`print(n)`
`print(f)`
`print(s2)`
	`10`
	`5.0`
	`25`

1.4. Getting the Type of Variable

In Python, we can determine the type of a variable using the type() function. This built-in function returns the type of the object passed to it.

Code	Output

`# Define variables with different data types`
`n = 42`
`f = 3.14`
`s = "Hello, World!"`
`li = [1, 2, 3]`
`d = {'key': 'value'}`
`bool = True`
`# Get and print the type of each variable`
`print(type(n))`
`print(type(f))`
`print(type(s))`
`print(type(li))`
`print(type(d))`
`print(type(bool))`
	`<class 'int'>`
	`<class 'float'>`
	`<class 'str'>`
	`<class 'list'>`
	`<class 'dict'>`
	`<class 'bool'>`

1.5. Delete a Variable Using `del` Keyword

We can remove a variable from the namespace using the `del` keyword. This effectively deletes the variable and frees up the memory it was using.

Code	Output

`# Assigning value to variable`
`x = 10`
`print(x)`
`# Removing the variable using del`
`del x`
`print(x)`
	`10`
	`Traceback (most recent call last):`
	`  print(x)`
	`  ^`
	`NameError: name 'x' is not defined`

2. Python Operators

In Python programming, Operators in general are used to perform operations on values and variables. These are standard symbols used for logical and arithmetic operations. In this section, as shown in Table 1 we will look into different types of Python operators.

Table 1: Types of Operators in Python.

Operators	Type
`+ , - , * , /, %`	Arithmetic Operators
`==, !=, >, <, >=, <=`	Comparison Operators
`AND, OR, NOT`	Logical Operators
`&, |, >>, <<, ^, ~`	Bitwise Operators
`=, +=, -=, *=, /= , %=`	Assignment Operators

2.1. Arithmetic Operators

Python Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication and division.

In Python 3.x the result of division is a floating-point while in Python 2.x division of 2 integers was an integer. To obtain an integer result in Python 3.x floored (// integer) is used.

Code	Output

`# Variables`
`a = 15`
`b = 4`
`# Addition`
`print("Addition:", a + b)`
`# Subtraction`
`print("Subtraction:", a - b)`
`# Multiplication`
`print("Multiplication:", a * b)`
`# Division`
`print("Division:", a / b)`
`# Floor Division`
`print("Floor Division:", a // b)`
`# Modulus`
`print("Modulus:", a % b)`
`# Exponentiation`
`print("Exponentiation:", a ** b)`
	`Addition: 19`
	`Subtraction: 11`
	`Multiplication: 60`
	`Division: 3.75`
	`Floor Division: 3`
	`Modulus: 3`
	`Exponentiation: 50625`

2.2. Comparison Operators

In Python, Comparison (or Relational) operators compares values. It either returns True or False according to the condition.

Code	Output

`a = 13`
`b = 33`
`# greater than`
`print(a > b)`
`# less than`
`print(a < b)`
`# equal to`
`print(a == b)`
`# not equal to`
`print(a != b)`
`# greater or equal`
`print(a >= b)`
`#less or equal`
`print(a <= b)`
	`False`
	`True`
	`False`
	`True`
	`False`
	`True`

2.3. Logical Operators

Python logical operators are used to combine conditional statements, allowing you to perform operations based on multiple conditions. These Python operators, alongside arithmetic operators, are special symbols used to carry out computations on values and variables. In Python, Logical operators are used on conditional statements (either True or False). They perform Logical AND, Logical OR, and Logical NOT operations as shown in table 2.

Table 2: Python Logical Operators

Operator	Description	Syntax
and	Returns True if both the operands are true.	`x and y`
or	Returns True if either of the operands is true.	`x or y`
not	Returns True if the operand is false.	`not x`

Code	Output

`# Example: Logical Operators (AND, OR, NOT) with generic variables`
`a, b, c = True, False, True`
 	
`# AND: Both conditions must be True`
`if a and c:`
`  print("Both a and c are True (AND condition).")`
 	
`# OR: At least one condition must be True`
`if b or c:`
`  print("Either b or c is True (OR condition).")`
 	
`# NOT: Reverses the condition`
`if not b:`
`  print("b is False (NOT condition).")`
	`Both a and c are True (AND condition).`
	`Either b or c is True (OR condition).`
	`b is False (NOT condition).`

2.4. Bitwise Operators

Python bitwise operators as shown in table 3are used to perform bitwise calculations on integers. The integers are first converted into binary and then operations are performed on each bit or corresponding pair of bits, hence the name bitwise operators. The result is then returned in decimal format.

Table 3: Python bitwise operators.

Operator	Description	Syntax
&	Bitwise AND.	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Python **Bitwise AND (&)** operator takes two equal-length bit patterns as parameters. The two-bit integers are compared. If the bits in the compared positions of the bit patterns are 1, then the resulting bit is 1. If not, it is 0. For example, Take two bit values X and Y, where $X = 7 = (111)_2$ and $Y = 4 = (100)_2$.

$$\begin{array}{r} 111 \\ \&100 \\ \hline 100 \end{array} = 4$$

The Python **Bitwise OR (|)** Operator takes two equivalent length bit designs as boundaries; if the two bits in the looked-at position are 0, the next bit is zero. If not, it is 1. For example, Take two bit values X and Y, where $X = 7 = (111)_2$ and $Y = 4 = (100)_2$. Take Bitwise OR of both X, Y.

$$\begin{array}{r} 111 \\ |\ 100 \\ \hline 111 \end{array} = 7$$

The Python **Bitwise XOR (^)** Operator also known as the exclusive OR operator, is used to perform the XOR operation on two operands. XOR stands for "exclusive or", and it returns true if and only if exactly one of the operands is true. In the context of bitwise operations, it compares corresponding bits of two operands. If the bits are different, it returns 1; otherwise, it returns 0. For example, Take two bit values X and Y, where $X = 7 = (111)_2$ and $Y = 4 = (100)_2$. Take Bitwise and of both X & Y.

$$\begin{array}{r} 111 \\ \hat{}100 \\ \hline 011 \end{array} = 3$$

Python **Bitwise Not (~)** Operator works with a single value and returns its one's complement. This means it toggles all bits in the value, transforming 0 bits to 1 and 1 bits to 0, resulting in the one's complement of the binary number. For example, where `X = 7= (101)`$_2$. Python integers are signed and use two's complement arithmetic, so `~x = -(x+1) = -(7+1) = -8.`

Bitwise Right Shift (>>) Operator Shifts the bits of the number to the right and fills 0 on voids left (fills 1 in the case of a negative number) as a result. Similar effect as of dividing the number with some power of two.

$$2 >> \quad 0000\ 0111$$

$$0000\ 0001 = 1$$

Bitwise Left Shift (<<) Operator Shifts the bits of the number to the left and fills 0 on voids right as a result. Similar effect as of multiplying the number with some power of two.

$$2 << \quad 0000\ 0111$$

$$0001\ 1100 = 28$$

Code	Output

`a = 7`
`b = 4`
`print(a & b)`
`print(a | b)`
`print(~a)`
`print(a ^ b)`
`print(a >> 2)`
`print(a << 2)`
	`4`
	`7`
	`-8`
	`3`
	`1`
	`28`

2.5. Assignment Operators

Python Assignment operators are used to assign values to the variables. This operator is used to assign the value of the right side of the expression to the left side operand.

The **Addition Assignment** Operator is used to add the right-hand side operand with the left-hand side operand and then assigning the result to the left operand.

The **Subtraction Assignment** Operator is used to subtract the right-hand side operand from the left-hand side operand and then assigning the result to the left-hand side operand.

The **Multiplication Assignment** Operator is used to multiply the right-hand side operand with the left-hand side operand and then assigning the result to the left-hand side operand.

The **Division Assignment** Operator is used to divide the left-hand side operand with the right-hand side operand and then assigning the result to the left operand.

The **Modulus Assignment** Operator is used to take the modulus, that is, it first divides the operands and then takes the remainder and assigns it to the left operand.

Code	Output

`a = 3`
`b = 5`
`# a = a + b`
`a += b`
`# Output`
`print("After Addition Assignment a= ", a)`
`# a = a - b`
`a -= b`
`# Output`
`print("After Subtraction Assignment a= ", a)`
`# a = a * b`
`a *= b`
`# Output`
`print("After Multiplication Assignment a= ", a)`
`# a = a / b`
`a /= b`
`# Output`
`print("After Division Assignment a= ", a)`
`# a = a % b`
`a %= b`
`# Output`
`print("After Modulus Assignment a=", a)`
	`After Addition Assignment a= 8`
	`After Subtraction Assignment a= 3`
	`After Multiplication Assignment a= 15`
	`After Division Assignment a= 3.0`
	`After Modulus Assignment a= 3.0`

3. Python Data Types

Data types in Python are a way to classify data items. They represent the kind of value, which determines what operations can be performed on that data. Since everything is an object in Python programming, Python data types are classes and variables are instances (objects) of these classes. The following are standard or built-in data types in Python as shown in figure 1:

- Numeric: Integer, Float, Complex number
- Sequence Type: String, List, Tuple
- Set
- Dictionary
- Boolean
- Binary

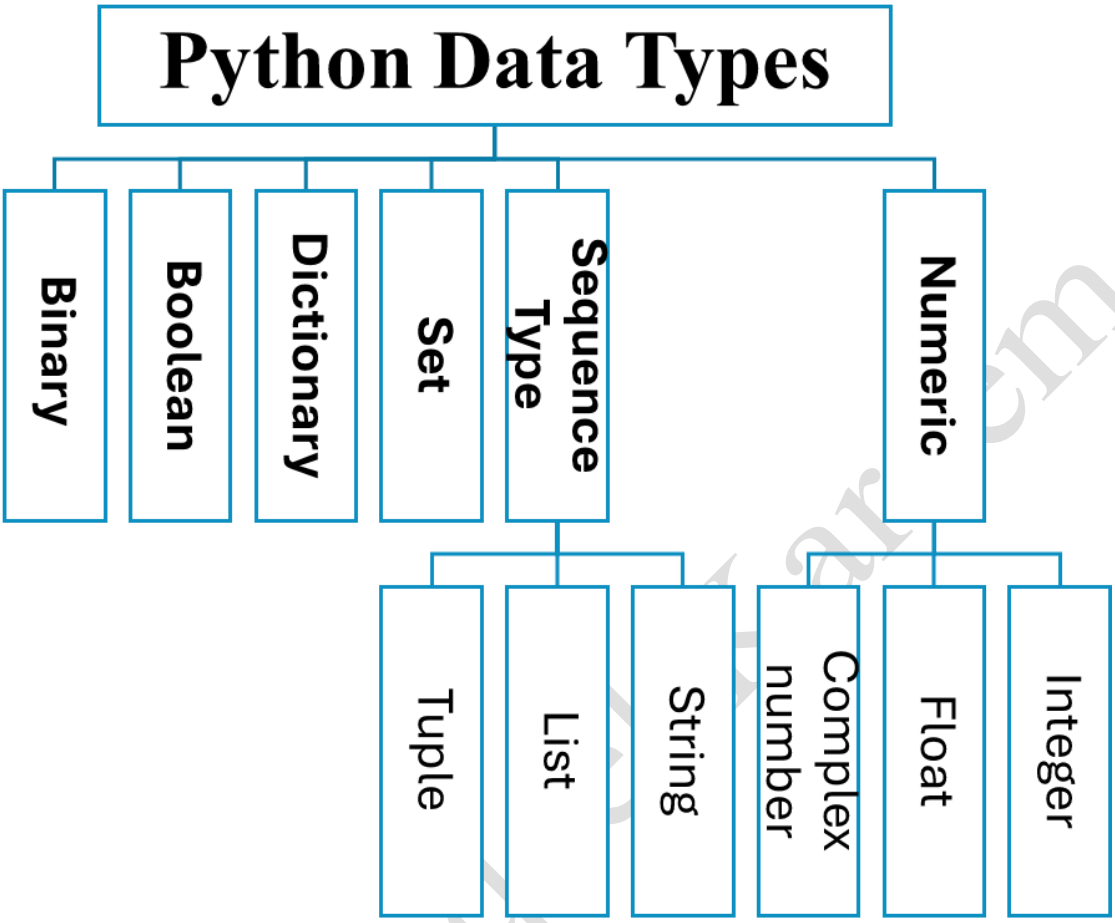


Figure 1: Python Data Types

3.1. Numeric Data Types

Python numbers represent data that has a numeric value. A numeric value can be an integer, a floating number or even a complex number. These values are defined as int, float and complex classes.

- Integers: value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). There is no limit to how long an integer value can be.
- Float: value is represented by float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- Complex Numbers: It is represented by a complex class. It is specified as (real part) + (imaginary part)j. For example - `2+3j`.

Code	Output
a = 5 print(type(a)) b = 5.0 print(type(b)) c = 2 + 4j print(type(c))	<class 'int'> <class 'float'> <class 'complex'>

3.2. Sequence Data Types

A sequence is an ordered collection of items, which can be of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence data types of Python:

- **String Data Type** : Python Strings are arrays of bytes representing Unicode characters. In Python, there is no character data type, a character is a string of length one. It is represented by str class. Strings in Python can be created using single quotes, double quotes or even triple quotes. We can access individual characters of a String using index.

Code	Output

s = 'Welcome to the python World'
print(s)
# check data type 	
print(type(s))
# access string with index 	
print(s[1])
print(s[2])
print(s[-1])
	Welcome to the python World
	<class 'str'>
	e
	l
	d

- **List Data Type:** Lists are similar to arrays found in other languages. They are an ordered and mutable collection of items. It is very flexible as items in a list do not need to be of the same type. Lists in Python can be created by just placing sequence inside the square brackets[].

Code	Output

# Empty list 	
a = []
# list with int values 	
a = [1, 2, 3]
print(a)
# list with mixed values int and String 	
b = ["abbas", "aqeel", "kareem", 4, 5]
print(b)
	[1, 2, 3]
	['abbas', 'aqeel', 'kareem', 4, 5]

- **Tuple Data Type:** Tuple is an ordered collection of Python objects. The only difference between a tuple and a list is that tuples are immutable. Tuples cannot be modified after it is created.

Code	Output

# initiate empty tuple 	
tup1 = ()
tup2 = ('abbas', 'aqeel')
print("\nTuple with the use of String: ", tup2)
	Tuple with the use of String:
	('abbas', 'aqeel')

3.3. Boolean Data Type

Python Boolean Data type is one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true) and those equal to False are falsy (false). However non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false. It is denoted by class bool.

Code	Output
print(type(True)) print(type(False)) print(type(true))	<class 'bool'> <class 'bool'> Traceback (most recent call last): File "/basic.py", line 3, in <module> print(type(true)) ^^^^ NameError: name 'true' is not defined. Did you mean: 'True'?

3.4. Set Data Type

In Python Data Types, Set is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements. Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'. The type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

Code	Output

# initializing empty set 	
s1 = set()
s1 = set("abbasaqeelkareem")
print("Set with the use of String: ", s1)
s2 = set(["abbas", "aqeel", "kareem"])
print("Set with the use of List: ", s2)
	Set with the use of String: {'r', 's', 'k', 'l', 'b', 'm', 'q', 'e', 'a'}
	Set with the use of List:
	{'aqeel', 'abbas', 'kareem'}

3.5. Dictionary Data Type

A dictionary in Python is a collection of data values, used to store data values like a map, unlike other Python Data Types, a Dictionary holds a key: value pair. Key-value is provided in dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon : , whereas each key is separated by a 'comma'.

Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable. The dictionary can also be created by the built-in function dict().

Code	Output

`# initialize empty dictionary`
`d = {}`
`d = {1: 'abbas', 2: 'aqeel', 3: 'kareem'}`
`print(d)`
`# creating dictionary using dict() constructor`
`d1 = dict({1: 'abbas', 2: 'aqeel', 3: 'kareem'})`
`print(d1)`
	`{1: 'abbas', 2: 'aqeel', 3: 'kareem'}`
	`{1: 'abbas', 2: 'aqeel', 3: 'kareem'}`

4. Conditional Statements

Conditional statements in Python are used to execute certain blocks of code based on specific conditions. These statements help control the flow of a program, making it behave differently in different situations.

4.1. If Conditional Statement

If statement is the simplest form of a conditional statement. It executes a block of code if the given condition is true.

Code	Output

`age = 20`
`if age >= 18:`
`  print("Eligible to vote.")`
	`Eligible to vote.`

4.2. If else Conditional Statement

`If Else` allows us to specify a block of code that will execute if the condition(s) associated with an `if` or `elif` statement evaluates to False. Else block provides a way to handle all other cases that don't meet the specified conditions.

Code	Output

`age = 10`
`if age <= 12:`
`  print("Travel for free.")`
`else:`
`  print("Pay for ticket.")`
	`Travel for free.`

4.3. elif Statement

elif statement in Python stands for "else if." It allows us to check multiple conditions, providing a way to execute different blocks of code based on which condition is true. Using elif statements makes our code more readable and efficient by eliminating the need for multiple nested if statements.

Code	Output

age = 25
if age <= 12:
    print("Child.")
elif age <= 19:
    print("Teenager.")
elif age <= 35:
    print("Young adult.")
else:
    print("Adult.")
	Young adult.

4.4. Nested if..else Conditional Statement

Nested if..else means an if-else statement inside another if statement. We can use nested if statements to check conditions within conditions.

Code	Output

age = 70
is_member = True
if age >= 60:
    if is_member:
        print("30% senior discount!")
    else:
        print("20% senior discount.")
else:
    print("Not eligible for a senior discount.")
	30% senior discount!

4.5. Match-Case Statement

match-case statement is Python's version of a switch-case found in other languages. It allows us to match a variable's value against a set of patterns.

Code	Output

number = 2
 	
match number:
    case 1:
        print("One")
    case 2 | 3:
        print("Two or Three")
    case _:
        print("Other number")
	Two or Three

5. Loops

Loops in Python are used to repeat actions efficiently. The main types are For loops (counting through items) and While loops (based on conditions). In this article, we will look at Python loops and understand their working with the help of examples.

5.1. For Loop

For loops is used to iterate over a sequence such as a list, tuple, string or range. It allow to execute a block of code repeatedly, once for each item in the sequence.

Code	Output

`n = 4`
`for i in range(0, n):`
`  print(i)`
	`0`
	`1`
	`2`
	`3`

Using else Statement with for Loop: We can also combine else statement with for loop like in while loop. But as there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.

Code	Output

`li = ["abbas", "aqeel", "kareem"]`
`for index in range(len(li)):`
`  print(li[index])`
`else:`
`  print("Inside Else Block")`
	`abbas`
	`aqeel`
	`kareem`
	`Inside Else Block`

5.2. While Loop

In Python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed. All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements. In below code, loop runs as long as the condition cnt < 3 is true. It increments the counter by 1 on each iteration and prints "Hello Geek" three times.

Code	Output

`cnt = 0`
`while (cnt < 3):`
`  cnt = cnt + 1`
`  print("Hello")`
	`Hello`
	`Hello`
	`Hello`

Using else Statement with for Loop : Else clause is only executed when our while condition becomes false. If we break out of the loop or if an exception is raised then it won't be executed.

Code	Output

`cnt = 0`
`while (cnt < 3):`
`  cnt = cnt + 1`
`  print("Hello")`
`else:`
`  print("In Else Block")`
	`Hello`
	`Hello`
	`Hello`
	`In Else Block`

Infinite While Loop: If we want a block of code to execute infinite number of times then we can use the while loop in Python to do so.

Code
`while (True):` `  print("Hello Geek")`

5.3. Nested Loops

Python programming language allows to use one loop inside another loop which is called nested loop. Following section shows few examples to illustrate the concept.

Code	Output

`for i in range(1, 5):`
`  for j in range(i):`
`  print(i, end=' ')`
`  print()`
	1
	2 2
	3 3 3
	4 4 4 4

5.4. Loop Control Statements

Loop control statements change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

- **Continue Statement:** The continue statement in Python returns the control to the beginning of the loop.

Code	Output

`for letter in 'abbasaqeel':`
`  if letter == 'e' or letter == 's':`
`  continue`
`  print('Current Letter :', letter)`
	Current Letter : a
	Current Letter : b
	Current Letter : b
	Current Letter : a
	Current Letter : a
	Current Letter : q
	Current Letter : l

- **Break Statement:** The break statement in Python brings control out of the loop.

Code	Output

`for letter in 'abbasaqeel':`
`  if letter == 'e' or letter == 's':`
`  break`
`print('Current Letter :', letter)`
	Current Letter : s

- **Pass Statement:** We use pass statement in Python to write empty loops. Pass is also used for empty control statements, functions and classes.

Code	Output

`for letter in 'abbasaqeel':`
`  pass`
`print('Last Letter :', letter)`
	`Last Letter : l`

6. String

In Python, a string is a sequence of characters enclosed in quotes. It can include letters, numbers, symbols or spaces. Since Python has no separate character type, even a single character is treated as a string with length one. Strings are widely used for text handling and manipulation.

6.1. Creating a String

Strings can be created using either single ('...') or double ("...") quotes. Both behave the same.

Code	Output

`s1 = 'abbas'  # single quote`
`s2 = "abbas"  # double quote`
`print(s1)`
`print(s2)`
	`abbas`
	`abbas`

6.2. Multi-line Strings

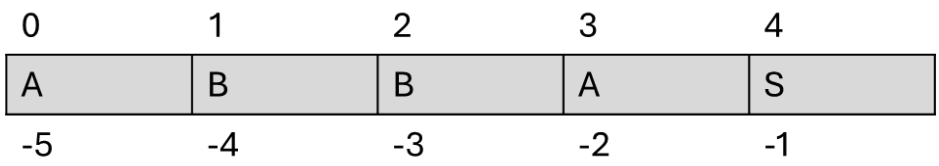
Use triple quotes ('''...''') or ("""...""") for strings that span multiple lines. Newlines are preserved.

Code	Output

`s = """I am Learning`
`Python String on EECT"""`
`print(s)`
`s = '''I'm a`
`student'''`
`print(s)`
	`I am Learning`
	`Python String on EECT`
	`I'm a`
	`student`

6.3. Accessing characters in String

Strings are indexed sequences. Positive indices start at 0 from the left; negative indices start at -1 from the right as represented in below image:



Code	Output

`s = "abbas"`
`print(s[0])  # first character`
`print(s[4])  # 5th character`
`print(s[-1])  # last character`
`print(s[-5])  # 5th character from end`
	`a`
	`s`
	`s`
	`a`

6.4.String Slicing

Slicing is a way to extract a portion of a string by specifying the start and end indexes. The syntax for slicing is string[start:end], where start starting index and end is stopping index (excluded).

Code	Output

`s = "abbasaqeel"`
`print(s[1:4])  # characters from index 1 to 3`
`print(s[:3])  # from start to index 2`
`print(s[3:])  # from index 3 to end`
`print(s[::-1])  # reverse string`
	`bba`
	`abb`
	`asaqeel`
	`leeqasabba`

6.5.String Iteration

Strings are iterable; you can loop through characters one by one.

Code	Output

`s = "Python"`
`for char in s:`
`  print(char)`
	`P`
	`y`
	`t`
	`h`
	`o`
	`n`

6.6.Deleting a String

In Python, it is not possible to delete individual characters from a string since strings are immutable. However, we can delete an entire string variable using the del keyword.

Code
`s = "abbas"` `del s`

6.7.Updating a String

As strings are immutable, “updates” create new strings using slicing or methods such as replace().

Code	Output

`s = "abbasaqeel"`
`s1 = "a" + s[1:]`
`s2 = s.replace("abbas", "Abbaskareem")`
`print(s1)`
`print(s2)`
	`abbasaqeel`
	`Abbaskareemaqeel`

6.8.Common String Methods

Python provides various built-in methods to manipulate strings. Below are some of the most useful methods:

1. len():

The len() function returns the total number of characters in a string (including spaces and punctuation).

Code	Output
`s = "abbasaqeelkareem"` `print(len(s))`	`16`

2. upper () and lower ()

upper() method converts all characters to uppercase whereas, lower() method converts all characters to lowercase.

Code	Output

`s = "Hello World"`
`print(s.upper())`
`print(s.lower())`
	`HELLO WORLD`
	`hello world`

3. strip () and replace ()

strip() removes leading and trailing whitespace from the string and replace() replaces all occurrences of a specified substring with another.

Code	Output

`s = "   aba   "`
`print(s.strip())`
`s = "Python is fun"`
`print(s.replace("fun", "awesome"))`
	`aba`
	`Python is awesome`

6.9.Concatenating and Repeating Strings

We can concatenate strings using + operator and repeat them using * operator.

Code	Output

`s1 = "Hello"`
`s2 = "World"`
`print(s1 + " " + s2)`
`s = "Hello "`
`print(s * 3)`
	`Hello World`
	`Hello Hello Hello`

6.10. Formatting Strings

Python provides several ways to include variables inside strings.

1. Using f-strings

The simplest and most preferred way to format strings is by using f-strings.

Code	Output

`name = "Alice"`
`age = 22`
`print(f"Name: {name}, Age: {age}")`
	`Name: Alice, Age: 22`

2. Using format()

Another way to format strings is by using format() method.

Code	Output

`name = "Alice"`
`age = 22`
`print(f"Name: {name}, Age: {age}")`
	`My name is Alice and I am 22 years old.`

6.11. String Membership Testing

`in` keyword checks if a particular substring is present in a string.

Code	Output

`s = "abbasaqeel"`
`print("abbas" in s)`
`print("kareem" in s)`
	`True`
	`False`

7. Lists

In Python, a list is a built-in data structure that can hold an ordered collection of items. Unlike arrays in some languages, Python lists are very flexible:

- Can contain duplicate items
- Mutable: items can be modified, replaced, or removed
- Ordered: maintains the order in which items are added
- Index-based: items are accessed using their position (starting from 0)
- Can store mixed data types (integers, strings, booleans, even other lists)

7.1. Creating a List

Lists can be created in several ways, such as using square brackets, the list() constructor or by repeating elements. Let's look at each method one by one with example:

1. Using Square Brackets

We use square brackets [] to create a list directly.

Code	Output

`a = [1, 2, 3, 4, 5] # List of integers`
`b = ['apple', 'banana', 'cherry'] # List of strings`
`c = [1, 'hello', 3.14, True] # Mixed data types`
`print(a)`
`print(b)`
`print(c)`
	`[1, 2, 3, 4, 5]`
	`['apple', 'banana', 'cherry']`
	`[1, 'hello', 3.14, True]`

2. Using list() Constructor

We can also create a list by passing an iterable (like a tuple, string or another list) to the list() function.

Code	Output

`a = list((1, 2, 3, 'apple', 4.5))`
`print(a)`
`b = list("aba")`
`print(b)`
	`[1, 2, 3, 'apple', 4.5]`
	`['a', 'b', 'a']`

3. Creating List with Repeated Elements

We can use the multiplication operator * to create a list with repeated items.

Code	Output

`a = [2] * 5`
`b = [0] * 7`
`print(a)`
`print(b)`
	`[2, 2, 2, 2, 2]`
	`[0, 0, 0, 0, 0, 0, 0]`

7.2.Accessing List Elements

Elements in a list are accessed using indexing. Python indexes start at 0, so a[0] gives the first element. Negative indexes allow access from the end (e.g., -1 gives the last element).

Code	Output
a = [10, 20, 30, 40, 50] print(a[0]) print(a[-1]) print(a[1:4]) # elements from index 1 to 3	10 50 [20, 30, 40]

7.3. Adding Elements into List

We can add elements to a list using the following methods:

- append(): Adds an element at the end of the list.
- extend(): Adds multiple elements to the end of the list.
- insert(): Adds an element at a specific position.
- clear(): removes all items.

Code	Output
a = [] a.append(10) print("After append(10):", a) a.insert(0, 5) print("After insert(0, 5):", a) a.extend([15, 20, 25]) print("After extend([15, 20, 25]):", a) a.clear() print("After clear():", a)	After append(10): [10] After insert(0, 5): [5, 10] After extend([15, 20, 25]): [5, 10, 15, 20, 25] After clear(): []

7.4. Updating Elements into List

Since lists are mutable, we can update elements by accessing them via their index.

Code	Output
a = [10, 20, 30, 40, 50] a[1] = 25 print(a)	[10, 25, 30, 40, 50]

7.5. Removing Elements from List

We can remove elements from a list using:

- remove(): Removes the first occurrence of an element.
- pop(): Removes the element at a specific index or the last element if no index is specified.
- del statement: Deletes an element at a specified index.

Code	Output
a = [10, 20, 30, 40, 50] a.remove(30) print("After remove(30):", a) popped_val = a.pop(1) print("Popped element:", popped_val) print("After pop(1):", a) del a[0] print("After del a[0]:", a)	After remove(30): [10, 20, 40, 50] Popped element: 20 After pop(1): [10, 40, 50] After del a[0]: [40, 50]

7.6.Iterating Over Lists

We can iterate over lists using loops, which is useful for performing actions on each item.

Code	Output
a = ['apple', 'banana', 'cherry'] for item in a: print(item)	apple banana cherry

7.7.Nested Lists

A nested list is a list within another list, which is useful for representing matrices or tables. We can access nested elements by chaining indexes.

Code	Output

matrix = [[1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]]
print(matrix[1][2])
	6

8. Tuples

A tuple in Python is an immutable ordered collection of elements.

- Tuples are similar to lists, but unlike lists, they cannot be changed after their creation (i.e., they are immutable).
- Tuples can hold elements of different data types.
- The main characteristics of tuples are being ordered, heterogeneous and immutable.

8.1.Creating a Tuple

A tuple is created by placing all the items inside parentheses (), separated by commas. A tuple can have any number of items and they can be of different data types.

Code	Output

`tup = ()`
`print(tup)`
`# Using String`
`tup = ('Geeks', 'For')`
`print(tup)`
`# Using List`
`li = [1, 2, 4, 5, 6]`
`print(tuple(li))`
`# Using Built-in Function`
`tup = tuple('Geeks')`
`print(tup)`
	`()`
	`('Geeks', 'For')`
	`(1, 2, 4, 5, 6)`
	`('G', 'e', 'e', 'k', 's')`

8.2. Creating a Tuple with Mixed Datatypes

Tuples can contain elements of various data types, including other tuples, lists, dictionaries and even functions.

Code	Output

`tup = (5, 'Welcome', 7, 'python')`
`print(tup)`
`# Creating a Tuple with nested tuples`
`tup1 = (0, 1, 2, 3)`
`tup2 = ('python', 'fun')`
`tup3 = (tup1, tup2)`
`print(tup3)`
`# Creating a Tuple with repetition`
`tup1 = ('python',) * 3`
`print(tup1)`
`# Creating a Tuple with the use of loop`
`tup = ('python')`
`n = 5`
`for i in range(int(n)):`
`  tup = (tup,)`
`  print(tup)`
	`(5, 'Welcome', 7, 'python')`
	`((0, 1, 2, 3), ('python', 'fun'))`
	`('python', 'python', 'python')`
	`('python',)`
	`(('python',),)`
	`((('python',),),)`
	`((('python',),),),)`
	`(((('python',),),),),)`

8.3. Tuple Basic Operations

1. Accessing of Tuples

We can access the elements of a tuple by using indexing and slicing, similar to how we access elements in a list. Indexing starts at 0 for the first element and goes up to n-1, where n is the number of elements in the tuple. Negative indexing starts from -1 for the last element and goes backward.

Code	Output

# Accessing Tuple with Indexing 	
tup = tuple("python")
print(tup[0])
# Accessing a range of elements using slicing 	
print(tup[1:4])
print(tup[:3])
# Tuple unpacking 	
tup = ("python", "For", "students")
# This line unpack values of Tuple1 	
a, b, c = tup
print(a)
print(b)
print(c)
	p
	('y', 't', 'h')
	('p', 'y', 't')
	python
	For
	students

2. Concatenation of Tuples

Tuples can be concatenated using the + operator. This operation combines two or more tuples to create a new tuple.

Code	Output
tup1 = (0, 1, 2, 3) tup2 = ('Python', 'For', 'Student') tup3 = tup1 + tup2 print(tup3)	(0, 1, 2, 3, 'Python', 'For', 'Student')

3. Slicing of Tuple

Slicing a tuple means creating a new tuple from a subset of elements of the original tuple. The slicing syntax is tuple[start:stop:step].

Code	Output

tup = tuple('abbasaqeel')
# Removing First element 	
print(tup[1:])
# Reversing the Tuple 	
print(tup[::-1])
# Printing elements of a Range 	
print(tup[4:9])
	('b', 'b', 'a', 's', 'a', 'q', 'e', 'e', 'l')
	('l', 'e', 'e', 'q', 'a', 's', 'a', 'b', 'b', 'a')
	('s', 'a', 'q', 'e', 'e')

4. Deleting a Tuple

Since tuples are immutable, we cannot delete individual elements of a tuple. However, we can delete an entire tuple using del statement.

Code
tup = (0, 1, 2, 3, 4) del tup

5. Tuple Unpacking with Asterisk (*)

In Python, the " * " operator can be used in tuple unpacking to grab multiple items into a list. This is useful when you want to extract just a few specific elements and collect the rest together.

Code	Output

`tup= (1, 2, 3, 4, 5)`
`a, *b, c = tup`
`print(a)`
`print(b)`
`print(c)`
	`1`
	`[2, 3, 4]`
	`5`

9. Dictionary

Python dictionary is a data structure that stores the value in key: value pairs. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable.

- Keys are case sensitive which means same name but different cases of Key will be treated distinctly.
- Keys must be immutable which means keys can be strings, numbers or tuples but not lists.
- Duplicate keys are not allowed and any duplicate key will overwrite the previous value.
- Internally uses hashing. Hence, operations like search, insert, delete can be performed in Constant Time.
- From Python 3.7 Version onward, Python dictionary are Ordered.

9.1.Create a Dictionary

Dictionary can be created by placing a sequence of elements within curly {} braces, separated by a 'comma'.

Code	Output

`d1 = {1: 'python', 2: 'For', 3: 'cyber'}`
`print(d1)`
`# create dictionary using dict() constructor`
`d2 = dict(a = "python", b = "for", c = "cyber")`
`print(d2)`
	`{1: 'python', 2: 'For', 3: 'cyber'}`
	`{'a': 'python', 'b': 'for', 'c': 'cyber'}`

9.2.Accessing Dictionary Items

We can access a value from a dictionary by using the key within square brackets or get() method.

Code	Output

d = { "name": "Prajjwal", 1: "Python", (1, 2): [1,2,4] }
# Access using key 	
print(d["name"])
# Access using get() 	
print(d.get("name"))
	Prajjwal
	Prajjwal

9.3. Adding and Updating Dictionary Items

We can add new key-value pairs or update existing keys by using assignment.

Code	Output

d = {1: 'python', 2: 'For', 3: 'cyber'}
# Adding a new key-value pair 	
d["age"] = 22
# Updating an existing value 	
d[1] = "Python dict"
print(d)
	{1: 'Python dict', 2: 'For', 3: 'cyber', 'age': 22}

9.4. Removing Dictionary Items

We can remove items from dictionary using the following methods:

- del: Removes an item by key.
- pop(): Removes an item by key and returns its value.
- clear(): Empties the dictionary.
- popitem(): Removes and returns the last key-value pair.

Code	Output

d = {1: 'python', 2: 'For', 3: 'student', 'age':22}
# Using del to remove an item 	
del d["age"]
print(d)
# Using pop() to remove an item and return the value 	
val = d.pop(1)
print(val)
# Using popitem to removes and returns 	
# the last key-value pair. 	
key, val = d.popitem()
print(f"Key: {key}, Value: {val}")
# Clear all items from the dictionary 	
d.clear()
print(d)
	{1: 'python', 2: 'For', 3: 'student'}
	python
	Key: 3, Value: student
	{}

9.5.Iterating Through a Dictionary

We can iterate over keys [using keys() method] , values [using values() method] or both [using item() method] with a for loop.

Code	Output

`d = {1: 'python', 2: 'For', 3: 'student', 'age':22}`
`# Using del to remove an item`
`del d["age"]`
`print(d)`
`# Using pop() to remove an item and return the value`
`val = d.pop(1)`
`print(val)`
`# Using popitem to removes and returns`
`# the last key-value pair.`
`key, val = d.popitem()`
`print(f"Key: {key}, Value: {val}")`
`# Clear all items from the dictionary`
`d.clear()`
`print(d)`
	`{1: 'python', 2: 'For', 3: 'student'}`
	`python`
	`Key: 3, Value: student`
	`{}`

10.Sets

Python set is an unordered collection of multiple items having different datatypes. In Python, sets are mutable, unindexed and do not contain duplicates. The order of elements in a set is not preserved and can change.

- Can store None values.
- Implemented using hash tables internally.
- Do not implement interfaces like Serializable or Cloneable.
- Python sets are not inherently thread-safe; synchronization is needed if used across threads.

10.1. Creating a Set

In Python, the most basic and efficient method for creating a set is using curly braces. Also Python Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'.

Code	Output

set1 = {1, 2, 3, 4}
print(set1)
set1 = set()
print(set1)
set1 = set("PythonforCyber")
print(set1)
# Creating a Set with the use of a List 	
set1 = set(["Python", "For", "Cyber"])
print(set1)
# Creating a Set with the use of a tuple 	
tup = ("Python", "for", "Cyber")
print(set(tup))
# Creating a Set with the use of a dictionary 	
d = {"Python": 1, "for": 2, "Cyber": 3}
print(set(d))
	{1, 2, 3, 4}
	set()
	{'t', 'n', 'f', 'y', 'r', 'e', 'P', 'h', 'b', 'o', 'C'}
	{'Python', 'For', 'Cyber'}
	{'Python', 'for', 'Cyber'}
	{'Python', 'for', 'Cyber'}

10.2. Adding Elements to a Set

We can add items to a set using add() method and update() method. add() method can be used to add only a single item. To add multiple items we use update() method.

Code	Output

# Creating a set 	
set1 = {1, 2, 3}
# Add one item 	
set1.add(4)
# Add multiple items 	
set1.update([5, 6])
print(set1)
	{1, 2, 3, 4, 5, 6}

10.3. Accessing a Set

We can loop through a set to access set items as set is unindexed and do not support accessing elements by indexing. Also we can use in keyword which is membership operator to check if an item exists in a set.

Code	Output

set1 = set(["Python", "For", "Cyber."])
# Accessing element using For loop 	
for i in set1:
  print(i, end=" ")
# Checking the element# using in keyword 	
print("Python" in set1)
	Python For Cyber. True

10.4. Removing Elements from the Set

We can remove an element from a set in Python using several methods: remove(), discard() and pop(). Each method works slightly differently:

- Using **remove() Method or discard() Method**: remove() method removes a specified element from the set. If the element is not present in the set, it raises a KeyError. discard() method also removes a specified element from the set. Unlike remove(), if the element is not found, it does not raise an error.

Code	Output

`# Using Remove Method`
`set1 = {1, 2, 3, 4, 5}`
`set1.remove(3)`
`print(set1)`
`# Attempting to remove an element that does not exist`
`try:`
`  set1.remove(10)`
`except KeyError as e:`
`  print("Error:", e)`
`# Using discard() Method`
`set1.discard(4)`
`print(set1)`
`# Attempting to discard an element that does not exist`
`set1.discard(10)  # No error raised`
`print(set1)`
	`{1, 2, 4, 5}`
	`Error: 10`
	`{1, 2, 5}`
	`{1, 2, 5}`

- Using **pop() Method:** pop() method removes and returns an arbitrary element from the set. This means we don't know which element will be removed. If the set is empty, it raises a KeyError.

Code	Output

`set1 = {1, 2, 3, 4, 5}`
`val = set1.pop()`
`print(val)`
`print(set1)`
`# Using pop on an empty set`
`set1.clear()  # Clear the set to make it empty`
`try:`
`  set1.pop()`
`except KeyError as e:`
`  print("Error:", e)`
	`1`
	`{2, 3, 4, 5}`
	`Error: 'pop from an empty set'`

- Using **clear() Method**: clear() method removes all elements from the set, leaving it empty.

Code	Output
set1 = {1, 2, 3, 4, 5} set1.clear() print(set1)	set()

10.5. Frozen Sets

A frozenset in Python is a built-in data type that is similar to a set but with one key difference that is immutability. This means that once a frozenset is created, we cannot modify its elements that is we cannot add, remove or change any items in it. Like regular sets, a frozenset cannot contain duplicate elements.

Code	Output

# Creating a frozenset from a list 	
fset = frozenset([1, 2, 3, 4, 5])
print(fset)
# Creating a frozenset from a set 	
set1 = {3, 1, 4, 1, 5}
fset = frozenset(set1)
print(fset)
	frozenset({1, 2, 3, 4, 5})
	frozenset({1, 3, 4, 5})

11.Arrays

In Python, array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. Unlike Python lists (can store elements of mixed types), arrays must have all elements of same type. Having only homogeneous elements makes it memory-efficient.

Code	Output

import array as arr
a = arr.array('i', [1, 2, 3])
# accessing First Araay 	
print(a[0])
# adding element to array 	
a.append(5)
print(a)
	1
	array('i', [1, 2, 3, 5])

11.1. Create an Array

Array in Python can be created by importing an array module. array(data_type , value_list) is used to create array in Python with data type and value list specified in its arguments.

Code	Output
import array as arr a = arr.array('i', [1, 2, 3]) for i in range(0, 3): print(a[i], end=" ")	1 2 3

11.2. Adding Elements to an Array

Elements can be added to the Python Array by using built-in insert() function. Insert is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. append() is also used to add the value mentioned in its arguments at the end of the Python array.

Code	Output

`import array as arr`
`a = arr.array('i', [1, 2, 3])`
`print(*a)`
`a.insert(1, 4)  # Insert 4 at index 1`
`print(*a)`
	`1 2 3`
	`1 4 2 3`

11.3. Accessing Array Items

In order to access the array items refer to the index number. Use the index operator [ ] to access an item in a array in Python. The index must be an integer.

Code	Output

`import array as arr`
`a = arr.array('i', [1, 2, 3, 4, 5, 6])`
`print(a[0])`
`print(a[3])`
`b = arr.array('d', [2.5, 3.2, 3.3])`
`print(b[1])`
`print(b[2])`
	`1`
	`4`
	`3.2`
	`3.3`

11.4. Removing Elements from the Array

Elements can be removed from the Python array by using built-in remove() function. It will raise an Error if element doesn't exist. Remove() method only removes the first occurrence of the searched element. To remove range of elements, we can use an iterator. pop() function can also be used to remove and return an element from the array. By default it removes only the last element of the array. To remove element from a specific position, index of that item is passed as an argument to pop() method.

Code	Output

`import array`
`a = array.array('i', [1, 2, 3, 1, 5])`
`# remove first occurance of 1`
`a.remove(1)`
`print(a)`
`# remove item at index 2`
`a.pop(2)`
`print(a)`
	`array('i', [2, 3, 1, 5])`
	`array('i', [2, 3, 5])`

11.5. Slicing of an Array

In Python array, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use Slice operation:

- Elements from beginning to a range use [:Index]
- Elements from end use [:-Index]
- Elements from specific Index till the end use [Index:]
- Elements within a range, use [Start Index:End Index]
- Print complete List, use [:].
- For Reverse list, use [::-1].

Code	Output

`import array as arr`
`a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
`b = arr.array('i', a)`
`res = a[3:8]`
`print(res)`
`res = a[5:]`
`print(res)`
`res = a[:]`
`print(res)`
	`[4, 5, 6, 7, 8]`
	`[6, 7, 8, 9, 10]`
	`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

11.6. Searching Element in an Array

In order to search an element in the array we use a python in-built index() method. This function returns the index of the first occurrence of value mentioned in arguments.

Code	Output

`import array`
`a = array.array('i', [1, 2, 3, 1, 2, 5])`
`# index of 1st occurrence of 2`
`print(a.index(2))`
`# index of 1st occurrence of 1`
`print(a.index(1))`
	`1`
	`0`

11.7. Updating Elements in an Array

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

Code	Output

import array
a = array.array('i', [1, 2, 3, 1, 2, 5])
# update item at index 2 	
a[2] = 6
print(a)
# update item at index 4 	
a[4] = 8
print(a)
	array('i', [1, 2, 6, 1, 2, 5])
	array('i', [1, 2, 6, 1, 8, 5])

11.8. Different Operations on Python Arrays

1. Counting Elements in an Array

We can use count() method to count given item in array.

Code	Output
import array a = array.array('i', [1, 2, 3, 4, 2, 5, 2]) count = a.count(2) print(count)	3

2. Reversing Elements in an Array

In order to reverse elements of an array we need to simply use reverse method.

Code	Output
import array a = array.array('i', [1, 2, 3, 4, 5]) a.reverse() print(*a)	5 4 3 2 1

3. Extend Element from Array

In Python, an array is used to store multiple values or elements of the same datatype in a single variable. The extend() function is simply used to attach an item from iterable to the end of the array. In simpler terms, this method is used to add an array of values to the end of a given or existing array.

Code	Output
import array as arr a = arr.array('i', [1, 2, 3,4,5]) a.extend([6,7,8,9,10]) print(a)	array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])