PYTHON

Python Built In Method

Python

Python Built In Method

No	Function	Description
1	abs()	Returns the non-negative value of a number, regardless of its sign.
2	all()	Returns True if every item in an iterable is true.
3	any()	Returns True if at least one item in an iterable is true.
4	ascii()	Converts an object to a string, replacing non-ASCII characters with escape sequences.
5	bin()	Converts a number to its binary representation.
6	bool()	Returns the boolean representation of a given object.
7	bytearray()	Creates an array of bytes from a sequence.
8	bytes()	Creates an immutable bytes object.
9	callable()	Checks if an object can be called like a function or method.
10	chr()	Converts a Unicode code point to its corresponding character.
11	classmethod()	Turns a method into a class method.
12	compile()	Compiles a source code string into a code object for execution.

No	Function	Description
13	complex()	Creates a complex number from real and imaginary parts.
14	delattr()	Removes a specified attribute from an object.
15	dict()	Creates a dictionary from key-value pairs.
16	dir()	Lists all attributes and methods of an object.
17	divmod()	Returns both the quotient and remainder when dividing two numbers.
18	enumerate()	Converts a collection into an enumerated object with index-value pairs.
19	eval()	Evaluates a string expression and returns the result.
20	exec()	Executes a given string or object as Python code.
21	filter()	Filters items in an iterable based on a function's criteria.
22	float()	Converts a number or string to a floating-point number.
23	format()	Formats a value into a specified string format.
24	frozenset()	Creates an immutable set from elements.
25	getattr()	Retrieves the value of a specified attribute from an object.
26	globals()	Returns the global symbol table as a dictionary.
27	hasattr()	Checks if an object has a specific attribute or method.
28	hash()	Returns the hash value of an object, useful for hash tables.
29	help()	Launches the interactive help system for Python.
30	hex()	Converts a number to its hexadecimal string representation.
31	id()	Returns the unique identifier of an object.
32	input()	Allows the user to input data from the keyboard.
33	int()	Converts a value to an integer.
34	isinstance()	Checks if an object is an instance of a specific class or type.
35	issubclass()	Checks if a class is a subclass of another class.
36	iter()	Converts an iterable into an iterator.

No	Function	Description
37	len()	Returns the number of items in an object.
38	list()	Creates a list from an iterable.
39	locals()	Returns the local symbol table as an updated dictionary.
40	map()	Applies a function to all items in an iterable and returns the result.
41	max()	Returns the largest item in an iterable.
42	memoryview()	Creates a memory view object that allows direct access to the internal data of a byte-based object without copying it.
43	min()	Returns the smallest item in an iterable.
44	next()	Returns the next item in an iterator.
45	object()	Creates a new object (base class).
46	oct()	Converts a number to its octal string representation.
47	open()	Opens a file and returns a file object for reading or writing.
48	ord()	Converts a character to its corresponding Unicode code point.
49	pow()	Computes x raised to the power of y.
50	print()	Outputs text or data to the console.
51	property()	Creates, gets, sets, or deletes an object's property.
52	range()	Generates a sequence of numbers, starting from 0 by default.
53	repr()	Returns a string representation of an object, suitable for debugging.
54	reversed()	Returns an iterator that yields items in reverse order.
55	round()	Rounds a number to a specified number of decimal places.
56	set()	Creates a new set object from an iterable.
57	setattr()	Sets a value for a specified attribute on an object.
58	slice()	Creates a slice object for extracting portions of iterables.
59	sorted()	Returns a sorted list from an iterable.
60	staticmethod()	Converts a method into a static method, bound to the class.

No	Function	Description
61	str()	Converts an object to its string representation.
62	sum()	Sums the elements of an iterable.
63	super()	Returns a proxy object representing the parent class.
64	tuple()	Creates a tuple from an iterable.
65	type()	Returns the type or class of an object.
66	vars()	Returns an object's `__dict__` attribute, showing its internal variables.
67	zip()	Combines multiple iterables into a single iterator of tuples.

1. abs()

Point	Description
Use Case	The `abs()` function returns the absolute (non-negative) value of a number. For complex numbers, it returns the magnitude.
Syntax	`abs(n)`
Parameter	`n` *(Required)* — A numeric value (integer, float, or complex number) for which the absolute value is to be calculated.

Useful Information

No.	Point	Description
1	Works with Different Number Types	Accepts integers, floats, and complex numbers.
2	Returns Non-Negative Value	Always returns a positive value or zero.
3	For Complex Numbers	Returns the magnitude: `abs(a + bj)` = $\sqrt{(a^2 + b^2)}$.

```
# Example 1: Absolute value of a complex number
x = abs(1 + 2j)
```

```
print(x)
# Output: 2.23606797749979
# Explanation: sqrt(1**2 + 2**2) = sqrt(1 + 4) = sqrt(5)

# Example 2: Absolute value of a negative float
x = abs(-2.50)
print(x)
# Output: 2.5
# Explanation: Converts negative number to positive
```

```
2.23606797749979
2.5
```

2. all()

Point	Description
Use Case	The `all()` function checks whether **every element** in an iterable evaluates to `True`. It is often used to validate data integrity or check uniform conditions across datasets.
Syntax	`all(iterable)`
Parameter	`iterable` *(Required)* — Any iterable (e.g., list, tuple, set, or dictionary) whose elements are to be evaluated as Boolean values.

Useful Information

No.	Point	Description
1	Evaluates Truthiness	Returns `True` only if all items are truthy (non-zero, non-empty, etc.).
2	Returns `True` for Empty Iterables	If the iterable is empty, the function returns `True` by default.
3	Works Across Iterable Types	Compatible with lists, tuples, sets, and dictionaries.

No.	Point	Description
4	Evaluates Dictionary Keys Only	When applied to a dictionary, it evaluates keys—not values.

```python
# Example 1: Check if all feature values are normalized (between 0 and 1)
features = [0.8, 0.5, 0.95, 1.0]
are_all_normalized = all(0 <= x <= 1 for x in features)
print(are_all_normalized)
# Output: True
# Explanation: All values fall within the 0-1 normalization range

# Example 2: Check if all model predictions are greater than a threshold
predictions = {0.92, 0.87, 0.89, 0.95}
threshold = 0.85
is_confident = all(p > threshold for p in predictions)
print(is_confident)
# Output: True
# Explanation: Every prediction exceeds the confidence threshold

# Example 3: Check for missing values in dataset (0 represents missing)
data_row = [1, 1, 0, True]
is_valid_row = all(data_row)
print(is_valid_row)
# Output: False
# Explanation: The row contains a missing or invalid (0) value

# Example 4: Dictionary with feature presence (keys)
features_present = {1: "age", 2: "income", 0: "education"}
check_all_present = all(features_present)
print(check_all_present)
# Output: False
# Explanation: Key '0' evaluates to False, so not all features are
present

# Example 5: Empty input vector
empty_vector = ()
check_empty = all(empty_vector)
print(check_empty)
# Output: True
# Explanation: By definition, all() returns True for empty iterables
```

```
True
True
False
False
True
```

3. any()

Point	Description
Use Case	The `any()` function checks whether **at least one element** in an iterable is `True`. It returns `True` as soon as it finds the first truthy value; otherwise, it returns `False`.
Syntax	`any(iterable)`
Parameter	`iterable` *(Required)* — An iterable like a list, tuple, set, or dictionary containing elements to evaluate.

Useful Information

No.	Point	Description
1	Returns on First True Value	Stops checking once a single `True` or truthy value is found.
2	Returns `False` for Empty Iterables	If the iterable is empty, `any()` returns `False`.
3	Accepts Multiple Iterable Types	Works with lists, tuples, sets, and dictionaries.
4	Checks Dictionary Keys Only	When applied to a dictionary, it evaluates the truthiness of keys—not values.

```python
# Example 1: Check if any input features are missing (0 indicates
missing)
input_vector = [1, 0, 1, 1]
has_missing = any(x == 0 for x in input_vector)
```

```python
print(has_missing)
# Output: True
# Explanation: At least one feature is marked as missing (0)

# Example 2: Identify if any prediction scores fall below a safe
threshold
model_outputs = [0.92, 0.88, 0.76, 0.95]
threshold = 0.80
any_low_confidence = any(score < threshold for score in model_outputs)
print(any_low_confidence)
# Output: True
# Explanation: One of the predictions is below the defined threshold

# Example 3: Dictionary with encoded feature flags (keys)
feature_flags = {0: "inactive", 1: "active"}
any_active_flag = any(feature_flags)
print(any_active_flag)
# Output: True
# Explanation: Key '1' is truthy, so `any()` returns True

# Example 4: Check if any model evaluation metric is perfect
metrics = (0.9, 1.0, 0.85)
has_perfect_score = any(m == 1.0 for m in metrics)
print(has_perfect_score)
# Output: True
# Explanation: A perfect score (1.0) exists in the tuple

# Example 5: Empty input check
empty_input = {}
x = any(empty_input)
print(x)
# Output: False
# Explanation: No elements to evaluate, so returns False
```

```
True
True
True
True
False
```

4. ascii()

Point	Description
Use Case	The `ascii()` function returns a string representation of an object, where **non-ASCII characters** are escaped using Unicode code points. It ensures the result is readable in ASCII-only environments.
Syntax	`ascii(object)`
Parameter	`object` *(Required)* — Any object such as a string, list, tuple, or dictionary that needs to be represented in ASCII form.

Useful Information

No.	Point	Description
1	Converts to ASCII-Safe Format	Escapes non-ASCII characters like `å` into `\u00e5` or similar.
2	Supports Various Data Types	Accepts strings, lists, dictionaries, tuples, etc.
3	Output is Always a String	Returns a string with all non-ASCII characters escaped.
4	Useful in Logging and Serialization	Especially helpful when saving or logging data in environments that only support ASCII characters.
5	Doesn't Modify the Original Object	Returns a new ASCII-safe string representation, leaving the original untouched.

```
# Example : String containing non-ASCII characters
text = "Model trained by Ståle on dataset µData"
ascii_output = ascii(text)
print(ascii_output)
# Output: 'Model trained by St\xe5le on dataset \xb5Data'
# Explanation: Non-ASCII characters like 'å' and 'µ' are escaped in
ASCII-safe form
```

```
'Model trained by St\xe5le on dataset \xb5Data'
```

5. bin()

Point	Description
Use Case	The `bin()` function converts a given integer into its **binary string representation**, prefixed with `0b`.
Syntax	`bin(number)`
Parameter	`number` *(Required)* — Any integer value you want to convert to binary.

Useful Information

No.	Point	Description
1	Returns a Binary String	Output is a string showing the binary equivalent of the integer.
2	Always Prefixed with `0b`	The result starts with `0b`, indicating a binary value.
3	Only Works with Integers	Passing a non-integer will raise a `TypeError`.
4	Useful in Bitwise Operations	Commonly used in scenarios involving low-level bit manipulation.
5	Output is Immutable	The binary value is returned as a new string object; the original integer remains unchanged.

```python
num = 36
binary_value = bin(num)

print(binary_value)
# Output: '0b100100'
# Explanation: The integer 36 is represented as '100100' in binary,
# and the '0b' prefix signifies that it's a binary number.
```

```
0b100100
```

6. bool()

Point	Description
Use Case	The `bool()` function is used to evaluate a value and return either `True` or `False` based on its truthiness.
Syntax	`bool(object)`
Parameter	`object` *(Optional)* — Any value such as string, number, list, or other object types. If no argument is given, it returns `False`.

Useful Information

No.	Point	Description
1	Evaluates Truthiness	Returns `True` for most non-empty or non-zero objects.
2	Returns `False` For Certain Values	Empty sequences (`[]`, `()`, `{}`), `0`, `None`, and `False` itself evaluate to `False`.
3	Converts to Boolean Type	Useful for conditional checks or logical operations.
4	Works with Any Object Type	Can be used with numbers, strings, lists, dictionaries, and even custom objects.
5	No Argument Returns `False`	Calling `bool()` with no arguments gives `False`.

```python
# Example 1: Non-zero integer
x = bool(1)
print(x)
# Output: True

# Example 2: Empty list
x = bool([])
print(x)
```

```
# Output: False

# Example 3: None value
x = bool(None)
print(x)
# Output: False
```

```
True
False
False
```

7. bytearray()

Point	Description
Use Case	The `bytearray()` function returns a **mutable sequence of bytes**, similar to a list of integers ranging from `0` to `255`. It's commonly used for binary data manipulation.
Syntax	`bytearray(source, encoding, errors)`
Parameter	`source` *(Optional)* — The data to convert into a bytearray (e.g., integer, string, list, or bytes).
	`encoding` *(Optional)* — Specifies the encoding if the source is a string.
	`errors` *(Optional)* — Defines how to handle encoding errors (e.g., `'strict'`, `'ignore'`).

Useful Information

No.	Point	Description
1	Mutable Alternative to `bytes`	Unlike `bytes()`, `bytearray()` allows modifying its contents.
2	Converts Strings, Lists, and More	Can convert strings (with encoding), lists of integers, or other byte-like objects.

No.	Point	Description
3	Initializes Empty Buffer	If an integer is passed, creates a zero-initialized bytearray of that length.
4	Supports Encoding and Error Handling	Useful when dealing with string-to-byte conversions.
5	Works Well for Binary I/O	Commonly used in file and network data manipulation.

```
# Example 1: Create a bytearray of length 4 (all bytes initialized to zero)
x = bytearray(4)
print(x)
# Output: bytearray(b'\x00\x00\x00\x00')
# Explanation: Returns a bytearray of 4 bytes, each initialized to zero.

# Example 2: Create a bytearray from a string using UTF-8 encoding
text = "intensity"
x = bytearray(text, "utf-8")
print(x)
# Output: bytearray(b'intensity')
# Explanation: Each character in the string is converted to its UTF-8 byte representation.

# Example 3: Modify a byte in the bytearray
x[0] = 73  # Replace 'i' (ASCII 105) with 'I' (ASCII 73)
print(x)
# Output: bytearray(b'Intensity')
```

```
bytearray(b'\x00\x00\x00\x00')
bytearray(b'intensity')
bytearray(b'Intensity')
```

8. bytes()

Point	Description
Use Case	The `bytes()` function returns an **immutable sequence of bytes**. It's used for handling binary data or converting data types (like strings or lists) into a bytes object.
Syntax	`bytes(source, encoding, errors)`
Parameter	`source` *(Optional)* — The input to convert into a bytes object (can be an integer, string, list, etc.).
	`encoding` *(Optional)* — The encoding to use if the source is a string.
	`errors` *(Optional)* — Specifies how to handle encoding errors (`'strict'`, `'ignore'`, etc.).

Useful Information

No.	Point	Description
1	Immutable Byte Sequence	Unlike `bytearray()`, the resulting object cannot be altered after creation.
2	Accepts Multiple Data Types	Works with integers, strings, lists of integers (0–255), or other byte-like objects.
3	Zero-Initialized Buffer	Passing an integer creates a bytes object of that length with all bytes set to `0`.
4	Requires Encoding for Strings	If a string is passed, encoding must be specified to convert it into bytes.
5	Common in Binary Operations	Often used for reading/writing files, socket communication, and serialization.

```
# Example 1: Create an empty bytes object of size 4
x = bytes(4)
print(x)
# Output: b'\x00\x00\x00\x00'
# Explanation: Generates 4 bytes initialized to zero (immutable).

# Example 2: Convert a string to bytes using UTF-8 encoding
text = "intensity"
x = bytes(text, "utf-8")
print(x)
```

```
# Output: b'intensity'

# Example 3: Convert a list of integers to a bytes object
x = bytes([73, 110, 116])
print(x)
# Output: b'Int'
# Explanation: The integers represent ASCII values of characters.
```

```
b'\x00\x00\x00\x00'
b'intensity'
b'Int'
```

9. callable()

Point	Description
Use Case	The `callable()` function checks whether the given object can be called like a function, meaning it can be invoked using parentheses `()`. It returns `True` if the object is callable, otherwise `False`.
Syntax	`callable(object)`
Parameter	`object` *(Required)* — Any Python object you want to test for callability (e.g., functions, classes, or objects with a `__call__()` method).

Useful Information

No.	Point	Description
1	Returns a Boolean	Always returns either `True` (if callable) or `False` (if not).
2	Functions are Callable	User-defined and built-in functions return `True`.
3	Integers, Strings, Lists are Not Callable	Regular data types like `int`, `str`, `list` return `False`.
4	Classes and Objects Can Be Callable	If a class implements the `__call__()` method, its instances are callable.

No.	Point	Description
5	Helpful for Type-Checking	Useful when designing APIs or decorators that accept functions or callable objects.

```python
# Example 1: Check if a user-defined function is callable
def example_function():
    return "This is callable!"

print(callable(example_function))
# Output: True
# Explanation: Functions can be called, so this returns True.

# Example 2: Check if an integer is callable
number = 10
print(callable(number))
# Output: False
# Explanation: Integers are not callable objects.

# Example 3: Check if a class instance is callable (with __call__)
class CallableObject:
    def __call__(self):
        print("I can be called like a function.")

obj = CallableObject()
print(callable(obj))
# Output: True
# Explanation: The object defines __call__, so it's callable.
```

```
True
False
True
```

10. chr()

Point	Description
Use Case	The `chr()` function is used to return the **character** that corresponds to a specified **Unicode code point (integer)**. It's the inverse of the `ord()`

Point	Description
	function.
Syntax	`chr(number)`
Parameter	`number` *(Required)* — An integer value that represents a valid Unicode code point

Useful Information

No.	Point	Description
1	Converts Code to Character	Translates a Unicode integer value into its character equivalent.
2	Unicode Range	Only accepts integers in the valid Unicode range (`0` to `0×110000`).
3	Complement of ord()	Use `chr()` to get a character from a code point, and `ord()` to go the other way.
4	Useful in Encoding Tasks	Commonly used in tasks like encoding, compression, or handling ASCII tables.
5	Raises ValueError	If the number is outside the allowed range, a `ValueError` is raised.

```
# Example 1: Convert Unicode code point 97 to character
char = chr(97)
print(char)
# Output: a
# Explanation: Unicode 97 corresponds to the lowercase letter 'a'.

# Example 2: Convert multiple Unicode points
print(chr(65))   # Output: A
print(chr(8364)) # Output: €
print(chr(128512)) # Output: 😀

# Example 3: Invalid Unicode code point (out of range)
try:
    print(chr(1114112))  # Just beyond the Unicode limit
except ValueError as e:
    print("Error:", e)
# Output: Error: chr() arg not in range(0×110000)
```

```
a
A
€
😀
Error: chr() arg not in range(0x110000)
```

11. classmethod()

Point	Description
Use Case	The `classmethod()` function is used to convert a method into a class method, which can be called on the class itself or on instances of the class. It receives the class (`cls`) as its first argument instead of the instance (`self`).
Syntax	`classmethod(function)`
Parameter	`function` *(Required)* — A method that should behave as a class method.

Useful Information

No.	Point	Description
1	Works with Class, Not Instance	Receives the class (`cls`) as the first parameter, allowing access to class-level data.
2	Shared Across All Instances	Useful when a method needs to work with or modify class-level attributes, not instance data.
3	Typically Used with @classmethod Decorator	`classmethod()` is usually used in the form of a decorator `@classmethod` above the method definition.
4	Can Create Alternative Constructors	Often used to define factory methods that construct objects in different ways.
5	Not the Same as staticmethod()	Unlike `staticmethod()`, class methods still receive the class context (`cls`) as an argument.

```python
class IntensityLogger:
    log_entries = []  # Class-level attribute shared across all instances

    def __init__(self, message):
        self.message = message

    @classmethod
    def log(cls, message):
        cls.log_entries.append(message)  # Accesses class attribute
        print(f"[CLASS LOG] {message}")

# Calling the class method using the class
IntensityLogger.log("Tutorial started.")
# Output: [CLASS LOG] Tutorial started.

# Accessing the shared log across instances
print(IntensityLogger.log_entries)
# Output: ['Tutorial started.']
```

```
[CLASS LOG] Tutorial started.
['Tutorial started.']
```

```python
### Creating a Class Method Without a Decorator

class IntensityCounter:
    count = 0

    def increase(cls):
        cls.count += 1
        print("Increased:", cls.count)

# Convert the regular method into a class method
IntensityCounter.increase = classmethod(IntensityCounter.increase)

# Call it
IntensityCounter.increase()
# Output: Increased: 1
```

```
Increased: 1
```

12. compile()

Point	Description
Use Case	The `compile()` function transforms a string, bytes, or AST (Abstract Syntax Tree) object into a Python code object, which can then be executed using `exec()` or `eval()`. This is useful when dynamically executing Python code at runtime.
Syntax	`compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)`
Parameters	`source` *(Required)* — The actual code to compile (string, bytes, or AST object).
	`filename` *(Required)* — A filename string (can be dummy if not from a file).
	`mode` *(Required)* — Must be `'exec'`, `'eval'`, or `'single'`.
	`flags` *(Optional)* — Compilation flags. Default is `0`.
	`dont_inherit` *(Optional)* — Prevents inheriting future statements. Default is `False`.
	`optimize` *(Optional)* — Sets the optimization level (`-1`, `0`, `1`, `2`).

Useful Information

No.	Point	Description
1	Compiles to Executable Code	Converts source code to an internal code object that can be executed with `exec()` or `eval()`.
2	Supports Three Modes	`'eval'` for expressions, `'exec'` for multiple statements, and `'single'` for one-line statements.
3	Dynamic Code Execution	Ideal for scenarios where Python code needs to be created or modified at runtime.

No.	Point	Description
4	Filename is Arbitrary for Strings	If source is not from a file, `filename` can be any string (e.g., `'test'`).
5	Works With AST	Accepts AST objects from the `ast` module for advanced usage like code transformation.

```
# Example 1: Using `compile()` with `eval` mode
# Compiles a single expression and evaluates it
x = compile('55 + 10', 'test', 'eval')
result = eval(x)
print(result)
# Output: 65

# Example 2: Using `compile()` with `exec` mode
# Compiles multiple statements and executes them
code_block = '''
print("Executing block ... ")
x = 5 * 3
print("Result:", x)
'''
compiled_code = compile(code_block, 'demo', 'exec')
exec(compiled_code)
# Output:
# Executing block ...
# Result: 15

#Example 3: One-liner using `exec`
x = compile('print(60)', 'script', 'exec')
exec(x)
# Output: 60
```

```
65
Executing block ...
Result: 15
60
```

13. complex()

Point	Description
Use Case	The `complex()` function creates a complex number by specifying the real and imaginary components. It can also interpret a string representation of a complex number.
Syntax	`complex(real, imaginary)`
Parameters	`real` *(Required)* — The real part of the complex number. Can be an integer, float, or string like `'3+5j'`. Defaults to `0` if not provided.
	`imaginary` *(Optional)* — The imaginary part of the complex number. Only used if `real` is not a string. Defaults to `0`.

Useful Information

No.	Point	Description
1	Supports Two Input Types	Accepts numbers or a single string representing a complex number.
2	Returns a Complex Object	The result will be in the format `a + bj` where `a` is the real part and `b` is the imaginary part.
3	Imaginary Suffix Must Be `j`	In Python, imaginary numbers are written using `j`, not `i` (e.g., `3+5j`).
4	Input String Ignores Second Parameter	When the first argument is a string, the second parameter must be omitted.

```
#Example 1: Using two numeric values

# Create a complex number with real part 3 and imaginary part 5
x = complex(1, 2)
print(x)
# Output: (1+2j)

# Example 2: Using a string representation
# Create a complex number from a string
x = complex('1+2j')
print(x)
# Output: (1+2j)

#Example 3: Omitting the imaginary part
```

```python
# Only real part provided, imaginary defaults to 0
x = complex(4)
print(x)
# Output: (4+0j)
```

```
(1+2j)
(1+2j)
(4+0j)
```

14. delattr()

Point	Description
Use Case	The `delattr()` function removes a specified attribute from an object or class by referencing the attribute name as a string.
Syntax	`delattr(object, attribute)`
Parameter	`object` *(Required)* — The target object or class instance.
	`attribute` *(Required)* — A string representing the name of the attribute to be deleted.

Useful Information

No.	Point	Description
1	Deletes Class or Instance Attributes	Can remove attributes from both class definitions and instantiated objects.
2	Requires Attribute Name as String	Accepts the attribute name dynamically as a string.
3	Raises `AttributeError`	If the specified attribute does not exist.
4	Often Used in Meta-programming	Useful when managing dynamic or configurable object states.
5	Cannot Delete Special Attributes	Should not be used to remove Python's built-in special methods like `__init__`, `__dict__`, etc.

Example 1: Removing an Attribute from a Class

```
# Define a class with multiple attributes
class Person:
    name = "Alice"
    age = 30
    country = "Canada"

# Delete the 'age' attribute from the class
delattr(Person, 'age')

# Check if the attribute still exists
print(hasattr(Person, 'age'))
# Output: False
# Explanation: The 'age' attribute has been removed from the class
```

```
False
```

Example 2: Removing an Attribute from an Object Instance

```
# Define a class and initialize attributes
class User:
    def __init__(self):
        self.username = "intensity_writer"
        self.role = "ML Author"

# Create an object of the User class
u = User()

# Remove the 'role' attribute from the object
delattr(u, 'role')

# Verify that the attribute is removed
print(hasattr(u, 'role'))
# Output: False
# Explanation: 'role' no longer exists in the object instance
```

```
False
```

15. dict()

Point	Description
Use Case	The `dict()` function is used to create a new dictionary, which is an unordered and mutable collection of key-value pairs.
Syntax	`dict(**kwargs)`
Parameter	`**kwargs` *(Optional)* — Any number of keyword arguments in the form `key=value`, which become the dictionary's keys and values.

Useful Information

No.	Point	Description
1	Returns a Dictionary Object	Creates a dictionary from keyword arguments or other mappings.
2	Supports Various Data Formats	Can be initialized from lists of tuples, keyword arguments, or other dictionary-like objects.
3	Keys Must Be Hashable	Keys must be of an immutable type such as strings, numbers, or tuples.
4	More Than Just Keyword Args	Also accepts iterable key-value pairs via `dict([(key1, val1), (key2, val2)])`.
5	Frequently Used in Data Manipulation	Ideal for configurations, structured data, or ML parameters.

```python
# Example 1: Creating a Dictionary with Keyword Arguments

# Creating a dictionary using keyword arguments
person_info = dict(name="Raj", age=40, country="India")

print(person_info)
# Output: {'name': 'Raj', 'age': 40, 'country': 'India'}
```

```
{'name': 'Raj', 'age': 40, 'country': 'India'}
```

```python
# Example 2: Using `dict()` with a List of Tuples

# Creating a dictionary from a list of tuples
params = dict([("learning_rate", 0.01), ("batch_size", 32)])

print(params)
# Output: {'learning_rate': 0.01, 'batch_size': 32}
```

```
{'learning_rate': 0.01, 'batch_size': 32}
```

16. dir()

Point	Description
Use Case	The `dir()` function lists all the valid attributes (properties and methods) of a specified object. This includes custom attributes as well as built-in Python attributes.
Syntax	`dir(object)`
Parameter	`object` *(Optional)* — The object whose attributes you want to inspect. If no object is provided, it returns the names in the current local scope.

Useful Information

No.	Point	Description
1	Helps with Introspection	Useful to explore available methods and attributes of objects, modules, and classes.
2	Shows Built-in Attributes	Returns not just user-defined, but also special and inherited attributes.
3	Works Without Parameters	If no argument is passed, it returns the names in the current scope.
4	Returns a List	Always returns a list of attribute names as strings.
5	Commonly Used in Debugging	Helps developers understand object capabilities during runtime.

```python
# Define a simple class with some attributes
class ModelConfig:
    model_name = "RandomForest"
    n_estimators = 100
    max_depth = 10

# Use dir() to inspect available attributes and methods
print(dir(ModelConfig))

# Output (partial, varies by environment):
# ['__class__', '__delattr__', ..., 'max_depth', 'model_name',
'n_estimators']
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'max_depth', 'model_name',
'n_estimators']
```

17. divmod()

Point	Description
Use Case	The `divmod()` function returns a tuple containing both the **quotient** and **remainder** from dividing one number (dividend) by another (divisor).
Syntax	`divmod(dividend, divisor)`
Parameters	`dividend` *(Required)* — The number to be divided.
	`divisor` *(Required)* — The number by which to divide.

Useful Information

No.	Point	Description
1	Returns a Tuple	The result is a tuple in the form `(quotient, remainder)`.
2	Works with Integers and Floats	Supports both integer and floating-point operands.
3	Efficient Alternative	Faster and cleaner than calling `a // b` and `a % b` separately.
4	Useful in Iteration	Often used in loop counters or position-based calculations.
5	Type-Dependent	If both inputs are integers, the result is integer-based; if floats are used, so are float results.

```
# Use divmod to divide 9 by 4
result = divmod(9, 4)

# Display the result
print(result)

# Output:
# (2, 1)
```

```
(2, 1)
```

18. enumerate()

Point	Description
Use Case	The `enumerate()` function adds a counter to an iterable and returns it as an `enumerate` object, which can be converted into a list or used directly in loops.
Syntax	`enumerate(iterable, start=0)`
Parameters	`iterable` *(Required)* — Any iterable like a list, tuple, or string.
	`start` *(Optional)* — The number to start counting from. Default is `0`.

Useful Information

No.	Point	Description
1	Returns an Enumerate Object	Wraps the iterable with index-value pairs.
2	Common in Loops	Ideal for use in `for` loops to get both index and item.
3	Custom Starting Index	You can define any starting point for the counter.
4	Works with Any Iterable	Accepts strings, lists, tuples, etc.
5	Lightweight and Efficient	Does not modify the original iterable and avoids manual indexing.

```python
# Define a tuple
layers = ('Input 1', 'Input 2', 'Input 3', 'Input 4', 'Input 5')

# Use enumerate to index
indexed_layers = enumerate(layers)

# Convert to list and print the result
print(list(indexed_layers))

# Output:
# [(0, 'Input 1'), (1, 'Input 2'), (2, 'Input 3'), (3, 'Input 4'), (4,
'Input 5')]
```

```
[(0, 'Input 1'), (1, 'Input 2'), (2, 'Input 3'), (3, 'Input 4'), (4,
'Input 5')]
```

19. eval()

Point	Description
Use Case	The `eval()` function parses a string and executes it as a valid Python expression. It's often used to evaluate expressions dynamically during runtime.
Syntax	`eval(expression, globals=None, locals=None)`
Parameter	`expression` *(Required)*: A valid Python expression provided as a string.
	`globals` *(Optional)*: A dictionary of global variables accessible during evaluation.
	`locals` *(Optional)*: A dictionary of local variables accessible during evaluation.

Useful Information

No.	Point	Description
1	Must Be a Valid Expression	The input must be a valid Python expression, not a block of code.
2	Useful for Dynamic Evaluation	Especially helpful when you want to evaluate formulas or conditions at runtime.
3	Security Risk	Never use `eval()` on untrusted input as it can execute arbitrary code.
4	Can Use Global & Local Context	Evaluate expressions using specified global and local variable scopes.
5	Returns the Result	`eval()` returns the result of the evaluated expression, unlike `exec()` which returns nothing.

```python
# Define a dictionary of hyperparameters
params = {
    "learning_rate": 0.01,
    "epochs": 10,
    "batch_size": 32
}

# Expression to calculate total steps dynamically
expr = "(epochs * 5000) // batch_size"

# Use eval() with params as the local context
total_steps = eval(expr, {}, params)

print("Total training steps:", total_steps)

# Output:
# Total training steps: 1562

# Explanation:
# eval() computed the number of training steps based on given parameters.
```

```
Total training steps: 1562
```

20. exec()

Point	Description
Use Case	The `exec()` function dynamically executes Python code blocks provided as strings or code objects. Unlike `eval()`, it supports full code blocks including variable assignments, loops, and function definitions.
Syntax	`exec(object, globals=None, locals=None)`
Parameter	`object` *(Required)*: A string or a compiled code object containing valid
	`globals` *(Optional)*: A dictionary representing the global context for execution.
	`locals` *(Optional)*: A dictionary representing the local context for execution.

Useful Information

No.	Point	Description
1	Supports Full Code Blocks	Can run multi-line statements, such as function definitions or loops.
2	Returns Nothing	Unlike `eval()`, `exec()` doesn't return a value.
3	Executes in Custom Contexts	You can specify global and local scopes using dictionaries.
4	Dynamically Define Code	Commonly used in metaprogramming or configuration-driven pipelines.
5	Use with Caution	Avoid using with untrusted input due to security vulnerabilities.

```
# Define code block as a string (simulating dynamic input)
code_block = """
learning_rate = 0.001
epochs = 5
print("Training will run for", epochs, "epochs at LR =", learning_rate)
"""

# Execute the code
```

```
exec(code_block)

# Output:
# Training will run for 5 epochs at LR = 0.001

# Explanation:
# The exec() function executes the multi-line string as Python code.
```

```
Training will run for 5 epochs at LR = 0.001
```

21. filter()

Point	Description
Use Case	The `filter()` function is used to filter elements in an iterable based on a condition provided by a function. It returns an iterator containing only the elements for which the function returns `True`.
Syntax	`filter(function, iterable)`
Parameters	`function` *(Required)* — A function that defines the filtering condition.
	`iterable` *(Required)* — Any iterable (like a list, tuple, or set) to apply the filter to.

Useful Information

No.	Point	Description
1	Returns an Iterator	Must convert to list/tuple to view results directly.
2	Function Must Return Boolean	The function should return `True` for items to keep.
3	Lazy Evaluation	Items are processed only when needed.
4	Often Combined with Lambda	Lambda functions simplify inline condition definitions.
5	Works with Any Iterable	Accepts lists, tuples, sets, dictionaries, and even generators.

```python
# Sample dataset of candidate ages
candidate_ages = [15, 12, 17, 18, 25, 36]

# Define filtering logic: Accept only those aged 18 or older
def is_eligible(age):
    return age ⩾ 18

# Apply filter to the list
eligible_candidates = filter(is_eligible, candidate_ages)

# Convert the result to a list and print
print(list(eligible_candidates))
```

```
# Output:
# [18, 25, 36]
```

```
[18, 25, 36]
```

22. float()

Point	Description
Use Case	The `float()` function is used to convert a valid numeric value or string into a floating-point number (a number with decimals).
Syntax	`float(value)`
Parameter	`value` *(Required)* — A number or a numeric string to be converted into a float.

Useful Information

No.	Point	Description
1	Works with Integers and Strings	Accepts both whole numbers and numeric strings like `"3.14"`.
2	Returns Float Representation	Converts the input into a number with decimal precision.

```
# Convert an integer into a floating-point number
int_val = 5
float_val = float(int_val)
print(float_val)
# Output:
# 5.0

# Convert a numeric string into a float
string_val = "5.300"
float_val_str = float(string_val)
print(float_val_str)
```

```
# Output:
# 5.3
```

```
5.0
5.3
```

23. format()

Point	Description
Use Case	The `format()` function converts a given value into a formatted string based on a specified format pattern.
Syntax	`format(value, format_spec)`
Parameter	`value` *(Required)* — The value to format.
	`format_spec` *(Required)* — A string defining how the value should be formatted.

Common Format Specifiers

Specifier	Description
`<`	Left-align the text within the available space.
`>`	Right-align the text within the available space.
`^`	Center-align the text within the available space.
`+`	Prefix positive numbers with `+`, negative with `-`.
`-`	Show only `-` for negative values (default behavior).
` ` (space)	Prefix a space for positive numbers.
`,`	Add comma as a thousand separator (e.g., 1,000).
`_`	Add underscore as a thousand separator (e.g., 1_000).
`b`	Binary format.

Specifier	Description
`c`	Unicode character representation.
`d`	Decimal (base-10) integer.
`e` / `E`	Scientific notation (lower or upper case).
`f` / `F`	Fixed-point number (with or without capital `F`).
`g` / `G`	General number formatting.
`o`	Octal format.
`x` / `X`	Hexadecimal format (lower or upper case).
`n`	Number format based on current locale.
`%`	Percentage format.

```
# Format a floating-point value as a percentage
confidence_score = 0.85
formatted_confidence = format(confidence_score, '%')
print(formatted_confidence)
# Output:
# 85.000000%

# Format an integer to hexadecimal
user_id = 255
hex_user_id = format(user_id, 'x')
print(hex_user_id)
# Output:
# ff

# Format a large number with comma separator (for better readability in
reports)
total_parameters = 12345678
formatted_params = format(total_parameters, ',')
print(formatted_params)
# Output:
# 12,345,678
```

```
85.000000%
ff
```

```
12,345,678
```

24. frozenset()

Point	Description
Use Case	The `frozenset()` function creates an **immutable set** object, meaning its contents cannot be changed after creation. It is hashable and can be used as keys in dictionaries or added to sets.
Syntax	`frozenset(iterable)`
Parameter	`iterable` *(Required)* — Any iterable collection (like list, set, tuple, dictionary, etc.) whose elements will be used to construct the frozenset.

Useful Information

No.	Point	Description
1	Immutable	Unlike a normal set, you cannot add or remove items from a `frozenset`.
2	Hashable	Because of its immutability, a `frozenset` can be used as a dictionary key or as an element in another set.
3	Removes Duplicates	Like a regular set, duplicate elements from the input iterable are removed.
4	Accepts Any Iterable	Input can be a list, tuple, string, dictionary (only keys are used), or another set.

```
# Sample input
feature_tags = ['deep_learning', 'transformer', 'attention',
'transformer', 'nlp']

# Create an immutable version of feature tags
immutable_tags = frozenset(feature_tags)

# Output the frozen set
print("Frozen feature tags:", immutable_tags)

# Attempt to modify the frozenset (this will raise an error)
```

```
# immutable_tags.add('reinforcement_learning')  # Uncommenting this line
will cause an AttributeError

# Output:
# Frozen feature tags: frozenset({'attention', 'nlp', 'transformer',
'deep_learning'})
```

```
Frozen feature tags: frozenset({'attention', 'transformer',
'deep_learning', 'nlp'})
```

25. getattr()

Point	Description
Use Case	The `getattr()` function retrieves the value of a specified attribute from an object.
Syntax	`getattr(object, attribute_name, default_value)`
Parameter	`object` *(Required)* — The object from which you want to fetch the attribute.
	`attribute_name` *(Required)* — The name (string) of the attribute you want to retrieve.
	`default_value` *(Optional)* — Value to return if the attribute is not found (prevents `AttributeError`).

Useful Information

No.	Point	Description
1	Dynamic Access	Useful when attribute names are not known beforehand (e.g., from config files or user input).
2	Fallback Option	If the attribute doesn't exist, you can provide a default value to avoid exceptions.
3	Works on Classes & Instances	You can use `getattr()` on both class-level and instance-level attributes.

No.	Point	Description
4	Prevents Crashes	Avoids runtime errors by handling missing attributes gracefully.

```
# Example 1: Fetching a Valid Attribute

# Define a simple configuration class for an AI model
class ModelConfig:
    model_name = "BERT"
    version = "v1.0"
    task = "text-classification"

# Retrieve an existing attribute
model_task = getattr(ModelConfig, 'task')
print("Model Task:", model_task)

# Output:
# Model Task: text-classification
```

```
Model Task: text-classification
```

```
# Example 2: Retrieving a Non-Existing Attribute with Default Fallback

# Trying to access a non-existing attribute 'dropout'
# Provide default value to avoid error

dropout_value = getattr(ModelConfig, 'dropout', 'not configured')
print("Dropout:", dropout_value)

# Output:
# Dropout: not configured
```

```
Dropout: not configured
```

26. globals()

Point	Description
Use Case	The `globals()` function returns a dictionary representing the current global symbol table. This is where all global variables and functions are stored during program execution.
Syntax	`globals()`
Parameters	This function takes **no parameters**.

Useful Information

No.	Point	Description
1	Returns Global Scope	The result is a dictionary containing all global identifiers like functions, classes, and variables.
2	Dynamic Access	Useful for dynamically modifying or accessing global variables during runtime.
3	Includes Special Keys	Contains Python's built-in special variables such as `'__name__'`, `'__file__'`, and `'__doc__'`.
4	Caution	Avoid excessive use in production code—it can lead to hard-to-maintain and error-prone logic.

```
# Example 1: View Global Symbol Table

# Global variable used in model setup
model_name = "TransformerXL"

# Retrieve the global symbol table
global_scope = globals()

# Access the value of a global variable using its name
print("Model Name:", global_scope["model_name"])

# Output:
# Model Name: TransformerXL
```

```
Model Name: TransformerXL
```

```
# Example 2: Check for Script Metadata in Globals

# Print special script-related global variables
globals_dict = globals()

# Check if script metadata like '__name__' exists
print("__name__:", globals_dict["__name__"])

# Output:
# __name__: __main__
```

```
__name__: __main__
```

27. hasattr()

Point	Description
Use Case	The `hasattr()` function checks whether a given object contains a specific attribute. It returns `True` if the attribute exists; otherwise, it returns `False`.
Syntax	`hasattr(object, attribute)`
Parameters	`object` *(Required)* — The object to inspect.
	`attribute` *(Required)* — A string representing the name of the attribute to check.

Useful Information

No.	Point	Description
1	Attribute Existence Check	Helps safely check if an object has a specific property before accessing it.

No.	Point	Description
2	Common in OOP	Widely used in class-based programming for reflection and conditional logic.
3	Prevents Errors	Avoids potential `AttributeError` by checking first.
4	Input Must Be String	The attribute name must be passed as a string.

```python
# Define a simple class representing a student
class Student:
    name = "Alice"
    age = 20

# Check if the Student class has certain attributes
print(hasattr(Student, 'name'))   # True - because 'name' exists
print(hasattr(Student, 'grade'))  # False - 'grade' is not defined

# Output:
# True
# False
```

```
True
False
```

28. hash()

Point	Description
Use Case	The `hash()` function returns the hash value (an integer) of an object, which is useful when storing objects in hash-based collections like dictionaries and sets.
Syntax	`hash(object)`
Parameter	`object` *(Required)* — Any hashable object such as strings, numbers, or tuples (with only hashable items).

Useful Information

No.	Point	Description
1	Works on Immutable Types	Only hashable (immutable) types like `int`, `float`, `str`, and tuples (with immutable elements) are accepted.
2	Enables Dictionary and Set Keys	Hash values allow objects to be used as keys in dictionaries or elements in sets.
3	Consistent During Execution	Hashes remain consistent during a program's execution but may differ between runs (due to hash randomization).
4	Custom Classes	You can override `__hash__()` in user-defined classes to define custom hashing logic.

```
# Example 1: Hash a String

# Get hash value of a string
string_name = "Deep learning"
print(hash(string_name))

# Output: Varies depending on Python session
```

```
-6144713030499523922
```

```
# Example 2: Hash a Tuple

# Hash a tuple (must contain only hashable elements)
coordinates = (10, 20)
print(hash(coordinates))

# Output: Varies each session but consistent during runtime
```

```
-4873088377451060145
```

```
# Example 3: Hash in Dictionary

# Define a configuration ID using a tuple as a key
```

```python
model_config = {
    ("resnet", 50): "ImageNet ResNet-50",
    ("transformer", 12): "BERT Base"
}

# Access a value using a tuple key (hash internally used)
print(model_config[("resnet", 50)])

# Output:
# ImageNet ResNet-50
```

```
ImageNet ResNet-50
```

29. help()

Point	Description
Use Case	The `help()` function launches Python's built-in help utility, which provides documentation for objects, modules, classes, functions, keywords, and more.
Syntax	`help(object)`
Parameter	`object` *(Optional)* — The object you want help documentation for (can be a module, class, function, string keyword, etc.). If no argument is passed, it enters interactive help mode.

Useful Information

No.	Point	Description
1	Interactive Mode	If called without arguments, enters an interactive help shell. Type `quit` to exit.
2	Useful for Learning	Ideal for quickly checking documentation, method signatures, and usage.
3	Works with Keywords	Supports help on Python keywords like `for`, `def`, `class`, etc.

No.	Point	Description
4	Works in Terminal or IDE	Best used in command-line or script environment; limited output in some IDE consoles.

```
# Example 1: Get Help on Built-in Type

# Get help on the built-in list class
help(list)

# Output: Detailed documentation about list methods like append(),
extend(), pop(), etc.
```

```
# Example 2: Help on a Custom Function

# Define a custom function
def greet(name):
    """Returns a greeting message."""
    return f"Hello, {name}!"

# Get help documentation
help(greet)

# Output:
# Help on function greet in module __main__:
# greet(name)
#     Returns a greeting message.
```

```
Help on function greet in module __main__:

greet(name)
    Returns a greeting message.
```

```
# Example 3: Help on Python Keyword

# Help on Python keyword 'for'
help("for")
```

```
# Output: Explains the usage and syntax of the 'for' loop in Python
```

```
# Example 4: Launch Interactive Help Mode

# Launch help console (command-line only)
help()

# You can then type in 'str', 'dict', 'import', etc., and explore help
interactively
```

30. hex()

Point	Description
Use Case	The `hex()` function is used to convert an integer into its hexadecimal representation (base-16). The resulting string starts with the prefix `0x`.
Syntax	`hex(number)`
Parameter	`number` *(Required)* — An integer value to be converted to a hexadecimal string.

Useful Information

No.	Point	Description
1	Returns a String	The result is always a string starting with `'0x'`.
2	Works Only on Integers	The input must be an integer; otherwise, a `TypeError` is raised.
3	Commonly Used In	Useful in low-level programming, memory addresses, cryptography, and when working with color codes or bitwise operations.

```
# Convert 16 to hexadecimal
value = 16
hex_value = hex(value)
print(f"The hexadecimal form of {value} is: {hex_value}")
```

```
# Output:
# The hexadecimal form of 16 is: 0×10
```

```
The hexadecimal form of 16 is: 0×10
```

31. id()

Point	Description
Use Case	The `id()` function returns the **unique identity number** (memory address) of an object. This identity is constant for the object's lifetime.
Syntax	`id(object)`
Parameter	`object` *(Required)* — Any object such as a number, string, list, tuple, dictionary, function, class instance, etc.

Useful Information

No.	Point	Description
1	Identity is Unique	The `id()` is unique and constant for the life of the object.
2	Represents Memory Address	The returned integer is typically the memory address where the object is stored.
3	Same Objects, Same ID	Immutable objects like small integers or strings may reuse IDs internally for optimization.
4	Helpful For Debugging	Can be used to check if two variables point to the same object in memory.

```
# Create two variables pointing to the same list
a = [1, 2, 3]
b = a

# Compare their memory addresses
print(id(a))  # Example: 140093760517952
print(id(b))  # Same as above since both point to the same list
```

```
# Output: 'a' and 'b' reference the same object in memory.
```

```
132871744035520
132871744035520
```

32. input()

Point	Description
Use Case	The `input()` function is used to **receive input from the user** during program execution.
Syntax	`input(prompt)`
Parameter	`prompt` *(Optional)* — A string message displayed to the user before input is accepted.

Useful Information

No.	Point	Description
1	Always Returns a String	No matter what the user enters, the return type is always `str`.
2	Type Conversion Required	If you need numeric input, you must explicitly convert it using `int()`, `float()`, etc.
3	Prompt Message is Optional	You can display a message with `input()`, or just use it alone.
4	Common in CLI Programs	Frequently used to collect data interactively in console-based scripts.

```
# Example 1: Simple Name Prompt

# Ask the user to enter their name
user_name = input("What is your name? ")
```

```
# Greet the user using their input
print("Welcome to Intensity Coding, " + user_name + "!")

# Example Output (if user enters "Aditi"):
# What is your name? Aditi
# Welcome to Intensity Coding, Aditi!
```

```
What is your name? Aditi
Welcome to Intensity Coding, Aditi!
```

```
# Example 2: Convert Input to Integer for Age Calculation

# Prompt user to enter their age
age_input = input("Enter your age: ")

# Convert input from string to integer
age = int(age_input)

# Use the age in a simple calculation
print("In 5 years, you will be", age + 5)

# Example Output (if user enters "25"):
# Enter your age: 25
# In 5 years, you will be 30
```

```
Enter your age: 25
In 5 years, you will be 30
```

33. int()

Point	Description
Use Case	The `int()` function is used to **convert a specified value** into an integer (whole number).
Syntax	`int(value, base)`

Point	Description
Parameters	`value` *(Required)* — A number or string that represents an integer or float.
	`base` *(Optional)* — The numeric base for conversion when the `value` is a string (default is 10).

Useful Information

No.	Point	Description
1	Converts Float to Integer	Decimal values are truncated (not rounded).
2	Parses Strings	Can convert string numbers like `"101"` to integers.
3	Supports Base Conversion	Useful for binary, octal, or hexadecimal conversions using the `base` parameter.
4	Raises Error on Invalid Input	Providing an invalid string (like `'abc'`) will raise a `ValueError`.

```
# Example 1: Convert Float to Integer
# Convert a float to an integer
num = int(7.9)

print(num)
# Output:
# 7
```

```
7
```

```
# Example 2: Convert String to Integer
# Convert a numeric string to an integer
str_num = "45"
converted = int(str_num)

print(converted)
# Output:
# 45
```

```
45
```

```python
# Example 3: Convert Binary String Using Base

# Convert binary (base 2) string to integer
binary_str = "1010"
result = int(binary_str, 2)

print(result)
# Output:
# 10
# Explanation: '1010' in binary = 10 in decimal
```

```
10
```

34. isinstance()

Point	Description
Use Case	The `isinstance()` function checks whether a given object belongs to a specified type or class.
Syntax	`isinstance(object, type)`
Return Type	`True` if the object is an instance of the specified type/class (or any from a tuple of types), otherwise `False`.
Parameter	`object` *(Required)* - The object whose type you want to verify.
	`type` *(Required)* - A type or class (e.g., `int`, `str`) or a tuple of types to check against.

Useful Information

No.	Detail	Explanation
1	Multiple Types	Pass a tuple to check if the object matches *any* of the types.
2	Class-Based Check	Can be used with custom classes and user-defined objects.
3	Type Safety	Commonly used in validation, type-safe programming, and polymorphic behavior.

```
# Example 1: Basic Type Check

# Check if a number is an integer
num = 47
print(isinstance(num, int))  # Output: True
```

```
True
```

```
# Example 2: Check Against Multiple Types

# Check if a value matches any of the listed types
value = "Intensity"
result = isinstance(value, (str, int, list))
print(result)  # Output: True
```

```
True
```

```
# Example 3: Check Custom Class Instance

# Define a class
class ModelConfig:
    model_type = "Transformer"
    layers = 12

# Create an instance
config = ModelConfig()
```

```
# Check if config is an instance of ModelConfig
print(isinstance(config, ModelConfig))  # Output: True
```

```
True
```

```
# Example 4: Invalid Type Check

# Check if a float is an instance of list
value = 3.14
print(isinstance(value, list))  # Output: False
```

```
False
```

35. issubclass()

Point	Description
Use Case	The `issubclass()` function checks whether a class is a derived subclass of another class.
Syntax	`issubclass(class, classinfo)`
Return Type	Returns `True` if the first class is a subclass (direct or indirect) of the second class or any class in a tuple. Otherwise, returns `False`.
Parameter	`class` *(Required)* The class you want to test.
	`classinfo` *(Required)* A class object or a tuple of classes to test against.

Useful Information

No.	Detail	Explanation
1	Works only with classes	Both parameters must be class types, not instances.

No.	Detail	Explanation
2	Inheritance check	Returns `True` even for multilevel inheritance.
3	Tuple Support	You can pass a tuple of classes to check against multiple base classes.

```
# Example 1: Basic Subclass Check

# Base class
class BaseModel:
    pass

# Derived class
class CustomModel(BaseModel):
    pass

# Check if CustomModel is a subclass of BaseModel
print(issubclass(CustomModel, BaseModel))  # Output: True
```

```
True
```

```
# Example 2: Check With Unrelated Classes

class Dataset:
    pass

class Trainer:
    pass

print(issubclass(Trainer, Dataset))  # Output: False
```

```
False
```

```
# Example 3: Using Tuple of Classes

class A: pass
```

```
class B: pass
class C(A): pass

# Check if class C is a subclass of either A or B
print(issubclass(C, (A, B)))  # Output: True
```

```
True
```

36. iter()

Point	Description
Use Case	The `iter()` function returns an iterator object from a given iterable. It can also be used with a sentinel value to create an iterator from a callable that stops when the sentinel value is returned.
Syntax	`iter(object)` or `iter(callable, sentinel)`
Parameter	`object` *(Required)* — An iterable object (e.g., list, tuple, string, etc.)
	`sentinel` *(Optional)* — A value which, when returned by the callable, signals the end of iteration

Useful Information

No.	Point	Description
1	Converts to Iterator	Transforms any iterable (like list, string, etc.) into an iterator.
2	Used with `next()`	The returned iterator is consumed using the built-in `next()` function.
3	Supports Callable with Sentinel	If a callable and a sentinel are provided, the iteration continues until the callable returns the sentinel value.
4	Common in Loops	Often used internally in `for` loops and comprehensions.

```
# Example 1: Creating an iterator from a list

# Create an iterable list
```

```python
fruits_name = ["apple", "mango", "banana"]

# Convert the list into an iterator
fruit_iterator = iter(fruits_name)

# Retrieve elements one by one using next()
print(next(fruit_iterator))  # Output: apple
print(next(fruit_iterator))  # Output: mango
print(next(fruit_iterator))  # Output: banana
```

```
apple
mango
banana
```

```python
# Example 2: Using iter() with a sentinel value

# Define a callable object (function that returns user input)
def input_reader():
    return input("Enter a number (type 'exit' to stop): ")

# Create an iterator that ends when 'exit' is returned
input_iter = iter(input_reader, 'exit')

# Process the input until the sentinel is reached
for val in input_iter:
    print("You entered:", val)

# Example Input:
# Enter a number (type 'exit' to stop): 123
# You entered: 123
# Enter a number (type 'exit' to stop): 99
# You entered: 99
# Enter a number (type 'exit' to stop): exit
```

```
Enter a number (type 'exit' to stop): 123
You entered: 123
Enter a number (type 'exit' to stop): 99
You entered: 99
Enter a number (type 'exit' to stop): exit
```

37. len()

Point	Description
Use Case	The `len()` function returns the total number of elements in a sequence or collection such as strings, lists, tuples, dictionaries, sets, etc.
Syntax	`len(object)`
Parameter	`object` *(Required)* — Any sequence (e.g., string, list, tuple) or collection (e.g., dictionary, set).

Useful Information

No.	Point	Description
1	Works with Multiple Types	Can be used on strings, lists, tuples, dictionaries, sets, etc.
2	Measures Size	For strings, it counts characters; for lists, it counts elements; for dictionaries, it counts key-value pairs.
3	Required in Loops	Commonly used to control loop ranges and validations.
4	Cannot Be Used on Non-Iterable Types	Raises a `TypeError` if the object doesn't support `__len__()` internally.

```
# Example 1: Length of a string

# String input
message = "Welcome to Intensity Coding"

# Get number of characters in the string
length = len(message)
print(length)
# Output: 27
```

```
27
```

```
# Example 2: Length of a list

topics = ["NLP", "Computer Vision", "Reinforcement Learning", "Generative Models"]

# Get number of items in the list
count = len(topics)
print(count)
# Output: 4
```

```
4
```

```
# Example 3: Length of a dictionary

# Dictionary containing model names
models = {
    "CNN": "Image classification",
    "RNN": "Sequence modeling",
    "Transformer": "Language tasks"
}

# Get number of key-value pairs
print(len(models))
# Output: 3
```

```
3
```

38. list()

Point	Description
Use Case	The `list()` function is used to create a new list object from an iterable. Lists are ordered, mutable (changeable), and allow duplicate values.
Syntax	`list(iterable)`

Point	Description
Parameter	`iterable` *(Optional)* — Any iterable object (such as a string, tuple, set, or another list). If no argument is provided, it creates an empty list.

Useful Information

No.	Point	Description
1	Converts Iterables	Can convert strings, tuples, sets, and other iterables into lists.
2	Creates Empty Lists	If no argument is passed, an empty list `[]` is created.
3	Lists Are Mutable	Once created, lists can be modified—items can be added, removed, or updated.
4	List Elements Maintain Order	The order of elements in the iterable is preserved in the list.

```
# Example 1: Convert a tuple to a list

topics = ('NLP', 'Vision', 'GANs')
topics_list = list(topics)
print(topics_list)
# Output: ['NLP', 'Vision', 'GANs']

# Example 2: Create a list from a string

text = "AI"
result = list(text)
print(result)
# Output: ['A', 'I']

# Example 3: Create an empty list

model_configs = list()
print(model_configs)
# Output: []
```

```
['NLP', 'Vision', 'GANs']
['A', 'I']
```

```
[]
```

39. locals()

Point	Description
Use Case	The `locals()` function provides access to the current local symbol table in the form of a dictionary. It is useful for introspection, debugging, or understanding what variables exist in a local scope.
Syntax	`locals()`
Parameter	None — This function does not accept any arguments.

Useful Information

No.	Point	Description
1	Local Scope Only	Returns variables and definitions that exist in the local scope where it is called.
2	Output is a Dictionary	Keys are variable names and values are their current content.
3	Common Use Case	Often used inside functions to inspect or manipulate local variables.
4	Not Meant for Modification	Although it returns a modifiable dictionary, changes may not affect the actual local variables directly.

```python
# Example 1: Get local variables in the current scope

# Define a function to simulate a local scope
def ai_info():
    model = "Transformer"
    version = 4
    config = {"layers": 12, "dropout": 0.1}

    # Fetch local symbol table
    local_vars = locals()
    print(local_vars)
```

```
# Call the function
ai_info()

# Output (example):
# {'model': 'Transformer', 'version': 4, 'config': {'layers': 12,
'dropout': 0.1}}
# Explanation: Returns all local variables as key-value pairs.
```

```
{'model': 'Transformer', 'version': 4, 'config': {'layers': 12,
'dropout': 0.1}}
```

```
# Example 2: Use locals() outside a function

# Global level usage
language = "Python"
level = "Beginner"

# View current local/global scope variables
current_scope = locals()
print(current_scope)

# Output (example):
# {'__name__': '__main__', 'language': 'Python', 'level': 'Beginner',
 ... }
```

40. map()

Point	Description
Use Case	The `map()` function applies a given function to each item in one or more iterables and returns a map object (an iterator).
Syntax	`map(function, iterable1, iterable2, ... )`
Parameter	`function` *(Required)* — A function that will be applied to each element.
	`iterable` *(Required)* — One or more iterables (lists, tuples, etc.) to process. Each iterable must match the number of function arguments.

Useful Information

No.	Point	Description
1	Returns Iterator	The result is a map object, which is an iterator. Use `list()` or `tuple()` to view it.
2	Multiple Iterables	You can pass more than one iterable, but the function must accept the same number of arguments.
3	Shortest Iterable Rule	If iterables are of different lengths, iteration stops at the shortest.
4	Functional Style	Common in functional programming and cleaner than using `for` loops for transformations.

```
# Example 1: Get lengths of model names

# Define a function to calculate length
def get_length(item):
    return len(item)

# List of model names
models = ['GPT', 'BERT', 'Transformer']

# Apply function to each element using map
result = map(get_length, models)

# Convert the map object to list
print(result)
print(list(result))

# Output: [3, 4, 11]
# Explanation: Returns the length of each string in the list.
```

```
<map object at 0×78d89d036890>
[3, 4, 11]
```

```
# Example 2: Combine two sets

# Function to combine strings
```

```python
def combine(a, b):
    return f"{a}-{b}"

# Two iterables
models = ['GPT', 'BERT', 'T5']
tasks = ['NLP', 'QA', 'Summarization']

# Use map to pair each model with a task
result = map(combine, models, tasks)

# Convert to list
print(list(result))

# Output: ['GPT-NLP', 'BERT-QA', 'T5-Summarization']
# Explanation: Combines corresponding elements from both iterables.
```

```
['GPT-NLP', 'BERT-QA', 'T5-Summarization']
```

41. max()

Point	Description
Use Case	The `max()` function returns the item with the highest value among two or more values, or from within an iterable.
Syntax	`max(val1, val2, ...)` **or** `max(iterable)`
Parameter	Individual values *(e.g., numbers or strings)* — Compared directly.
	An **iterable** (list, tuple, etc.) — The function returns the maximum element within the collection.

Useful Information

No.	Point	Description
1	Supports Strings	When comparing strings, it uses lexicographical (alphabetical) order.
2	Accepts Key Argument	You can use `key=` to customize the comparison logic.

No.	Point	Description
3	Works with Numbers or Iterables	Can compare direct values or elements inside lists, tuples, etc.
4	Raises Error on Empty Iterable	A `ValueError` is raised if the iterable is empty and no default is provided.
5	Mixed-Type Caution	Comparing different types (e.g., string vs. int) will raise an error.

```
# Example 1: Find maximum among numeric values

# Compare two numbers directly
result = max(15, 10)
print(result)

# Output: 15
```

```
15
```

```
# Example 2: Find max value in a tuple

scores = (12, 44, 31, 78)

# Get highest score
result = max(scores)
print(result)

# Output: 78
```

```
78
```

```
# Example 3: Max from strings (alphabetical comparison)

# Compare string values
names = ["Raj", "Dev", "Virat"]
```

```
# Find alphabetically last name
result = max(names)
print(result)

# Output: Virat
```

```
Virat
```

42. memoryview()

Point	Description
Use Case	The `memoryview()` function creates a memory view object that allows direct access to the internal data of a byte-based object without copying it. Useful for handling large binary data efficiently.
Syntax	`memoryview(object)`
Parameter	`object` *(Required)* — A `bytes` or `bytearray` object.

Useful Information

No.	Point	Description
1	No Data Copy	Provides a way to access data without copying, which is memory-efficient for large binary files.
2	Read and Write (for bytearray)	Supports read/write access for `bytearray`, read-only for `bytes`.
3	Indexable	You can access each byte using indexing.
4	Slicing Support	Supports slicing to view parts of the memory.
5	Not for Strings	Works only with `bytes` or `bytearray`, not plain `str` objects.

```
# Example 1: Create a memory view from bytes

# Create a bytes object
```

```
data = b"Intensity"

# Get a memoryview of the bytes
mem_view = memoryview(data)

print(mem_view)       # Output: <memory at 0x ... >
print(mem_view[0])    # Output: 73 → ASCII of 'I'
print(mem_view[1])    # Output: 110 → ASCII of 'n'
```

```
<memory at 0×78d89d09bac0>
73
110
```

```
# Example 2: Modify bytearray using memoryview

# Create a mutable bytearray
buffer = bytearray(b"AI")

# Create a memoryview of it
view = memoryview(buffer)

# Modify the first byte (replace 'A' → 'B')
view[0] = 66    # 66 is ASCII of 'B'

print(buffer)   # Output: bytearray(b'BI')
```

```
bytearray(b'BI')
```

```
# Example 3: Slice and inspect memory

# Create a memoryview over a bytearray
buffer = bytearray(b"coding")

view = memoryview(buffer)

# Slice the memoryview and convert to bytes
```

```
sub_view = view[1:5]
print(bytes(sub_view))  # Output: b"odin"
```

```
b'odin'
```

43. min()

Point	Description
Use Case	The `min()` function returns the smallest item among the arguments passed or from the items in an iterable. Works with numbers and strings.
Syntax	`min(val1, val2, ..., valN)` or `min(iterable)`
Parameter	Either multiple values or a single iterable like a list, tuple, etc.

Useful Information

No.	Point	Description
1	Accepts Iterable or Multiple Arguments	Works with both iterables (e.g., lists, tuples) and multiple individual arguments.
2	Supports Different Data Types	Works with numbers and strings (uses ASCII/lexicographic comparison for strings).
3	Custom Comparison	Can be extended using the `key` argument to customize how values are compared.
4	Raises Error on Empty Input	If the iterable is empty and no default is provided, a `ValueError` is raised.

```
# Example 1: Finding the smallest number

x = min(5, 15, 2, 8)
print(x)
# Output: 2
```

```
2
```

```
# Example 2: Using min() on strings

# Alphabetical comparison - finds the smallest based on lexicographical
order
names = ["Zara", "Liam", "Ava", "Noah"]
x = min(names)
print(x)
# Output: 'Ava' (since 'A' comes before others alphabetically)
```

```
Ava
```

```
# Example 3: min() with a tuple of numbers

# Tuple input
scores = (22, 45, 19, 88)
lowest = min(scores)
print(lowest)
# Output: 19
```

```
19
```

```
# Example 4: Using key argument for custom comparison

# Find the shortest string using key=len
models = ["ResNet", "GPT", "EfficientNet", "VGG"]
shortest_name = min(models, key=len)
print(shortest_name)
# Output: 'GPT'
```

```
GPT
```

44. next()

Point	Description
Use Case	The `next()` function retrieves the next item from an iterator. If the iterator is exhausted, a default value can be returned instead of raising an error.
Syntax	`next(iterator)` or `next(iterator, default)`
Parameter	`iterator` *(Required)* — An object with an `__iter__()` and `__next__()` method.
	`default` *(Optional)* — A value returned when the iterator is exhausted.

Useful Information

No.	Point	Description
1	Iterator Only	Works only on iterator objects created using `iter()`.
2	Raises StopIteration	If no default value is provided and the iterator is exhausted, a `StopIteration` error is raised.
3	Default for Safety	Use the `default` argument to avoid errors and return a fallback value.

```
# Example 1: Basic Usage with iter()

# Create an iterator from a list of AI model names
models = iter(["GPT", "BERT", "T5"])

# Manually access each element using next()
print(next(models))  # Output: GPT
print(next(models))  # Output: BERT
print(next(models))  # Output: T5
```

```
GPT
BERT
```

```
T5
```

```python
# Example 2: Using next() with a default fallback value

# List of data types used in ML
types = iter(["Tensor", "Array", "Matrix"])

# Access items and provide a default for when the list is exhausted
print(next(types, "None"))  # Output: Tensor
print(next(types, "None"))  # Output: Array
print(next(types, "None"))  # Output: Matrix
print(next(types, "None"))  # Output: None (since all elements are
already consumed)
```

```
Tensor
Array
Matrix
None
```

```python
# Example 3: Avoiding `StopIteration` in AI Data Stream

# Simulating batch data from a generator
def data_stream():
    yield "Batch 1"
    yield "Batch 2"

stream = data_stream()

# Safe access using next() with default
print(next(stream, "No more data"))  # Output: Batch 1
print(next(stream, "No more data"))  # Output: Batch 2
print(next(stream, "No more data"))  # Output: No more data
```

```
Batch 1
Batch 2
No more data
```

45. object()

Point	Description
Use Case	The `object()` function creates a basic empty object. It's the most fundamental base class in Python, from which all other classes inherit.
Syntax	`object()`
Parameter	None — This function does not take any parameters.

Useful Information

No.	Point	Description
1	Immutable Object	You cannot dynamically assign new attributes or methods to an `object()` instance.
2	Base of All Classes	Every class in Python inherits from the base `object` class implicitly or explicitly.
3	Default Methods	Methods like `__str__()`, `__eq__()`, and `__class__()` are inherited from `object`.
4	Used in Class Inheritance	`class MyClass(object)` is a common pattern to indicate inheritance from the base class.
5	Lightweight & Minimal	Useful for creating simple placeholders or identity comparisons.

```
# Example 1: Creating and Inspecting an `object` Instance

# Creating an empty object using object()
empty = object()

# View all built-in attributes and methods inherited from base object
print(dir(empty))

# Output: ['__class__', '__delattr__', '__dir__', '__doc__', ... ,
'__str__', '__subclasshook__']
# Explanation: These are the default attributes every Python object has.
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

```python
# Example 2: Attempting to Add Attributes (Fails)

# Trying to add attributes to a base object
obj = object()

# This will raise an AttributeError since object() is immutable and
doesn't allow setting attributes
try:
    obj.name = "AI Model"
except AttributeError as e:
    print("Error:", e)

# Output:
# Error: 'object' object has no attribute 'name'
```

```
Error: 'object' object has no attribute 'name'
```

```python
# Example 3: Base Class for Inheritance

# Define a custom class inheriting from base object explicitly
class ModelConfig(object):
    def __init__(self):
        self.name = "Transformer"
        self.layers = 12

config = ModelConfig()
print(config.name)   # Output: Transformer
print(config.layers) # Output: 12
```

```
Transformer
12
```

46. oct()

Point	Description
Use Case	The `oct()` function converts a given integer into its octal (base-8) representation as a string.
Syntax	`oct(number)`
Parameter	`number` *(Required)* — An integer value that you want to convert to an octal string.

Useful Information

No.	Point	Description
1	Prefix Format	The returned string always starts with `"0o"`, indicating an octal number.
2	Integer Required	Only integers can be passed to `oct()`. Floating-point or string inputs will raise a `TypeError`.
3	Common Use Case	Often used in low-level programming, file permissions (like UNIX chmod), or data encoding.
4	Reversible	You can convert the result back to an integer using `int(octal_string, 8)`.

```python
# Example 1: Convert Decimal to Octal

# Convert a decimal number to its octal representation
number = 15
octal_value = oct(number)

print(octal_value)
# Output: 0o17
```

```
0o17
```

47. open()

Point	Description
Use Case	The `open()` function is used to open a file and return a file object, which allows reading, writing, or appending based on the selected mode.
Syntax	`open(file, mode)`
Parameter	`file` *(Required)* — The file path (relative or absolute) to open.
	`mode` *(Optional)* — A string indicating how the file should be opened. Defaults to `"r"` (read mode).

File Open Modes in Python

Mode	Description
"r"	Read-only. Default mode. Error if file doesn't exist.
"a"	Append mode. Creates file if it doesn't exist.
"w"	Write mode. Overwrites file if it exists, creates if not.
"x"	Exclusive creation. Error if file already exists.

Text vs Binary Mode

Mode	Description
"t"	Text mode (default). Use for plain text files.
"b"	Binary mode. Use for binary files like images or models.

File Modes with Variations

Mode	Purpose
r	Read text. Error if file doesn't exist.
rb	Read binary. Error if file doesn't exist.
r+	Read and write text. Error if file doesn't exist.
rb+	Read and write binary. Error if file doesn't exist.
w	Write text. Overwrites or creates a new file.
wb	Write binary. Overwrites or creates a new file.
w+	Read and write text. Overwrites or creates a new file.
wb+	Read and write binary. Overwrites or creates a new file.
a	Append text. Creates file if not present. Inserts data at end of the file.
ab	Append binary. Creates file if not present. Inserts data at end of the file.
a+	Read and append text. Creates file if not present. Inserts data at end of the file.
ab+	Read and append binary. Creates file if not present. Inserts data at end of the file.

Useful Information

No.	Point	Description
1	Default Mode	If no mode is specified, the file is opened in `"r"` (read) and `"t"` (text) mode.
2	Binary vs Text	Use `"b"` for binary files like images, and `"t"` for text files.
3	Safe Handling	It's good practice to use a `with` statement for automatic resource management.
4	File Object Methods	Common methods: `read()`, `write()`, `readlines()`, `close()`.

```
# Example 1: Reading a File (Basic Read Mode)

# Open and read a text file using default 'rt' mode (read text)
with open("intensity.txt", "r") as file:
    content = file.read()
```

```
    print(content)

# Output:
# (Displays content of intensity.txt if it exists)
```

```
cat
dog
bird
```

```
#  Example 2: Writing Data to a File (Write Mode)

# Overwrite the file with new content using write mode
with open("intensity.txt", "w") as file:
    file.write("cat")

# Output: Content written to intensity.txt
```

```
# Example 3: Appending to a File (Append Mode)

# Append a new line without erasing existing content
with open("intensity.txt", "a") as file:
    file.write("\ndog")

# Output: Appends to intensity.txt
```

```
# Example 4: Binary File Handling (Binary Read Mode)

# Open an image file in binary mode
with open("intensity.png", "rb") as image_file:
    binary_data = image_file.read()
    print(type(binary_data))

# Output:
# <class 'bytes'>
```

```
<class 'bytes'>
```

48. ord()

Point	Description
Use Case	The `ord()` function returns the Unicode code point (integer representation) of a given character.
Syntax	`ord(character)`
Parameter	`character` *(Required)* — A string containing a **single Unicode character**.

Useful Information

No.	Point	Description
1	Input Must Be One Character	You must pass only a single character; multi-character strings raise a `TypeError`.
2	Unicode Support	Works with any character in the Unicode standard, including non-English characters.
3	Useful for Encoding Tasks	Often used in encryption, sorting, and data processing.

```
# Example 1: Convert a Lowercase Letter to Unicode

# Get the Unicode value of the character 'i'
x = ord("i")
print(x)

# Output:
# 105
```

```
105
```

```
# Example 2: Unicode of a Special Symbol
```

```python
# Get the Unicode code point of the dollar sign
symbol_code = ord("$")
print(symbol_code)

# Output:
# 36
```

```
36
```

```python
# Example 3: Unicode of a Non-English Character

# Get Unicode value of a non-English letter
char_code = ord("अ")  # Hindi letter 'A'
print(char_code)

# Output:
# 2309
```

```
2309
```

49. pow()

Point	Description
Use Case	The `pow()` function computes the power of a number. It returns `x` raised to the power of `y`, i.e., `x ** y`. If a third argument `z` is provided, it returns `(x ** y) % z`, which is useful in modular arithmetic.
Syntax	`pow(x, y, z)`
Parameter	`x` *(Required)* — The base number.
	`y` *(Required)* — The exponent.
	`z` *(Optional)* — A number to perform modulo operation with the result of `x ** y`.

Useful Information

No.	Point	Description
1	Works as Exponentiation	Equivalent to `x ** y` when the third parameter is not given.
2	Modular Arithmetic	Efficiently calculates `(x ** y) % z`, even for large numbers.
3	Works with Integers and Floats	Although modular form requires all three values to be integers.

```
# Example 1: Simple Power Calculation
# Calculate 2 raised to the power of 3
result = pow(2, 3)
print(result)
# Output:
# 8


# Example 2: Power with Modulo Operation
# Calculate (4 ** 3) % 5 using the third parameter
mod_result = pow(4, 3, 5)
print(mod_result)
# Output:
# 4
```

```
8
4
```

50. print()

Point	Description
Use Case	The `print()` function outputs the given message or object(s) to the standard output (typically the screen). It automatically converts objects to strings before printing.

Point	Description
Syntax	`print(object1, object2, ..., sep='separator', end='end', file=file, flush=flush)`
Parameters	`object(s)` *(Required)* — One or more objects to print (converted to strings).
	`sep` *(Optional)* — String inserted between objects. Default is a space `' '`.
	`end` *(Optional)* — String added after the last object. Default is newline `'\n'`.
	`file` *(Optional)* — The output stream (e.g., a file object). Default is `sys.stdout`.
	`flush` *(Optional)* — Boolean to force flushing the output buffer. Default is `False`.

Useful Information

No.	Point	Description
1	Converts Automatically	All objects passed are converted to string if not already.
2	Multiple Outputs	Supports printing multiple items separated by custom delimiters.
3	Custom Line Endings	Use `end=''` to avoid line breaks or add custom line endings.
4	File Output	Supports writing output to a file by assigning `file=open( ... )`.
5	Buffered or Unbuffered	Use `flush=True` to force immediate screen output, useful in real-time systems.

```
# Example 1: Basic Printing
# Print a simple message
print("Welcome to Intensity Coding")

# Output:
# Welcome to Intensity Coding
```

```
Welcome to Intensity Coding
```

```
# Example 2: Printing Multiple Objects
# Print multiple strings with default space separator
print("Machine", "Learning", "Tutorial")

# Output:
# Machine Learning Tutorial
```

```
Machine Learning Tutorial
```

```
# Example 3: Using Custom Separator and End
# Using a custom separator and ending with a question mark
print("How", "are", "you", sep="---", end="?")

# Output:
# How---are---you?
```

```
How---are---you?
```

```
# Example 4: Printing a Tuple
# Print a tuple as a single object
items = ("Machine", "Learning", "Tutorial")
print(items)

# Output:
# ("Machine", "Learning", "Tutorial")
```

```
('Machine', 'Learning', 'Tutorial')
```

```python
# Example 5: Redirecting Output to a File
# Write printed text to a file instead of console
with open("output.txt", "w") as f:
    print("Logging from Intensity Coding", file=f)

# Output: Written to output.txt
```

51. property()

Point	Description
Use Case	The `property()` function is used to define managed attributes in a class. It allows the use of methods as if they were attributes, enabling encapsulation and control over getting, setting, and deleting values.
Syntax	`property(fget=None, fset=None, fdel=None, doc=None)`
Parameter	`fget` *(Optional)* – Function for getting the attribute.
	`fset` *(Optional)* – Function for setting the attribute.
	`fdel` *(Optional)* – Function for deleting the attribute.
	`doc` *(Optional)* – A string that acts as the docstring for the property.

Useful Information

No.	Point	Description
1	Used in Classes	Mainly used within class definitions to define controlled access to instance variables.
2	Acts Like Attribute	Methods wrapped using `property()` can be accessed like regular attributes.
3	Enables Encapsulation	Provides a clean way to define getters, setters, and deleters.
4	Alternative to @property Decorator	`property()` function is equivalent to using the `@property`, `@<prop>.setter`, and `@<prop>.deleter` decorators.

```python
class Student:
    def __init__(self):
        self._name = ""

    # Getter function
    def get_name(self):
        return self._name

    # Setter function
    def set_name(self, value):
        if isinstance(value, str):
            self._name = value
        else:
            raise ValueError("Name must be a string")

    # Deleter function
    def del_name(self):
        print("Deleting name ... ")
        del self._name

    # Creating a property
    name = property(get_name, set_name, del_name, "Property for student
name")

# Using the class
s = Student()
s.name = "Alice"  # Calls set_name
print(s.name)     # Calls get_name
del s.name        # Calls del_name

# Output:
# Alice
# Deleting name ...
```

```
Alice
Deleting name ...
```

52. range()

Point	Description
Use Case	The `range()` function generates an immutable sequence of numbers, which is often used for iterating through loops.
Syntax	`range(start, stop, step)`
Parameters	`start` *(Optional)* — The beginning of the sequence. Default is `0`.
	`stop` *(Required)* — The endpoint (exclusive); the sequence stops before this value.
	`step` *(Optional)* — The amount by which the sequence increments. Default is `1`.

Useful Information

No.	Point	Description
1	Immutable	The `range` object is immutable and memory efficient.
2	Lazy Evaluation	Values are generated on demand and not stored in memory.
3	Can Be Casted	Convert `range` to a list or tuple using `list()` or `tuple()` for display.
4	Negative Steps	Supports decreasing sequences using a negative step.
5	Common in Loops	Frequently used with `for` loops for fixed-iteration operations.

```
# Example 1: Simple Range (0 to 5)
# Generate numbers from 0 to 5 (excluding 6)
for num in range(6):
    print(num)

# Output:
# 0
# 1
# 2
# 3
# 4
# 5
```

```
0
1
2
3
4
5
```

```
# Example 2: Custom Start and Stop
# Generate numbers from 3 to 5
for num in range(3, 6):
    print(num)

# Output:
# 3
# 4
# 5
```

```
3
4
5
```

```
# Example 3: Custom Step Value
# Generate every 2nd number from 3 to 19
for num in range(3, 20, 2):
    print(num, end=' ')

# Output:
# 3 5 7 9 11 13 15 17 19
```

```
3 5 7 9 11 13 15 17 19
```

```
# Example 4: Reverse Range Using Negative Step
# Countdown from 10 to 1
```

```python
for num in range(10, 0, -1):
    print(num, end=' ')

# Output:
# 10 9 8 7 6 5 4 3 2 1
```

```
10 9 8 7 6 5 4 3 2 1
```

```python
# Example 5: Convert range to list (for display)
# Convert a range to a list for readability
sequence = list(range(1, 6))
print("Generated sequence:", sequence)

# Output:
# Generated sequence: [1, 2, 3, 4, 5]
```

```
Generated sequence: [1, 2, 3, 4, 5]
```

53. repr()

Point	Description
Use Case	The `repr()` function returns a string that represents the specified object in a way that ideally could be used to recreate the object. It's primarily used for debugging and development purposes.
Syntax	`repr(object)`
Parameter	`object` *(Required)* — Any Python object for which a developer-readable string representation is needed.

Useful Information

No.	Point	Description
1	Returns Developer-Friendly Format	Typically includes quotes and escape characters to show how the object would be typed in code.
2	Different from `str()`	`repr()` is for debugging and development; `str()` is for end-user readability.
3	Often Used in Debug Logs	Helpful for logging detailed object structures.
4	Automatically Used in Shell	The Python shell automatically calls `repr()` on output values.

```
# Example 1: Using `repr()` with a String
text = "Intensity Coding"
debug_output = repr(text)

print(debug_output)
# Output: 'Intensity Coding'
# Explanation: repr() adds quotes, showing it as a literal string.
```

```
'Intensity Coding'
```

```
# Example 2: Difference Between `str()` and `repr()`
value = 'AI\nML'

print("Using str():", str(value))
# Output: Using str(): AI
# ML

print("Using repr():", repr(value))
# Output: Using repr(): 'AI\nML'
# Explanation: str() prints the string as-is; repr() shows escape
characters.
```

```
Using str(): AI
ML
Using repr(): 'AI\nML'
```

54. reversed()

Point	Description
Use Case	The `reversed()` function returns an iterator that accesses the given sequence in the reverse order. It does **not** modify the original sequence.
Syntax	`reversed(sequence)`
Parameter	`sequence` *(Required)* — Any iterable that supports reverse iteration such as lists, tuples, strings, or custom objects with `__reversed__()` implemented.

Useful Information

No.	Point	Description
1	Does Not Modify Original	Returns a reverse iterator; the original object remains unchanged.
2	Works on Sequences	Only works on sequence types like lists, tuples, and strings.
3	Must Convert to View	Convert to `list()` or `tuple()` to view or reuse the reversed output.
4	Iterator Exhaustion	Like any iterator, once it's used up in a loop or operation, it must be re-created.
5	Alternative to `[::-1]`	Functionally similar to slicing for reversing but memory-efficient.

```
# Example 1: Reverse a List Using `reversed()`
example_list = ["Machine", "Learning", "Tutorial"]

# Create a reverse iterator using reversed()
reverse_iter = reversed(example_list)

# Iterate and print each element in reverse
for i in reverse_iter:
    print(i)

# Output:
```

```
# Tutorial
# Learning
# Machine
```

```
Tutorial
Learning
Machine
```

```
# Example 2: Convert Reverse Iterator to List

# Original list
items = [10, 20, 30, 40]

# Use reversed() and convert to list
reversed_list = list(reversed(items))
print("Reversed List:", reversed_list)

# Output:
# Reversed List: [40, 30, 20, 10]
```

```
Reversed List: [40, 30, 20, 10]
```

```
# Example 3: Reverse a String

# Reversing a string using reversed()
text = "intensity"
reversed_text = ''.join(reversed(text))
print("Reversed String:", reversed_text)

# Output:
# Reversed String: ytisnetni
```

```
Reversed String: ytisnetni
```

```python
# Example 4: Reversed in For Loop (Tuple)

# Tuple of numbers
numbers = (1, 2, 3, 4)

# Using reversed() in a loop
for n in reversed(numbers):
    print(n, end=' ')

# Output:
# 4 3 2 1
```

```
4 3 2 1
```

55. round()

Point	Description
Use Case	The `round()` function returns a number rounded to a specified number of decimal places. If no decimal place is specified, it rounds to the nearest integer.
Syntax	`round(number, digits=0)`
Parameter	`number` *(Required)* — The numeric value to be rounded.
	`digits` *(Optional)* — The number of decimal places to round to. Defaults to `0` (returns an integer).

Useful Information

No.	Point	Description
1	Works with Integers and Floats	Can round both whole and floating-point numbers.
2	Default Rounding	If `digits` is not given, it rounds to the nearest integer.

No.	Point	Description
3	Halfway Cases	Python rounds ties to the nearest even number (banker's rounding).

```
# Example 1: Rounding to Two Decimal Places
value = 8.67892
rounded_value = round(value, 2)
print(rounded_value)
# Output: 8.68

# Example 2: Rounding Without Decimal Precision (Default)
value = 8.67892
rounded_value = round(value)
print(rounded_value)
# Output: 9
# Explanation: Rounded to the nearest whole number since no digits are
specified.
```

```
8.68
9
```

56. set()

Point	Description
Use Case	The `set()` function constructs a set object from an iterable. Sets are collections of unique elements that are unordered and unindexed.
Syntax	`set(iterable)`
Parameter	`iterable` *(Optional)* — A sequence, collection, or any iterable object (like lists, tuples, strings, etc.) used to populate the set.

Useful Information

No.	Point	Description
1	Removes Duplicates	Automatically eliminates repeated elements from the input.
2	Unordered Structure	Elements in a set do not maintain insertion order.
3	Supports Set Operations	Can be used for union, intersection, difference, etc.
4	Immutable Elements Only	Set elements must be of immutable types (e.g., numbers, strings, tuples).

```
# Creating a set from a tuple of fruits
example_list = ["cat", "dog", "dog", "cat", "bird"]
unique_element = set(example_list)

print(unique_element)
# Output: {'cat', 'dog', 'bird'}
```

```
{'cat', 'dog', 'bird'}
```

57. setattr()

Point	Description
Use Case	The `setattr()` function allows you to dynamically assign a value to an attribute of an object. If the attribute exists, it updates the value; if not, it creates the attribute.
Syntax	`setattr(object, attribute, value)`
Parameter	`object` *(Required)* — The target object whose attribute will be modified or created.
	`attribute` *(Required)* — The name of the attribute as a string.
	`value` *(Required)* — The new value to assign to the attribute.

Useful Information

No.	Point	Description
1	Modifies Object State	Useful for dynamically updating class instances.
2	Can Add New Attributes	If the attribute doesn't exist, it's created automatically.
3	Works Well with `getattr()`	Often paired with `getattr()` for attribute control.

```
# Example 1: Dynamically Update Attribute in a Class

# Define a class with default attributes
class ModelConfig:
    name = "Transformer"
    layers = 6
    pretrained = True

# Dynamically update the 'layers' attribute
setattr(ModelConfig, 'layers', 12)

# Confirm the update using getattr()
updated_layers = getattr(ModelConfig, 'layers')
print("Updated number of layers:", updated_layers)
# Output: Updated number of layers: 12
```

```
Updated number of layers: 12
```

```
# Example 3: Add Custom Metadata to an Object

class Experiment:
    pass

exp = Experiment()

# Add a custom attribute
setattr(exp, 'author', 'intensity_coding')
setattr(exp, 'version', 'v1.2')
```

```
print(exp.author)   # Output: intensity_coding
print(exp.version)  # Output: v1.2
```

```
intensity_coding
v1.2
```

58. slice()

Point	Description
Use Case	The `slice()` function generates a slice object that defines how elements should be sliced from a sequence like a list, tuple, or string. This is especially useful when passing slicing parameters programmatically.
Syntax	`slice(start, end, step)`
Parameter	`start` *(Optional)* — The index at which slicing starts (default is `0`).
	`end` *(Required)* — The index at which slicing ends (exclusive).
	`step` *(Optional)* — Interval between each index (default is `1`).

Useful Information

No.	Point	Description
1	Works with any Sequence	Can be used with lists, tuples, and strings.
2	slice() vs `:`	The `slice()` object mimics the behavior of slicing syntax `seq[start:end:step]`.
3	Ideal for Dynamic Slicing	Helpful when slicing needs to be computed or passed as a parameter.
4	Negative Indices	Supports negative indices for reverse slicing.

```
data = ["sample1", "sample2", "sample3", "sample4", "sample5", "sample6",
"sample7", "sample8"]
```

```python
# Example 1: Basic Slice
# Slice: Start at index 0, stop before index 2
s = slice(2)
print(data[s])
# Output: ['sample1', 'sample2']

# Example 2: Middle Range Slice
# Slice elements from index 3 up to but not including 5
s = slice(3, 5)
print(data[s])
# Output: ['sample4', 'sample5']


# Example 3: Using Step for Interval Slicing
# Slice every third item from start to end of the tuple
s = slice(0, 8, 3)
print(data[s])
# Output: ['sample1', 'sample4', 'sample7']
```

```
['sample1', 'sample2']
['sample4', 'sample5']
['sample1', 'sample4', 'sample7']
```

59. sorted()

Point	Description
Use Case	The `sorted()` function is used to return a new **sorted list** from the elements of any iterable (like a list, tuple, or dictionary), without modifying the original.
Syntax	`sorted(iterable, key=None, reverse=False)`
Parameters	`iterable` (Required): The collection (like a list, tuple, etc.) to be sorted.
	`key` (Optional): A function to customize sorting logic.
	`reverse` (Optional): Boolean value. If `True`, results are sorted in descending order.

Useful Information

Point	Explanation
1. Returns a New List	The `sorted()` function does not modify the original iterable. Instead, it returns a new sorted list.
2. Works with All Iterables	You can apply `sorted()` on any iterable—such as a list, tuple, dictionary, or string. For dictionaries, only the keys are sorted by default.
3. Custom Sorting with `key`	You can define a **custom function** using the `key` parameter to control sorting behavior—like sorting by **length**, substring, or custom objects.
4. Different from `.sort()`	`sorted()` is a built-in global function, whereas `.sort()` is a method available only on lists. Unlike `.sort()`, `sorted()` works with any iterable.
5. Type Consistency Required	If the iterable contains mixed data types (e.g., both integers and strings), `sorted()` will raise a TypeError, as elements cannot be compared directly.

```
# Example 1: Sorting Alphabetically (Default)

fields = ["Name", "ID", "Department", "AI", "Score"]

# Sort alphabetically
sorted_fields = sorted(fields)
print(sorted_fields)
# Output: ['AI', 'Department', 'ID', 'Name', 'Score']
```

```
['AI', 'Department', 'ID', 'Name', 'Score']
```

```
# Example 2: Sorting Numbers in Ascending Order

numbers = (1, 7, 2)

# Sort numbers numerically in ascending order
sorted_nums = sorted(numbers)
```

```
print(sorted_nums)
# Output: [1, 2, 7]
```

```
[1, 2, 7]
```

```
# Example 3: Sorting in Descending Order

fields = ["Name", "ID", "Department", "AI", "Score"]

# Sort the tuple in reverse (descending) order
sorted_field__desc = sorted(fields, reverse=True)

print(sorted_field__desc)
# Output: ['Score', 'Name', 'ID', 'Department', 'AI']
```

```
['Score', 'Name', 'ID', 'Department', 'AI']
```

```
# Example 4: Sorting Strings by Length

fields = ["Name", "ID", "Department", "AI", "Score"]

# Sort by string length using a custom key
sorted_fields = sorted(fields, key=len)
print("Sorted by field length:", sorted_fields)
# Output: Sorted by field length: ['ID', 'AI', 'Name', 'Score',
'Department']
```

```
Sorted by field length: ['ID', 'AI', 'Name', 'Score', 'Department']
```

60. staticmethod()

Point	Description
Use Case	The `staticmethod()` function is used to define a method that does not operate on class or instance data. It belongs to the class namespace but does not access `self` or `cls`.
Syntax	`staticmethod(function)`
Parameter	`function` *(Required)* — The function to be converted into a static method.

Useful Information

No.	Point	Description
1	No Access to Class/Instance	A static method does **not** access `self` or `cls`, meaning it can't access or modify class or instance variables.
2	Used for Utility Functions	Often used to group related helper methods inside a class for organizational clarity.
3	Can Be Called via Class/Instance	A static method can be accessed using the class name or any of its instances.
4	Alternative to @staticmethod Decorator	`staticmethod()` is rarely used directly in modern Python; instead, the `@staticmethod` decorator is preferred.
5	Keeps Code Encapsulated	Helps in keeping logically grouped functionality inside a class even when it doesn't depend on the class or instance state.

```python
# Defining a class to demonstrate the use of staticmethod()
class DataPreprocessor:
    def __init__(self, name):
        self.name = name

    # Convert a regular method into a static method using @staticmethod
    @staticmethod
    def normalize_data(data):
        """
        Normalize a list of numbers to range [0, 1].
        """
        min_val = min(data)
        max_val = max(data)
        return [(x - min_val) / (max_val - min_val) for x in data]
```

```python
# Calling the static method without creating an instance
normalized = DataPreprocessor.normalize_data([5, 10, 15])
print("Normalized Data:", normalized)

# Output:
# Normalized Data: [0.0, 0.5, 1.0]

# Alternative: Using `staticmethod()` Function Directly
class MathUtils:
    def square(x):
        return x * x

    # Manually converting square into a static method
    square = staticmethod(square)
print(MathUtils.square(6))  # Output: 36
```

```
Normalized Data: [0.0, 0.5, 1.0]
36
```

61. str()

Point	Description
Use Case	The `str()` function is used to convert a specified object into a human-readable string format.
Syntax	`str(object, encoding=encoding, errors=errors)`
Parameter	`object` *(Required)* — Any Python object to be converted into a string.
	`encoding` *(Optional)* — String encoding type. Used only when converting from bytes.
	`errors` *(Optional)* — Specifies how to handle encoding errors (e.g., `'strict'`, `'ignore'`, `'replace'`).

Useful Information

No.	Point	Description
1	Default Conversion	If only an object is passed, it returns its readable string form.
2	Encoding Support	`encoding` is used only when the input is a bytes-like object.
3	Works with Numbers	Converts integers, floats, and other numeric types to their string form.

```
# Example 1: Converting a float to string
x = str(7.5)
print(x)
# Output: '7.5'

# Example 2: Converting a boolean value to string
status = str(True)
print(status)
# Output: 'True'

# Example 3: Using str() on a list
items = ["AI", "ML", "NLP"]
output = str(items)
print(output)
# Output: "['AI', 'ML', 'NLP']"

# Example 4: Decoding bytes using str()
binary_data = b'Intensity'
decoded = str(binary_data, encoding='utf-8')
print(decoded)
# Output: 'Intensity'
# Explanation: The byte string is decoded to a readable string using UTF-
8 encoding.
```

```
7.5
True
['AI', 'ML', 'NLP']
Intensity
```

62. sum()

Point	Description
Use Case	The `sum()` function calculates the total of all numeric elements in an iterable.
Syntax	`sum(iterable, start)`
Parameter	`iterable` *(Required)* — A sequence (like list, tuple) containing numeric values to be added.
	`start` *(Optional)* — A number to be added to the final result. Default is `0`.

Useful Information

No.	Point	Description
1	Works with Numeric Iterables	Can be used with lists, tuples, or other iterable structures containing only numbers.
2	Supports Optional Start	Adds the `start` value to the total sum of elements.
3	Does Not Accept Non-Numeric Types	All elements must be numeric, or a `TypeError` will be raised.

```
# Example 1: Basic summation of a tuple
data = (1, 3, 5, 7, 9)
total = sum(data)
print(total)
# Output: 25

# Example 2: Using a starting value in sum
data = (1, 3, 5, 7, 9)
total = sum(data, 10)
print(total)
# Output: 35

# Example 3: Summing a list of floating point values
values = [0.25, 0.30, 0.28, 0.22]
total = sum(values)
print(total)
# Output: 1.05
```

```
25
35
1.05
```

63. super()

Point	Description
Use Case	The `super()` function is used to access methods and properties of a parent (base) class from within a child (derived) class.
Syntax	`super()`
Parameter	This function does **not require any parameters**. It automatically refers to the parent class in the current context.

Useful Information

No.	Point	Description
1	Used in Inheritance	Commonly used when child classes inherit and extend functionality from parent classes.
2	Ensures DRY Principles	Helps avoid code duplication by calling parent methods directly.
3	Supports Multiple Inheritance	Works properly with Python's method resolution order (MRO) in multi-inheritance scenarios.
4	Mainly Used Inside `__init__`	Often used in constructor overriding (`__init__`) to ensure base class initialization is not skipped.
5	Returns a Proxy Object	The result of `super()` is a proxy that delegates method calls to a parent or sibling class.

```python
# Example: Using super() to reuse parent class initialization logic
class ModelConfig:
    def __init__(self, model_name):
        self.model_name = model_name
        print(f"Base configuration loaded for model: {self.model_name}")
```

```python
# The MLTrainer class inherits from ModelConfig
class MLTrainer(ModelConfig):
    def __init__(self, model_name, epochs):
        # Call the parent class's __init__ method using super()
        super().__init__(model_name)
        self.epochs = epochs
        print(f"Training initialized for {self.epochs} epochs")

# Create an instance of MLTrainer
trainer = MLTrainer("IntensityNet", 10)

# Output:
# Base configuration loaded for model: IntensityNet
# Training initialized for 10 epochs
```

```
Base configuration loaded for model: IntensityNet
Training initialized for 10 epochs
```

64. tuple()

Point	Description
Use Case	The `tuple()` function is used to create an immutable ordered collection of items. Once created, the contents of a tuple cannot be modified (i.e., no additions, deletions, or updates).
Syntax	`tuple(iterable)`
Parameter	`iterable` *(Optional)* — A sequence (like a list, string, or set) or any iterable object to be converted into a tuple.

Useful Information

No.	Point	Description
1	Immutable	Tuples are fixed in size—once created, their elements cannot be changed.
2	Order Preserved	Tuples retain the order of the original iterable.

No.	Point	Description
3	Can Hold Multiple Types	Supports mixed data types (e.g., `int`, `str`, `list`, etc.).
4	Memory Efficient	Tuples consume less memory compared to lists due to immutability.
5	Useful in Function Returns	Commonly used to return multiple values from a function or represent fixed groupings.

```python
# Converting a list into a tuple using tuple()
features = ["height", "weight", "age"]
immutable_features = tuple(features)

print("Tuple of features:", immutable_features)
# Output:
# Tuple of features: ('height', 'weight', 'age')

# Using tuple() with a string
chars = tuple("AI")
print("Tuple from string:", chars)
# Output:
# Tuple from string: ('A', 'I')

# Using tuple() with a range object
numbers = tuple(range(3))
print("Tuple from range:", numbers)
# Output:
# Tuple from range: (0, 1, 2)

# Tuple can contain different data types
mixed = tuple([42, "text", 3.14, True])
print("Mixed type tuple:", mixed)
# Output:
# Mixed type tuple: (42, 'text', 3.14, True)
```

```
Tuple of features: ('height', 'weight', 'age')
Tuple from string: ('A', 'I')
Tuple from range: (0, 1, 2)
Mixed type tuple: (42, 'text', 3.14, True)
```

65. type()

Point	Description
Use Case	The `type()` function is used to determine the data type of an object. It can also dynamically create new types (classes) if additional parameters are provided.
Syntax	`type(object)` or `type(name, bases, dict)`
Parameters	`object` *(Required)* - If only one argument is passed, returns the type of the object.
	`name` *(Required if using 3-arg form)* - The name of the new class to be created.
	`bases` *(Optional)* - A tuple specifying the base classes (inheritance).
	`dict` *(Optional)* - A dictionary defining the attributes and methods of the class.

Useful Information

No.	Point	Description
1	Commonly Used for Type-Checking	Helps during debugging, logging, or validating input types in functions.
2	Supports One or Three Arguments	With one argument, it returns the type. With three, it dynamically creates a new class.
3	Creates Dynamic Types	Using three arguments returns `<class 'type'>` for dynamically created types.

```
# Example 1: Type Checking
features = ('height', 'weight', 'age')
print("Type of features:", type(features))
# Output: <class 'tuple'>

# Check the type of a label (string)
label = "healthy"
print("Type of label:", type(label))
# Output: <class 'str'>
```

```
# Check the type of a value (integer)
value = 95
print("Type of value:", type(value))
# Output: <class 'int'>
```

```
Type of features: <class 'tuple'>
Type of label: <class 'str'>
Type of value: <class 'int'>
```

```
# Example 2: Dynamically Creating a Class Using type()
# Create a simple class dynamically using type()
# Equivalent to:
# class Model:
#     version = 1.0

Model = type('Model', (), {'version': 1.0})

# Create an instance
ml_model = Model()
print("Model version:", ml_model.version)
# Output: Model version: 1.0
```

```
Model version: 1.0
```

66. vars()

Point	Description
Use Case	The `vars()` function is used to retrieve the `__dict__` attribute of an object, which holds its writable attributes in the form of a dictionary.
Syntax	`vars(object)`
Parameter	`object` *(Optional)* — An object that has a `__dict__` attribute. If omitted, it returns the local symbol table as a dictionary.

Useful Information

No.	Point	Description
1	Requires `__dict__` Support	Only works on objects that define a `__dict__` attribute (like user-defined classes).
2	Used for Introspection	Commonly used to inspect and debug the current state of an object's attributes.
3	Local Namespace (No Argument)	When no argument is provided, `vars()` returns the local variable scope as a dictionary.
4	Not for Built-in Types	Will raise a `TypeError` if called on a built-in type that doesn't support `__dict__`.

```
# Define a user-defined class
class Developer:
    role = "ML Engineer"
    experience = 5
    team = "Intensity Coding"

# Use vars() to inspect the attributes of the class
dev_attributes = vars(Developer)

# Print the dictionary of attributes
print(dev_attributes)
# Output: {'role': 'ML Engineer', 'experience': 5, 'team': 'Intensity
Coding'}
```

```
{'__module__': '__main__', 'role': 'ML Engineer', 'experience': 5,
'team': 'Intensity Coding', '__dict__': <attribute '__dict__' of
'Developer' objects>, '__weakref__': <attribute '__weakref__' of
'Developer' objects>, '__doc__': None}
```

```
# Using vars() without arguments to check local scope
language = "Python"
level = "Advanced"

local_vars = vars()
```

```
print(local_vars)
# Output will include keys like 'language' and 'level' with their values
```

67. zip()

Point	Description
Use Case	The `zip()` function combines elements from multiple iterables (e.g., lists, tuples) into tuples, grouping items based on their positions.
Syntax	`zip(iterator1, iterator2, ...)`
Parameter	`iterator1, iterator2, ...` *(Required)* — Two or more iterable objects to be zipped together.

Useful Information

No.	Point	Description
1	Shortest Length Wins	If the input iterables are of unequal length, the resulting iterator stops at the shortest.
2	Produces an Iterator	Returns a `zip` object which is an iterator — convert it to list or tuple to view results.
3	Supports More than Two Inputs	You can pass more than two iterables to group multiple sets together.
4	Lazy Evaluation	Since it returns an iterator, it uses memory efficiently and supports large datasets.

```
# Example 1: Zipping two tuples of equal length
names = ("Alice", "Bob", "Charlie")
roles = ("Data Scientist", "ML Engineer", "Researcher")

paired = zip(names, roles)

# Convert the zip object to a tuple to see the result
print(tuple(paired))
# Output: (('Alice', 'Data Scientist'), ('Bob', 'ML Engineer'),
('Charlie', 'Researcher'))
```

```
(('Alice', 'Data Scientist'), ('Bob', 'ML Engineer'), ('Charlie',
'Researcher'))
```

```python
# Example 2: Zipping tuples of unequal lengths
contributors = ("Raj", "Anita", "Sanjay")
teams = ("NLP", "Computer Vision")

zipped_output = zip(contributors, teams)
print(tuple(zipped_output))
# Output: (('Raj', 'NLP'), ('Anita', 'Computer Vision'))
# Explanation: Stops after the shortest iterable (`teams`) ends
```

```
(('Raj', 'NLP'), ('Anita', 'Computer Vision'))
```

```python
# Example 3: Zipping three lists together
tech = ["Python", "TensorFlow", "PyTorch"]
years = [1991, 2015, 2016]
use_case = ["General Purpose", "Deep Learning", "Research DL"]

zipped_info = zip(tech, years, use_case)
print(list(zipped_info))
# Output: [('Python', 1991, 'General Purpose'), ('TensorFlow', 2015,
'Deep Learning'), ('PyTorch', 2016, 'Research DL')]
```

```
[('Python', 1991, 'General Purpose'), ('TensorFlow', 2015, 'Deep
Learning'), ('PyTorch', 2016, 'Research DL')]
```

Found this helpful ?

Follow on LinkedIn
Master AI/ML with Intensity Coding

Bhavdip Patel

@bhavdippatel2020

Like | Comment | Share | Save

Each Article Includes Everything You Need

THEORY MADE SIMPLE
Complex ideas explained in an easy way

PYTHON CODE
Hands-on coding for practice

VISUAL LEARNING
Visual diagrams for clarity

MATH BEHIND AI/ML
Step-by-step explanation of core concepts

Explore more tutorials at Intensity Coding

www.intensitycoding.com

Intensity Coding
Something For Everyone