**Python**

* **Python** is a high-level, interpreted programming language known for its simplicity, readability, and versatility.
* Developed by **Guido van Rossum** and released in **1991**, Python supports multiple programming paradigms, including object-oriented, procedural, and functional programming.

**History of Python**

* **Late 1980s**: Guido van Rossum begins work on Python at Centrum Wiskunde & Informatica (CWI), Netherlands.
* **1991**: Python 0.9.0 released, including classes, functions, exception handling.
* **2000**: Python 2.0 released with list comprehensions and garbage collection.
* **2008**: Python 3.0 introduced with many improvements but not backward compatible.
* **Current Versions**: Python 3.x is widely used in modern development.

**Features**

Object-oriented Language

GUI programming support – Ex: tkinter, turtle, kiwi

Interpreted language - Checks line by line

Automatic garbage collection - clears the memory when unused

Large standard library

Dynamic memory allocation - Allocating memory during run time

Beginner- Friendly

Dynamically Typed and Type Checking

**Tokens**

A **token** is the smallest element in a Python program that has some meaning. **Tokens** are the **smallest units of a program** that are meaningful to the compiler or interpreter.  
Everything in a Python script—identifiers, keywords, literals, operators, and punctuators—are tokens.

* They help the **interpreter parse and understand** the code.
* Python **parses code by breaking it into tokens** first.
* Syntax errors often occur when Python cannot tokenize something correctly.

**Keywords**

**Keywords** are **reserved words** in Python that have **special meanings**. These cannot be used as identifiers (like variable names, function names, or class names). Keywords form the building blocks of Python syntax and define its structure and rules.

Python has **35+ keywords** (as of Python 3.10+), and their number may slightly vary between versions.

| **Keyword** | **Description** | | | **Use Case Example** | |
| --- | --- | --- | --- | --- | --- |
| and | Logical **AND** operator / Nested if statements | | | if a > 0 and b < 10: | |
| as | Create an alias | | | import math as m | |
| assert | Debugging aid; raises AssertionError if the condition is false | | | assert x > 0, "x must be positive" | |
| break | Exits the current loop | | | for i in range(5):  if i == 3:  break | |
| class | Defines a class | | | class Person: | |
| continue | Skips the current iteration of a loop | | | for i in range(5): if i == 2: continue | |
| def | Defines a function | | | def greet(name): | |
| del | Deletes an object | | | del x | |
| elif | Else-if condition | | | if x == 1: ... elif x == 2: | |
| else | Executes when if condition is false | | | if x == 1: ... else: | |
| except | Catches exceptions | | | try: ... except ValueError: | |
| False | Boolean value | | | is\_valid = False | |
| finally | Block that always executes in a try statement | | | try: ... finally: | |
| for | Creates a for loop | | | for i in range(5): | |
| from | Import specific part of a module | | | from math import sqrt | |
| global | Declares a global variable | | | global count | |
| if | Conditional branching | | | if score > 90: | |
| import | Imports a module | | | import os | |
| in | Checks membership | | | if x in list: | |
| is | Tests object identity | | | if a is b: | |
| lambda | Defines anonymous function | | | f = lambda x: x + 1 | |
| None | Represents the absence of value | | | x = None | |
| nonlocal | Refers to a variable in the nearest enclosing (non-global) scope | | | nonlocal x inside a nested function def outer():  x = "outer"  def inner():  nonlocal x  x = "inner"  inner()  print(x) | |
| not | Logical NOT operator / Inverted condition | | | if not done: | |
| or | Logical OR operator / Separate if statements | | | if a < 0 or b > 5: | |
| pass | Null statement, placeholder | | | if condition: pass | |
| raise | Raises an exception | | | raise ValueError("Invalid input") | |
| return | Exits a function and optionally returns a value | | | return total | |
| True | Boolean value | | | is\_logged\_in = True | |
| try | Starts a block of code to catch exceptions | | | try: risky\_code() | |
| while | Creates a while loop | | | while x < 10: | |
| with | Context manager (e.g., file handling) | | | with open("file.txt") as f: | |
| yield | Returns a value from a generator | | | yield item | |
| **Concept** | **Description** | **Example** | **Purpose** | |
| **Class** | Blueprint for creating objects | class Car: | Organize related data/methods | |
| **Function** | Block of reusable code | def greet(): | Perform a task | |
| **Object** | Instance of a class | car1 = Car() | Access class data/methods | |

A **context manager** is a construct that allows you to **manage resources efficiently** — especially when working with **files, network connections, or database sessions**. It ensures that resources are properly **acquired and released**, even if errors occur.

**Literals**  
A **literal** is a **fixed value** directly assigned to a variable or used in an expression. Unlike variables, **literals do not change** and represent constant values written directly into the source code.

| **Literal Type** | **Subtype / Category** | **Example** | **Value / Output** | **Purpose / Use Case** |
| --- | --- | --- | --- | --- |
| **1. Numeric** | Integer | x = 42 | 42 | Counting, indexing, arithmetic |
|  | Float | pi = 3.14 | 3.14 | Scientific/financial calculations |
|  | Complex | z = 2 + 3j | 2+3j | Complex number computations |
| **2. String** | Single Quote | 'hello' | hello | Basic string value |
|  | Double Quote | "hello" | hello | Alternative to single quote (can contain apostrophes) |
|  | Triple Quote | '''Hello\nWorld''' | Multi-line string | Used for multi-line strings, docstrings |
|  | Escape Characters | "He said: \"Hi!\"" | He said: "Hi!" | Include special characters (quotes, tabs, newlines) |
|  | Raw String | r"C:\Users\Lynda" | C:\Users\Lynda | Disables escape processing (e.g., in regex, file paths) |
| **3. Boolean** | True | is\_valid = True | True | Used in logical conditions |
|  | False | is\_done = False | False | Flags, logical operations |
| **4. Special** | None | data = None | None | Placeholder for no value / uninitialized variables |
| **5. Collection** | List | [1, 2, 3] | List of integers | Mutable ordered collection |
|  | Tuple | (1, 2, 3) | Tuple of integers | Immutable ordered collection |
|  | Set | {1, 2, 3} | Unordered, unique values | Removes duplicates, set operations |
|  | Dictionary | {"name": "Lynda"} | Key-value pair | Mapping, quick lookups |

**Escape Characters in String Literals**

| **Escape Code** | **Represents** | **Example** | **Output** |
| --- | --- | --- | --- |
| \\ | Backslash | "C:\\Files" | C:\Files |
| \' | Single quote | 'It\'s OK' | It's OK |
| \" | Double quote | "He said \"Hi\"" | He said "Hi" |
| \n | New line | "Line1\nLine2" | Multi-line |
| \t | Tab | "A\tB" | A B |
| \r | Carriage return | "123\r45" | 453 (overwrites part) |
| \b | Backspace | "AB\bC" | AC |
| \f | Form feed | "A\fB" | Platform-dependent |
| \a | Bell / Alert | "\a" | System sound (if supported) |
| \v | Vertical tab | "A\vB" | Platform-dependent |
| \ooo | Octal value | "\101" | A |
| \xhh | Hexadecimal value | "\x41" | A |

**Operators**

An **operator** is a **symbol** that performs an operation on one or more **operands** (variables or values). Python has a rich set of built-in operators to perform arithmetic, logical, relational, assignment, and more.

| **Category** | **Description** | **Example** |
| --- | --- | --- |
| Arithmetic | Perform mathematical operations | +, -, \*, /, % |
| Relational (Comparison) | Compare two values | ==, !=, <, >, <=, >= |
| Logical | Combine conditional expressions | and, or, not |
| Assignment | Assign values to variables | =, +=, -=, etc. |
| Bitwise | Work at the binary level | &, ` |
| Membership | Check for membership in a sequence | in, not in |
| Identity | Check if variables refer to same object | is, is not |
| Unary | Operate on one operand | -x, not x |

**Operator Types with Examples**

**1. Arithmetic Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Addition | 3 + 2 | 5 |
| - | Subtraction | 3 - 2 | 1 |
| \* | Multiplication | 3 \* 2 | 6 |
| / | Division | 3 / 2 | 1.5 |
| // | Floor Division | 3 // 2 | 1 |
| % | Modulus (remainder) | 3 % 2 | 1 |
| \*\* | Exponentiation | 2 \*\* 3 | 8 |

**2. Relational (Comparison) Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | 3 == 3 | True |
| != | Not equal to | 3 != 2 | True |
| > | Greater than | 3 > 2 | True |
| < | Less than | 3 < 2 | False |
| >= | Greater than or equal to | 3 >= 3 | True |
| <= | Less than or equal to | 2 <= 3 | True |

**3. Logical Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| and | True if both are True | True and False | False |
| or | True if at least one is True | True or False | True |
| not | Reverses result | not True | False |

**4. Assignment Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| = | Assign | x = 5 | x = 5 |
| += | Add and assign | x += 1 | x = x + 1 |
| -= | Subtract and assign | x -= 1 | x = x - 1 |
| \*= | Multiply and assign | x \*= 2 | x = x \* 2 |
| /= | Divide and assign | x /= 2 | x = x / 2 |
| //= | Floor divide and assign | x //= 2 | x = x // 2 |
| %= | Modulus and assign | x %= 2 | x = x % 2 |
| \*\*= | Exponent and assign | x \*\*= 2 | x = x \*\* 2 |

**5. Bitwise Operators**

| **Operator** | **Description** | **Example** | **Binary Result** |
| --- | --- | --- | --- |
| & | Bitwise AND | 5 & 3 | 1 |
| ` | Bitwise OR | `5 | 3` |
| ^ | Bitwise XOR | 5 ^ 3 | 6 |
| ~ | Bitwise NOT | ~5 | -6 |
| << | Zero Fill Left Shift | 5 << 1 | 10 |
| >> | Signed Right Shift | 5 >> 1 | 2 |

**6. Membership Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| in | Checks if value is in sequence | 'a' in 'cat' | True |
| not in | Checks if value is not in sequence | 'z' not in 'cat' | True |

**7. Identity Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| is | True if both refer to same object | a is b | True (if same memory) |
| is not | True if not the same object | a is not b | True (if different memory) |

**8. Unary Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| - | Unary minus | -5 | -5 |
| + | Unary plus | +5 | 5 |
| not | Logical NOT | not True | False |

**Operator Precedence**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** |  |
| () | Parentheses |  |
| \*\* | Exponentiation |  |
| +x  -x  ~x | Unary plus, unary minus, and bitwise NOT |  |
| \*  /  //  % | Multiplication, division, floor division, and modulus |  |
| +  - | Addition and subtraction |  |
| <<  >> | Bitwise left and right shifts |  |
| **Operator** | **Description** |  |
| & | Bitwise AND |  |
| ^ | Bitwise XOR |  |
| | | Bitwise OR |  |
| ==  !=  >  >=  <  <=  is  is not  in  not in | Comparisons, identity, and membership operators |  |
| not | Logical NOT |  |
| and | AND |  |
| or | OR |  |

**Punctuators**

**Punctuators** are symbols used to **organize code**, **define boundaries**, **enclose blocks**, or **separate statements**.

| **Punctuator** | **Name** | **Purpose / Use** | **Example** |
| --- | --- | --- | --- |
| () | Parentheses | Group expressions, function calls, tuples | print("Hello"), x = (1 + 2) |
| [] | Square Brackets | Define lists, indexing, slicing | my\_list[0], my\_list[1:3] |
| {} | Curly Braces | Define dictionaries, used in set formatting and f-strings | {"a": 1}, f"Hello {name}" |
| : | Colon | Starts indented block (if, loop, function), dictionary key-value separation | if x > 0:, {"a": 1} |
| , | Comma | Separates list elements, tuple items, arguments, dictionary entries | a, b = 1, 2, print(a, b) |
| . | Dot / Period | Access object or module members | string.upper(), math.pi |
| ; | Semicolon | Separate multiple statements on a single line (not recommended in Python) | x = 1; y = 2 |
| @ | At symbol / Decorator | Decorate functions or classes | @classmethod, @staticmethod |
| = | Assignment Operator | Assign value to a variable | x = 10 |
| -> | Arrow / Return Annotation | Specify the return type in function signatures (type hints) | def add(a: int, b: int) -> int: |
| ... | Ellipsis | Placeholder for incomplete code, extended slicing (NumPy, etc.) | def stub(): ... |
| := | Walrus Operator | Assignment expressions (Python 3.8+), assign and return value in one step | if (n := len(data)) > 0: |
| ' ' / " " | Single/Double Quotes | Define string literals | 'hello', "world" |
| \ | Backslash | Escape characters inside strings, continue lines | \"Hello\", print("Line 1\\nLine 2") |
| # | Hash | Add a comment line, ignored by interpreter | # This is a comment |

**Strings**

A **string** is a sequence of characters enclosed within **single quotes '...'**, **double quotes "..."**, or **triple quotes '''...''' / """..."""** for multi-line strings.

str1 = 'Hello'

str2 = "World"

str3 = '''This is

a multiline string.'''

**Purpose of Strings**

* Store and manipulate **textual data**.
* Represent **names**, **messages**, **file paths**, **URLs**, **text logs**, etc.
* Enable interaction with **users, files, databases, APIs**, etc.

| **Use Case** | **Example** |
| --- | --- |
| User input/output | input("Enter your name: "), print("Welcome") |
| Data parsing | Reading JSON, CSV, XML data |
| File processing | open("file.txt").read() |
| Web scraping/API calls | URLs, HTTP responses |
| Regex pattern matching | re.match(r"\d+", "123") |
| Logging & debugging | logger.info("Error occurred") |

**String Characteristics**

| **Property** | **Details** |
| --- | --- |
| Immutable | Strings can't be changed once created. Operations create new strings. |
| Indexed | Each character has a position (starting from 0). |
| Iterable | You can loop through characters in a string. |
| Unicode support | Python 3 strings support Unicode (e.g., emojis, multilingual text). |

**String Operations**

s = "Python"

# Indexing from the start

print(s[0]) # 'P'

# Indexing from the end

print(s[-1]) # 'n'

# Slicing

print(s[1:4]) # 'yth'

# Slicing from the start

print(s[:5]) # Pytho

# Slicing to the end

print(s[2:]) # thon

# Slicing from the end using negative indexing

print(s[-4:-2]) # th

Looping through a string

for x in "banana":

print(x)

String Length

a = "Hello, World!"

print(len(a)) # 13

**String Methods**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| capitalize() | Converts the first character to uppercase | "hello".capitalize() → 'Hello' |
| casefold() | Converts string to lowercase (more aggressive than lower()) | "HELLO".casefold() → 'hello' |
| center(width) | Centers string within given width | "hi".center(10) → ' hi ' |
| count(sub) | Returns number of times a substring appears | "banana".count("a") → 3 |
| encode() | Encodes string using specified encoding (default UTF-8) | "tëst".encode() |
| endswith(suffix) | Returns True if string ends with given suffix | "report.pdf".endswith(".pdf") → True |
| expandtabs(n) | Replaces \t with spaces (default 8) | "a\tb".expandtabs(4) → 'a b' |
| find(sub) | Returns first index of substring or -1 if not found | "hello".find("l") → 2 |
| format() | Formats string using {} placeholders | "Name: {}".format("Lynda") → 'Name: Lynda' |
| format\_map(dict) | Like format(), but takes a dictionary | "{name}".format\_map({"name": "Lynda"}) |
| index(sub) | Like find(), but raises ValueError if not found | "hello".index("e") → 1 |
| isalnum() | Returns True if all characters are alphanumeric | "abc123".isalnum() → True |
| isalpha() | Returns True if all characters are alphabetic | "abc".isalpha() → True |
| isascii() | Returns True if all characters are ASCII | "hello".isascii() → True |
| isdecimal() | True if all characters are decimal numbers (0–9) | "123".isdecimal() → True |
| isdigit() | True if all characters are digits (includes superscripts, etc.) | "²3".isdigit() → True |
| isidentifier() | True if valid Python identifier | "var1".isidentifier() → True |
| islower() | True if all letters are lowercase | "abc".islower() → True |
| isnumeric() | True if all characters are numeric | "½".isnumeric() → True |
| isprintable() | True if all characters are printable | "hello\n".isprintable() → False |
| isspace() | True if string contains only whitespace | " \t\n".isspace() → True |
| istitle() | True if each word is capitalized like a title | "This Is A Title".istitle() → True |
| isupper() | True if all letters are uppercase | "ABC".isupper() → True |
| join(iterable) | Joins iterable items with the string as separator | ".".join(["a", "b"]) → 'a.b' |
| ljust(width) | Left-justifies the string in a field of given width | "hi".ljust(5) → 'hi ' |
| lower() | Converts string to lowercase | "HI".lower() → 'hi' |
| lstrip() | Removes leading whitespace | " hi".lstrip() → 'hi' |
| maketrans() | Returns a translation mapping table | str.maketrans("ae", "12") |
| partition(sep) | Splits into 3 parts: before, sep, after | "a-b".partition("-") → ('a', '-', 'b') |
| replace(a, b) | Replaces all occurrences of a with b | "red apple".replace("red", "green") |
| rfind(sub) | Returns last occurrence index or -1 | "banana".rfind("a") → 5 |
| rindex(sub) | Like rfind() but raises ValueError if not found | "banana".rindex("a") → 5 |
| rjust(width) | Right-justifies the string | "hi".rjust(5) → ' hi' |
| rpartition(sep) | Splits from the right into 3 parts | "a-b-c".rpartition("-") → ('a-b', '-', 'c') |
| rsplit() | Splits from right; useful for limiting splits | "a-b-c".rsplit("-", 1) → ['a-b', 'c'] |
| rstrip() | Removes trailing whitespace | "hi ".rstrip() → 'hi' |
| split() | Splits on spaces or given separator | "a b".split() → ['a', 'b'] |
| splitlines() | Splits on newline characters | "a\nb".splitlines() → ['a', 'b'] |
| startswith(val) | Returns True if string starts with val | "hello".startswith("he") → True |
| strip() | Removes both leading and trailing whitespace | " hi ".strip() → 'hi' |
| swapcase() | Swaps lowercase to uppercase and vice versa | "AbC".swapcase() → 'aBc' |
| title() | Capitalizes the first letter of each word | "hello world".title() → 'Hello World' |
| translate(map) | Translates string based on translation table | "abc".translate(str.maketrans("a", "1")) → '1bc' |
| upper() | Converts to uppercase | "hello".upper() → 'HELLO' |
| zfill(width) | Pads string on the left with zeros | "42".zfill(5) → '00042' |

**F-strings** - Use it for readable formatting

txt = f"The price is {20 \* 59 : .2f} dollars"

print(txt)

**Datetime**

The datetime **package** in Python is not a separate library but a **built-in module** (datetime) that includes classes for manipulating dates and times. It is extensively used in **time-based applications**, **data analysis**, **logging**, **reporting**, and much more.

**Key Classes and Their Purpose**

| **Class** | **Purpose** |
| --- | --- |
| datetime.date | Represents a **date** (year, month, day) only |
| datetime.time | Represents a **time** (hour, minute, second, microsecond) only |
| datetime.datetime | Represents a full **timestamp** (date + time) |
| datetime.timedelta | Represents the **difference** between two dates/times |
| datetime.tzinfo | Base class for dealing with **timezone** info |
| datetime.timezone | A concrete class for **UTC offset-based timezone handling** |

**datetime.date – Calendar Date**

from datetime import date

today = date.today()

print(today) # e.g., 2025-05-08

# Create a specific date

d = date(2025, 12, 25)

print(d.year, d.month, d.day)

**Useful methods:**

* .today() – Current date
* .fromisoformat("YYYY-MM-DD")
* .weekday() – Monday = 0
* .isoformat() – YYYY-MM-DD

**datetime.time – Clock Time**

from datetime import time

t = time(14, 30, 0) # 2:30 PM

print(t.hour, t.minute, t.second)

**Useful attributes:**

* t.hour, t.minute, t.second, t.microsecond

**datetime.datetime – Full Date & Time**

from datetime import datetime

now = datetime.now()

dt = datetime(2025, 5, 8, 14, 30)

print(now.date(), now.time())

**Common methods:**

* .now() – Current local date & time
* .utcnow() – UTC time
* .fromisoformat() and .isoformat()
* .strftime(fmt) – Format to string
* .strptime(s, fmt) – Parse string to datetime

**datetime.timedelta – Time Difference**

from datetime import timedelta

delta = timedelta(days=5, hours=3)

future = datetime.now() + delta

print(future)

**Useful for:**

* Adding/subtracting days/hours
* Getting duration in seconds/days

**datetime.timezone – Time Zone Handling**

from datetime import timezone, timedelta

tz = timezone(timedelta(hours=5, minutes=30)) # IST (UTC+5:30)

now = datetime.now(tz)

**Date Formatting with strftime()**

| **Directive** | **Description** | **Example** |
| --- | --- | --- |
| %a | Weekday name, abbreviated | Wed |
| %A | Weekday name, full | Wednesday |
| %w | Weekday number, Sunday = 0 | 3 |
| %d | Day of the month (zero-padded) | 31 |
| %b | Month name, abbreviated | Dec |
| %B | Month name, full | December |
| %m | Month number (zero-padded) | 12 |
| %y | Year without century (zero-padded) | 18 |
| %Y | Full year | 2018 |
| %H | Hour (24-hour clock, zero-padded) | 17 |
| %I | Hour (12-hour clock, zero-padded) | 05 |
| %p | AM or PM | PM |
| %M | Minute (zero-padded) | 41 |
| %S | Second (zero-padded) | 08 |
| %f | Microsecond (zero-padded to 6 digits) | 548513 |
| %z | UTC offset in the form +HHMM or -HHMM | +0530 |
| %Z | Time zone name | CST |
| %j | Day of the year (001 to 366) | 365 |
| %U | Week number of the year (Sunday as first day), 00–53 | 52 |
| %W | Week number of the year (Monday as first day), 00–53 | 52 |
| %c | Locale’s appropriate date and time representation | Mon Dec 31 17:41:00 2018 |
| %C | Century (year divided by 100, truncated to int) | 20 |
| %x | Locale’s appropriate date representation | 12/31/18 |
| %X | Locale’s appropriate time representation | 17:41:00 |
| %% | A literal % character | % |
| %G | ISO 8601 year (may differ from %Y near year boundaries) | 2018 |
| %u | ISO 8601 weekday (1 = Monday, 7 = Sunday) | 1 |
| %V | ISO 8601 week number (01–53, Monday is the first day of the week) | 01 |

now = datetime.now()

formatted = now.strftime("%A, %d %B %Y %I:%M %p")

print(formatted)

**Real-Life Use Cases**

| **Use Case** | **Example** |
| --- | --- |
| Logging | Timestamp when an event occurs |
| Reports | Generate reports for a specific date range |
| Reminders | Notify user based on upcoming deadlines |
| Date Parsing | Parse input like "2025-12-31" into datetime |
| Calculations | Days between start and end dates |
| Time Zones | Convert local time to UTC and vice versa |

**Math**

**Built-in Math Functions**

| **Function** | **Description** | **Example** | **Output** |
| --- | --- | --- | --- |
| abs(x) | Returns the absolute value of a number | abs(-5) | 5 |
| round(x[, n]) | Rounds a number to n digits after the decimal | round(3.14159, 2) | 3.14 |
| max(iterable, ...) | Returns the largest of the input values | max(4, 7, 1) | 7 |
| min(iterable, ...) | Returns the smallest of the input values | min(4, 7, 1) | 1 |
| sum(iterable) | Returns the sum of all elements in an iterable | sum([1, 2, 3]) | 6 |
| pow(x, y[, z]) | Returns (x \*\* y) % z if z is provided, otherwise x \*\* y | pow(2, 3) or pow(2, 3, 5) | 8, 3 |
| divmod(a, b) | Returns a tuple (a // b, a % b) | divmod(9, 4) | (2, 1) |
| bin(x) | Returns the binary representation of an integer | bin(5) | '0b101' |
| oct(x) | Returns the octal representation of an integer | oct(8) | '0o10' |
| hex(x) | Returns the hexadecimal representation of an integer | hex(255) | '0xff' |
| int(x) | Converts a value to an integer | int(3.6) | 3 |
| float(x) | Converts a value to a floating point number | float("4.2") | 4.2 |
| complex(x, y) | Returns a complex number (x + yj) | complex(2, 3) | (2+3j) |
| bool(x) | Converts a value to Boolean | bool(0) | False |
| all(iterable) | Returns True if all elements in the iterable are true | all([True, 1, "a"]) | True |
| any(iterable) | Returns True if any element in the iterable is true | any([0, "", 5]) | True |

**Commonly Used Functions and Constants in math Module**

| **Category** | **Function / Constant** | **Description** | **Example / Output** |
| --- | --- | --- | --- |
| **Constants** | math.pi | π = 3.141592... | math.pi → 3.141592653589793 |
|  | math.e | Euler’s number = 2.718281... | math.e → 2.718281828459045 |
|  | math.inf | Represents infinity | math.inf → inf |
|  | math.nan | Not a number (undefined result) | math.nan → nan |

**Number Operations**

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| math.ceil(x) | Ceiling value (smallest integer ≥ x) | math.ceil(4.2) → 5 |
| math.floor(x) | Floor value (largest integer ≤ x) | math.floor(4.7) → 4 |
| math.trunc(x) | Truncates decimal (drops fractional part) | math.trunc(-3.9) → -3 |
| math.fabs(x) | Absolute value (float result) | math.fabs(-7) → 7.0 |
| math.factorial(x) | Factorial of x (x!) | math.factorial(5) → 120 |
| math.copysign(x, y) | Copy sign from y to x | math.copysign(3, -1) → -3.0 |
| math.isfinite(x) | Checks if x is finite | math.isfinite(math.inf) → False |

**Power and Logarithmic Functions**

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| math.pow(x, y) | x raised to power y (float result) | math.pow(2, 3) → 8.0 |
| math.sqrt(x) | Square root of x | math.sqrt(16) → 4.0 |
| math.exp(x) | e raised to power x | math.exp(1) → 2.718... |
| math.log(x) | Natural logarithm (base e) of x | math.log(10) → 2.302... |
| math.log10(x) | Base-10 logarithm of x | math.log10(100) → 2.0 |
| math.log2(x) | Base-2 logarithm of x | math.log2(8) → 3.0 |

**Trigonometric Functions**

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| math.sin(x) | Sine of x radians | math.sin(math.pi/2) → 1.0 |
| math.cos(x) | Cosine of x radians | math.cos(0) → 1.0 |
| math.tan(x) | Tangent of x radians | math.tan(math.pi/4) → 1.0 |
| math.asin(x) | Arcsine of x in radians | math.asin(1) → 1.57 (π/2) |
| math.acos(x) | Arccosine of x in radians | math.acos(1) → 0.0 |
| math.atan(x) | Arctangent of x in radians | math.atan(1) → 0.785 (π/4) |
| math.degrees(x) | Radians to degrees | math.degrees(math.pi) → 180.0 |
| math.radians(x) | Degrees to radians | math.radians(180) → π |

**Other Functions**

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| math.gcd(x, y) | Greatest common divisor | math.gcd(12, 18) → 6 |
| math.lcm(x, y) | Least common multiple (Python 3.9+) | math.lcm(4, 6) → 12 |
| math.comb(n, k) | Combinations: n choose k (Python 3.8+) | math.comb(5, 2) → 10 |
| math.perm(n, k) | Permutations: nPk (Python 3.8+) | math.perm(5, 2) → 20 |
| math.isclose(a, b) | Check if two floats are close in value | math.isclose(0.1+0.2, 0.3) → True |

**Data Types**

In Python, **datatypes** define the type of a variable and determine what kind of operations can be performed on it. Python is **dynamically typed**, so you don’t need to declare the type explicitly—it’s inferred at runtime.

**Purpose of Data Types**

* Categorize data (numbers, text, etc.)
* Enable appropriate operations (e.g., arithmetic on numbers, slicing on strings)
* Allow memory and type safety

| **Data Type** | **Category** | **Description** | **Example** |
| --- | --- | --- | --- |
| int | Numeric | Whole numbers, positive or negative | x = 10 |
| float | Numeric | Numbers with decimals | x = 3.14 |
| complex | Numeric | Complex numbers with real and imaginary parts | x = 3 + 4j |
| bool | Boolean | Represents True or False | x = True |
| str | Text | Sequence of Unicode characters (immutable) | x = "Hello" |
| list | Sequence | Ordered, mutable, allows duplicates | x = [1, 2, 3] |
| tuple | Sequence | Ordered, immutable, allows duplicates | x = (1, 2, 3) |
| range | Sequence | Represents a sequence of numbers | x = range(5) |
| dict | Mapping | Key-value pairs, unordered and mutable | x = {"a": 1, "b": 2} |
| set | Set | Unordered, no duplicates | x = {1, 2, 3} |
| frozenset | Set | Immutable version of set | x = frozenset([1, 2, 3]) |
| bytes | Binary | Immutable sequence of bytes | x = b"abc" |
| bytearray | Binary | Mutable version of bytes | x = bytearray([65, 66, 67]) |
| memoryview | Binary | Memory access to byte data without copying | x = memoryview(b"abc") |
| NoneType | Special | Represents absence of a value | x = None |

**type() FUNCTION**

Used to check the **data type** of an object.

print(type(42)) # <class 'int'>

print(type([1, 2, 3])) # <class 'list'>

**Arrays**

* An **array** is a data structure that stores **multiple elements of the same data type**.
* Unlike Python’s flexible list, array.array ensures **type safety**, **better memory efficiency**, and **fast computation**.

import array

arr = array.array('i', [1, 2, 3]) # 'i' stands for integer type

**Type Codes:**

| **Code** | **Type** |
| --- | --- |
| 'i' | signed int |
| 'f' | float |
| 'd' | double |

**Array Methods:**

| **Method** | **Purpose** |
| --- | --- |
| append(x) | Adds x to the end |
| extend(array2) | Adds all elements from array2 |
| pop(index) | Removes and returns element at index (default: last) |
| remove(x) | Removes the **first occurrence** of x |
| index(x) | Returns index of first occurrence of x |
| count(x) | Returns count of x in array |
| reverse() | Reverses the array |
| copy() | Returns a copy of the array |
| sort() (use list) | Convert to list first: sorted(list(array)) for sorting |

**Why Use Arrays?**

* **Faster** than lists for numerical operations.
* **Memory-efficient**
* **Type-safe**: throws error if incorrect data type is inserted.
* Ideal for **sorting, searching, and numerical algorithms**.

**Conditional Statements**

**Conditionals**

Conditionals control the flow of execution based on **boolean conditions**.

**Types of Conditionals:**

**1. if Statement**

x = 10

if x > 5:

print("Greater than 5")

**2. if-else**

if x > 5:

print("Greater")

else:

print("Smaller or equal")

**3. if-elif-else**

if x > 10:

print("Big")

elif x == 10:

print("Equal")

else:

print("Small")

**4. Nested if**

if x > 5:

if x < 15:

print("Between 5 and 15")

**5. Ternary Expression**

result = "Even" if x % 2 == 0 else "Odd"

**Loops**

**Purpose:**

To repeat a block of code multiple times.

**Types of Loops:**

**1. while Loop (Entry Check)**

count = 0

while count < 5:

print(count)

count += 1

**2. for Loop**

for i in range(5):

print(i)

**3. Nested Loops**

for i in range(2):

for j in range(3):

print(i, j)

**Loop Control Statements:**

| **Statement** | **Purpose** |
| --- | --- |
| break | Exits the loop immediately |
| continue | Skips the current iteration and moves to the next |
| pass | Placeholder for empty loops or blocks |

**range() Function**

Used to generate a **sequence of numbers** for loops.

| **Syntax** | **Meaning** |
| --- | --- |
| range(stop) | 0 to stop-1 |
| range(start, stop) | start to stop-1 |
| range(start, stop, step) | step-incremented from start to stop-1 |

list(range(1, 10, 2)) # [1, 3, 5, 7, 9]

**Random Numbers**

Use random module:

import random

random.randint(1, 100) # Random int between 1 and 100

random.choice([1, 2, 3]) # Random choice from list

**Coding Best Practices**

| **Practice** | **Reason** |
| --- | --- |
| Use **4 spaces** for indentation | Python standard (PEP 8) |
| Avoid mixing tabs and spaces | Causes indentation errors |
| Avoid deeply nested code | Makes code harder to read |
| Use **ternary** for simple logic | More readable |
| Add **comments** for complex logic | Helps future maintainers |

**OOPs**

Python supports both **procedural** and **object-oriented programming (OOP)** paradigms. OOP organizes code using **classes** and **objects**, making it reusable, modular, shows essential features only to users and easier to manage in large applications.

| **Concept** | **Description** |
| --- | --- |
| **Class** | Blueprint for creating objects (defines properties and behaviors) |
| **Object** | Instance of a class; the real entity with actual data It has 3 states: Behaviour, Identity, State |
| **Constructor** | \_\_init\_\_() method used for initializing object properties |
| **Encapsulation** | Hides internal state; access via methods; restricts direct access to variables |
| **Abstraction** | Hides complex implementation, shows only essential features |
| **Inheritance** | Allows one class (child) to inherit properties and methods from another (parent) |
| **Polymorphism** | Ability to use the same method name for different classes or data types |

**Creating a**

**Class and Object**

class Person:

def \_\_init\_\_(self, name, age): # Constructor

self.name = name

self.age = age

def greet(self):

print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Creating an object

p1 = Person("Alice", 25)

p1.greet()

**Encapsulation Example**

class BankAccount:

def \_\_init\_\_(self, balance):

self.\_\_balance = balance # Private variable

def deposit(self, amount):

self.\_\_balance += amount

def get\_balance(self):

return self.\_\_balance

acc = BankAccount(1000)

acc.deposit(500)

print(acc.get\_balance()) # Output: 1500

**Inheritance Example**

class Animal:

def speak(self):

return "Animal Sound"

class Dog(Animal): # Inherits Animal

def speak(self):

return "Bark"

d = Dog()

print(d.speak()) # Output: Bark

**Polymorphism Example**

class Cat:

def sound(self):

return "Meow"

class Dog:

def sound(self):

return "Bark"

def make\_sound(animal):

print(animal.sound())

make\_sound(Cat())

make\_sound(Dog())

**Abstraction with abc Module**

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* self.radius \* self.radius

c = Circle(5)

print(c.area()) # Output: 78.5

**Naming Convections**

* Method name is a verb
* Class name is a noun
* Meaningful names to variables, functions, classes, modules, etc.
* Camel case for double name identifiers