

Python Programming

Contents



- Python Programming
- Python `try...except`
- Errors in Python



Introduction to Python

Python is a high-level, interpreted programming language created by **Guido van Rossum** and first released in **1991**.

Why Python is so popular [Back to top](#)

-  **Readable, clear syntax** → easy for beginners.
-  **Rich ecosystem** → huge number of libraries.

- 🤖 **AI & Data Science ready** → works perfectly with NumPy, Pandas, TensorFlow, PyTorch, Scikit-learn.
- 💻 **Cross-platform** → runs everywhere (Windows, Linux, macOS).
- 🌐 **Large community** → tons of tutorials, forums, and support.

Comparison with Other Languages

Feature	Python	C++	Java
Syntax	Simple, concise	Complex, verbose	Verbose
Typing	Dynamic	Static	Static
Compilation	Interpreted	Compiled	Compiled (bytecode)
Speed	Slower	Faster	Fast
Use Cases	AI, Data Science	Systems, Games	Enterprise Apps

Interpreted vs Compiled

- **Interpreted:** Code runs line by line → easier to debug (**Python**).
- **Compiled:** Code is translated before execution (**C++, Java**).

Why Python Matters for Artificial Intelligence?

- ⚡ Python is the **default language for AI research and production**.
- 🧠 Frameworks like **TensorFlow, PyTorch, Keras** are built around Python.
- 📊 Works hand in hand with **NumPy & Pandas** for numerical computing and data analysis.
- ⚖️ Quick prototyping: scientists can test AI models fast.

👉 Without Python, AI would not have grown so fast because it provides the tools and simplicity researchers need.

How to Create a Python Script

What is a Python script?

A *Python script* is simply a plain text file containing Python code. The file usually has the `.py` extension and can be executed directly by the Python interpreter.

Why use scripts?

- Scripts are ideal for automating tasks (data processing, file operations).
- They can be shared and reused easily.
- They work well for production workflows and scheduled jobs.
- Unlike Jupyter Notebooks, scripts do not contain outputs or rich formatting—they focus purely on code.

When to prefer scripts over notebooks?

Use Script (<code>.py</code>)	Use Notebook (<code>.ipynb</code>)
Production pipelines	Exploratory data analysis
Batch processing	Interactive visualizations
Deployment (cron jobs, servers)	Teaching, tutorials, step-by-step demos
Simple automation	Sharing results with narrative text

Steps to create a script:

1. Open any text editor or code editor (e.g., Notepad, Visual Studio Code, Sublime Text).
2. Type your Python code. For example:

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

Hello, Python World!

3. Save the file with a `.py` extension, such as `myscript.py`.

Tip:

- Use an editor with syntax highlighting to make your code easier to read.
- VS Code is highly recommended because it shows errors and supports debugging.

Common Use Cases:

- Automating data downloads.
- Running ETL (Extract, Transform, Load) workflows.
- Training machine learning models.
- Processing files in bulk.
- Deploying data pipelines.

How to Run a Python Script

From terminal or command prompt:

To execute your `.py` file, open your terminal or command prompt and type the following command:

```
bash python filename.py
```

Example:

```
bash python myscript.py
```

This will run your script, and any output (e.g., `print()` results) will appear in the console.

Tip:

- If you have multiple Python versions installed, you might need to use `python3` instead of `python`.
- You can also run scripts directly in IDEs like VS Code by clicking the 'Run' button.

Alternative:

In interactive environments like Jupyter Notebook, you can use:

```
%run myscript.py
```

Hello, Python World!

Understanding `if __name__ == "__main__":`

When you run a Python file, Python automatically gives the special variable `__name__` a value:

- If the file is **run directly** → `__name__ == "__main__"`.
- If the file is **imported as a module** → `__name__` is set to the **module name**.

Why use it?

- Prevents certain code from running when the file is imported.
- Keeps code **organized**: reusable functions/classes vs. main script logic.
- It is a **best practice** in Python scripting.

Does a script work without it?

- Yes, Python scripts will run fine without it.
- But **without it**, everything in the file runs even when imported → can cause unexpected behavior.

Example

```
def greet():  
    print("Hello from greet() function!")  
  
print("This line runs always, even if imported.")  
  
if __name__ == "__main__":  
    print("Script is run directly → executing main part...")  
    greet()
```

```
This line runs always, even if imported.  
Script is run directly → executing main part...  
Hello from greet() function!
```

Explanation:

- If you run this file directly → both the top-level print and the `if __main__` block execute.
- If you **import this file** into another script → only the top-level print runs, the `if __main__` block is skipped.

👉 This prevents functions from running accidentally when the file is used as a **module**.

Python Comments

What are comments?

Comments are pieces of text in your code that are **ignored by the Python interpreter**. They help explain what your code is doing, making it easier to read and maintain.

Types of Comments:

- **Single-line comments:** Start with `#`.
- **Inline comments:** Placed after code on the same line.
- **Multi-line comments:** Use triple quotes (`'''` or `"""`). Although technically they are multi-line strings, they can serve as block comments or documentation.

Examples:

```
# This is a single-line comment  
  
print("Hello") # This is an inline comment  
  
'''  
This is a multi-line comment or docstring.  
It can span multiple lines.  
Useful for documentation.  
'''
```

Tips for Writing Good Comments:

- Keep comments short and relevant.
- Use them to explain *why* something is done, not *what* (the code itself shows the *what*).
- Update comments if you change your code.

When to Use Each Type:

Type	Use Case
Single-line	Quick notes or explanations.
Inline	Clarify a specific line of code.
Multi-line	Longer descriptions or temporary block comments.

Python Variables

What is a variable?

A variable is a **named container** used to store data in memory.

- In Python, you do **not** declare the type explicitly. The interpreter infers it automatically.
- Variable names are **case-sensitive**.

Naming Rules:

- Must start with a letter or underscore `_`
- Can contain letters, numbers, and underscores
- Cannot use reserved keywords (like `if`, `for`, `class`)

Examples:

```
x = 5           # integer
y = "Hello"     # string
is_active = True # boolean
_value = 3.14    # float
```

Best Practice: Use descriptive names:

```
user_age = 30
total_price = 99.99
```

Global Variables and `global` Keyword

Variables in Python have different **scopes** (places where they can be accessed).

Types of Scope:

- **Local:** Defined inside a function and used only there.
- **Global:** Defined outside functions and accessible everywhere.

Example of Global Variable:

```
x = 10 # Global variable

def show():
    print("x inside function:", x)

show()
print("x outside:", x)
```

```
x inside function: 10
x outside: 10
```

Problem when modifying global inside a function:

```
count = 0

def update():
    count += 1 # Error! Python thinks count is local
    print(count)
```

This gives an error because Python treats `count` as a new local variable.

Solution: Use `global` keyword:


```
count = 0

def update():
    global count
    count += 1
    print("Updated count:", count)

update()
```

Updated count: 1

When to use `global`:

- When you want to **modify** a global variable from inside a function.
- Avoid overusing it; prefer returning values instead.

Best Practice:

- Keep global variables to a minimum.
- Use function parameters and return values for better readability and testing.

Tip:

- Use `globals()` to inspect all global variables.

```
print(globals())
```

```
{'__name__': '__main__', '__doc__': 'Module created for script run in IPython', '__pac
```

Python Casting

What is Casting?

Casting means **explicitly converting** one data type to another.

Why use casting?

- To ensure the right type when performing operations.

- To avoid errors when combining different types.

Examples of functions:

- `int()` – Convert to integer.
- `float()` – Convert to float.
- `str()` – Convert to string.
- `bool()` – Convert to boolean.

Example:

```
x = int(3.7)           # Converts float 3.7 to integer 3
y = float("4.2")       # Converts string to float
z = str(10)            # Converts integer to string
print(x, y, z)
```

3 4.2 10

Python Booleans

What are Booleans?

Booleans are a data type that can hold only two values:

- `True`
- `False`

Why use Booleans?

- To control program logic and decisions.
- To evaluate conditions in `if`, `while`, and other statements.

Common Boolean Operators:

Operator	Meaning	Example
and	True if both operands are True	True and False -> False
or	True if at least one is True	True or False -> True
not	Inverts the value	not True -> False

Example Usage:

```
x = True
y = False
print(x and y)  # False
print(x or y)   # True
print(not x)    # False
```

False
True
False

These operators are essential for making decisions and controlling flow in Python programs.

Python Operators

Operators are special symbols used to perform operations on variables and values.

Arithmetic Operators: Used for numeric calculations.

Operator	Description	Example (<code>a=5</code> , <code>b=2</code>)	Result
<code>+</code>	Addition	<code>a + b</code>	<code>7</code>
<code>-</code>	Subtraction	<code>a - b</code>	<code>3</code>
<code>*</code>	Multiplication	<code>a * b</code>	<code>10</code>
<code>/</code>	Division	<code>a / b</code>	<code>2.5</code>
<code>//</code>	Floor Division	<code>a // b</code>	<code>2</code>
<code>%</code>	Modulus (Remainder)	<code>a % b</code>	<code>1</code>
<code>**</code>	Exponentiation	<code>a ** b</code>	<code>25</code>

Comparison Operators: Used to compare values, returning `True` or `False`.

Operator	Description	Example
<code>==</code>	Equals	<code>a == b</code> (False)
<code>!=</code>	Not equals	<code>a != b</code> (True)
<code>></code>	Greater than	<code>a > b</code> (True)
<code><</code>	Less than	<code>a < b</code> (False)

Logical Operators: Used to combine conditional statements.

Operator	Description	Example
<code>and</code>	True if both conditions are True	<code>(a > 2) and (b < 3)</code> (True)
<code>or</code>	True if at least one is True	<code>(a < 2) or (b < 3)</code> (True)
<code>not</code>	Inverts the boolean value	<code>not (a == b)</code> (True)

Example Usage:

```
a = 5
b = 2
print("Addition:", a + b)
print("Greater than:", a > b)
print("Logical AND:", (a > 2) and (b < 3))
```

```
Addition: 7
Greater than: True
Logical AND: True
```

Python Lists

Lists are one of the most commonly used data structures in Python.

What is a List?

- **Ordered:** The order in which you add items is preserved.
- **Mutable:** You can change, add, or remove items after creation.
- **Allows Duplicates:** The same value can appear more than once.

When to Use Lists?

- When you need to store a collection of items in a specific order.
- When you might need to modify the data later (unlike tuples).
- For example: a shopping list, a sequence of numbers, etc.

Common Operations:

- `append()`: Add an item to the end.
- `insert()`: Insert an item at a specific position.
- `remove()`: Remove the first occurrence of an item.
- `pop()`: Remove and return an item.
- `len()`: Get the length of the list.
- Indexing: Access items by position.

- Slicing: Get a sublist.

```
# Creating a list
fruits = ["apple", "banana", "cherry"]

# Adding an item
fruits.append("orange")

# Inserting an item at index 1
fruits.insert(1, "mango")

# Removing an item
fruits.remove("banana")

# Accessing elements
print("First item:", fruits[0])

# Slicing
print("First two items:", fruits[:2])

# Length of the list
print("Length:", len(fruits))

# Display the final list
print("Updated list:", fruits)
```

```
First item: apple
First two items: ['apple', 'mango']
Length: 4
Updated list: ['apple', 'mango', 'cherry', 'orange']
```

Python Slicing

Slicing allows you to extract parts of sequences like lists, tuples, and strings.

Basic Syntax:

```
sequence[start:end:step]
```

- **start**: the index where slicing begins (inclusive).
- **end**: the index where slicing stops (exclusive).
- **step**: how many items to skip each time (default is 1).

Slice	Description	Example (<code>text = 'Python'</code>)	Result
<code>[:]</code>	Copy the whole sequence	<code>text[:]</code>	<code>'Python'</code>
<code>[start : end]</code>	Subsequence from start to end-1	<code>text[0:2]</code>	<code>'Py'</code>
<code>[start :]</code>	From start index to the end	<code>text[2:]</code>	<code>'thon'</code>
<code>[: end]</code>	From the beginning to end-1	<code>text[: 4]</code>	<code>'Pyth'</code>
<code>[: :]</code>	Full copy (same as <code>[:]</code>)	<code>text[: :]</code>	<code>'Python'</code>
<code>[: : step]</code>	Take elements with step	<code>text[: : 2]</code>	<code>'Pto'</code>
<code>[: : -1]</code>	Reverse the sequence	<code>text[: : -1]</code>	<code>'nohtyP'</code>

```
# Basic slicing examples
text = "Python"

print("text[ :]      =>", text[ : ])      # Copy the whole sequence
print("text[0:2]    =>", text[0:2])      # Subsequence from index 0 to 1
print("text[2: ]    =>", text[2: ])      # From index 2 to the end
print("text[ : 4]   =>", text[ : 4])      # From beginning to index 3
print("text[ : : ]   =>", text[ : : ])    # Full copy (same as [ : ])
print("text[ : : 2]  =>", text[ : : 2])   # Every second character
print("text[ : : -1] =>", text[ : : -1])  # Reverse the sequence
```

```
text[ :]      => Python
text[0:2]    => Py
text[2: ]    => thon
text[ : 4]   => Pyth
text[ : : ]   => Python
text[ : : 2]  => Pto
text[ : : -1] => nohtyP
```

```
# Example usage with a list
nums = [10, 20, 30, 40, 50]

print("nums[1:4]    =>", nums[1:4])      # [20, 30, 40]
print("nums[ : 3]   =>", nums[ : 3])      # [10, 20, 30]
print("nums[2: ]    =>", nums[2: ])      # [30, 40, 50]
print("nums[ : : 2]  =>", nums[ : : 2])   # [10, 30, 50]
print("nums[ : : -1] =>", nums[ : : -1])  # [50, 40, 30, 20, 10]
```

```
nums[1:4]    => [20, 30, 40]
nums[:3]     => [10, 20, 30]
nums[2:]     => [30, 40, 50]
nums[:2]     => [10, 30, 50]
nums[::-1]   => [50, 40, 30, 20, 10]
```

Python Tuples

Tuples are collections that are:

- **Ordered:** Items maintain the order they are defined.
 - **Immutable:** Once created, items cannot be added, removed, or modified.
 - **Heterogeneous:** Can store different data types together.
-

Why Use Tuples?

- **Data safety:** Prevents accidental modification.
 - **Performance:** Tuples are faster than lists in many operations.
 - **Hashable:** Can be used as dictionary keys and stored in sets.
-

Tuple Creation

- Using parentheses: `(1, 2, 3)`
 - Without parentheses: `1, 2, 3`
 - Single-element tuple: `(5,)` (comma is required)
 - Empty tuple: `()`
-

Common Use Cases

- Returning multiple values from a function.
- Representing fixed collections (coordinates, RGB colors, etc.).
- Storing read-only data.

Tuple Operations

- Indexing: `t[0]`
- Slicing: `t[1:3]`
- Iteration: `for item in t`
- Membership: `x in t`
- Concatenation: `t1 + t2`
- Repetition: `t * 3`

Immutability

Tuples cannot be changed directly, but they can contain mutable objects (like lists) that can be modified.

Example:

```

# Creating tuples
t1 = (1, 2, 3)
t2 = 4, 5, 6           # parentheses optional
single = (42,)         # single-element tuple
empty = ()             # empty tuple

print("t1:", t1)
print("t2:", t2)
print("single:", single)
print("empty:", empty)

# Indexing and slicing
print("t1[0]:", t1[0])
print("t1[-1]:", t1[-1])
print("t1[1:3]:", t1[1:3])

# Tuple unpacking
x, y, z = t1
print("Unpacked:", x, y, z)

# Concatenation and repetition
print("t1 + t2:", t1 + t2)
print("t1 * 2:", t1 * 2)

# Membership test
print("2 in t1:", 2 in t1)
print("99 in t1:", 99 in t1)

# Nested tuple with mutable object
nested = (1, [10, 20], 3)
print("Before:", nested)
nested[1].append(30) # allowed because list is mutable
print("After:", nested)

# Length of tuple
print("Length of t1:", len(t1))

```

```

t1: (1, 2, 3)
t2: (4, 5, 6)
single: (42,)
empty: ()
t1[0]: 1
t1[-1]: 3
t1[1:3]: (2, 3)
Unpacked: 1 2 3
t1 + t2: (1, 2, 3, 4, 5, 6)
t1 * 2: (1, 2, 3, 1, 2, 3)
2 in t1: True
99 in t1: False
Before: (1, [10, 20], 3)
After: (1, [10, 20, 30], 3)
Length of t1: 3

```

Python Sets

Sets are collections that are:

- **Unordered**: Items are not stored in a specific order.
 - **Unindexed**: No positional access (no `set[0]`).
 - **Unique**: Duplicates are automatically removed.
 - **Mutable**: You can add and remove elements.
-

Why Use Sets?

- To remove duplicates from a list.
 - To perform mathematical set operations: union, intersection, difference.
 - For fast membership testing (`in`).
-

Creating Sets

- Using curly braces: `{1, 2, 3}`
 - Using `set(iterable)`
 - Empty set must be created with `set()` (not `{}` because `{}` is a dict).
-

Common Operations

- `add(x)` → add a single element.
 - `update([x, y])` → add multiple elements.
 - `remove(x)` → remove element, error if not found.
 - `discard(x)` → remove element, no error if not found.
 - `pop()` → remove a random element.
 - `clear()` → remove all elements.
-

Set Algebra

- `union()` or `|` → combine sets.
- `intersection()` or `&` → common elements.
- `difference()` or `-` → elements only in the first set.
- `symmetric_difference()` or `^` → elements in either but not both.

Example:

```
# Creating sets with duplicates
numbers = {1, 2, 3, 3, 2}
print("Initial set:", numbers) # duplicates removed

# Adding elements
numbers.add(4)
numbers.update([5, 6])
print("After adding:", numbers)

# Membership testing
print("Contains 2?", 2 in numbers)
print("Contains 10?", 10 in numbers)

# Removing elements
numbers.remove(3) # raises error if not found
numbers.discard(10) # no error even if not found
print("After removals:", numbers)

# Pop removes a random element
removed = numbers.pop()
print("Removed element:", removed)
print("Set after pop:", numbers)

# Clear all elements
numbers.clear()
print("After clear:", numbers)
```

```
Initial set: {1, 2, 3}
After adding: {1, 2, 3, 4, 5, 6}
Contains 2? True
Contains 10? False
After removals: {1, 2, 4, 5, 6}
Removed element: 1
Set after pop: {2, 4, 5, 6}
After clear: set()
```

```
# Set algebra examples
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

print("a union b:", a | b)
print("a intersection b:", a & b)
print("a difference b:", a - b)
print("b difference a:", b - a)
print("a symmetric_difference b:", a ^ b)
```

```
a union b: {1, 2, 3, 4, 5, 6}
a intersection b: {3, 4}
a difference b: {1, 2}
b difference a: {5, 6}
a symmetric_difference b: {1, 2, 5, 6}
```

Python Dictionaries

Dictionaries are collections used to store data in **key-value pairs**.

Properties

- **Ordered (since Python 3.7):** Items maintain insertion order.
- **Mutable:** You can add, change, or remove items.
- **Keys must be unique and immutable:** Typically strings, numbers, or tuples.
- **Values can be any type:** Strings, numbers, lists, other dictionaries, etc.

Why Use Dictionaries?

- Fast access to data via keys.
- Perfect for structured data like records, configurations, or mappings.

Common Operations

- Access value: `mydict["key"]`

- Add or update: `mydict["key"] = value`
 - Remove key: `del mydict["key"]`
 - Remove safely: `mydict.pop("key", default)`
 - Get keys: `mydict.keys()`
 - Get values: `mydict.values()`
 - Get items (pairs): `mydict.items()`
 - Check membership: `'key' in mydict`
-

Nested Dictionaries

- Dictionaries can contain other dictionaries to represent complex structures.
-

Example:

```
# Creating a dictionary
person = {"name": "Alice", "age": 30}
print("Initial:", person)

# Accessing values
print("Name:", person["name"])

# Updating a value
person["age"] = 31

# Adding a new key-value pair
person["city"] = "Kuala Lumpur"
print("After update:", person)

# Getting keys, values, items
print("Keys:", list(person.keys()))
print("Values:", list(person.values()))
print("Items:", list(person.items()))

# Membership test
print("Has 'age'?", "age" in person)
print("Has 'salary'?", "salary" in person)

# Removing a key
del person["city"]
print("After removing city:", person)

# Using pop with default
age = person.pop("age", None)
print("Popped age:", age)
print("After pop:", person)
```

```
Initial: {'name': 'Alice', 'age': 30}
Name: Alice
After update: {'name': 'Alice', 'age': 31, 'city': 'Kuala Lumpur'}
Keys: ['name', 'age', 'city']
Values: ['Alice', 31, 'Kuala Lumpur']
Items: [('name', 'Alice'), ('age', 31), ('city', 'Kuala Lumpur')]
Has 'age'? True
Has 'salary'? False
After removing city: {'name': 'Alice', 'age': 31}
Popped age: 31
After pop: {'name': 'Alice'}
```

```
# Nested dictionaries
students = {
    "s1": {"name": "Ali", "grade": "A"},
    "s2": {"name": "Sara", "grade": "B"}
}

print("All students:", students)
print("Student s1 name:", students["s1"]["name"])
print("Student s2 grade:", students["s2"]["grade"])
```

```
All students: {'s1': {'name': 'Ali', 'grade': 'A'}, 's2': {'name': 'Sara', 'grade': 'B'}}
Student s1 name: Ali
Student s2 grade: B
```

Python If...Else

Conditional statements let you control the flow of your program depending on conditions.

Syntax:

- `if` checks a condition.
- `elif` checks another condition if previous ones were False.
- `else` runs if none of the conditions were True.

Comparison Operators:

Operator	Description
<code>==</code>	Equals
<code>!=</code>	Not equal
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal
<code><=</code>	Less than or equal

Example:

```
x = 7

if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x equals 5")
else:
    print("x is less than 5")
```

x is greater than 5

More Example: Using logical operators:

```
y = 10
if x > 5 and y > 5:
    print("Both x and y are greater than 5")
```

Both x and y are greater than 5

Python Match

The **match** statement, introduced in Python 3.10, is a powerful control structure for **pattern matching**.

When to Use:

- Replacing long chains of `if/elif/else`.
- Matching different types of data (numbers, strings, structures).
- Improving code readability.

Advantages:

- Cleaner and more expressive.
- Supports destructuring complex data (like tuples and dictionaries).
- Allows wildcard `_` for default cases.

Example 1: Matching Integers

```
status = 404

match status:
    case 200:
        print("OK")
    case 404:
        print("Not Found")
    case _:
        print("Other status")
```

Not Found

Example 2: Matching Strings

```
command = "start"

match command:
    case "start":
        print("Starting process...")
    case "stop":
        print("Stopping process...")
    case _:
        print("Unknown command")
```

Starting process...

Example 3: Matching Tuples

```
point = (0, 10)

match point:
    case (0, y):
        print(f"Y-axis at {y}")
    case (x, 0):
        print(f"X-axis at {x}")
    case (x, y):
        print(f"Point at ({x}, {y})")
```

Y-axis at 10

Example 4: Matching with Guards (conditions)

```
value = 15

match value:
    case x if x < 10:
        print("Less than 10")
    case x if x < 20:
        print("Between 10 and 19")
    case _:
        print("20 or more")
```

Between 10 and 19

Notes:

- Use `_` to match any value (wildcard).
- You can destructure complex data structures.
- Always include a default `case _` to handle unexpected values.

Common Use Cases:

- HTTP status handling.
- Command dispatching.
- Data validation and routing logic.
- Replacing nested `if-else` blocks.

Python While Loops

While loops repeatedly execute a block of code as long as the condition is `True`.

When to use:

- You do not know in advance how many times to repeat.
- You loop until a condition changes (e.g., waiting for user input).

Syntax:

```
while condition:
    # code block
```

Example:

```
count = 0
while count < 3:
    print("Count:", count)
    count += 1
```

Count: 0
Count: 1
Count: 2

Example with `break`: Stops the loop early.

```
i = 0
while True:
    print("i:", i)
    if i >= 2:
        break
    i += 1
```

i: 0
i: 1
i: 2

Example with `continue`: Skips to the next iteration.

```
n = 0
while n < 5:
    n += 1
    if n == 3:
        continue
    print("n:", n)
```

n: 1
n: 2
n: 4
n: 5

Python For Loops

For loops iterate over items of a sequence (like a list, tuple, or string).

When to use:

- You know how many elements to iterate.
- You want to process each element.

Syntax:

```
for item in sequence:  
    # code block
```

Example:

```
for fruit in ["apple", "banana", "cherry"]:  
    print("Fruit:", fruit)
```

```
Fruit: apple  
Fruit: banana  
Fruit: cherry
```

Example with `range()`: Looping over a range of numbers.

```
for i in range(3):  
    print("Iteration:", i)
```

```
Iteration: 0  
Iteration: 1  
Iteration: 2
```

Example with `enumerate()`: Get index and value.

```
colors = ["red", "green", "blue"]  
for index, color in enumerate(colors):  
    print(f"{index}: {color}")
```

```
0: red
1: green
2: blue
```

Python Functions

Functions are reusable blocks of code that perform a specific task.

Why use functions?

- To avoid repeating code.
- To make your code more readable and organized.
- To break problems into smaller, manageable parts.

How to define a function:

```
def function_name(parameters):
    # code block
    return result
```

Basic Example:

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

print(greet("Alice"))
```

Hello, Alice!

Function with multiple parameters:

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

result = add(3, 5)
print("Sum:", result)
```

Sum: 8

Function with default arguments:

```
def power(base, exponent=2):  
    return base ** exponent  
  
print("Default exponent:", power(4))  
print("Custom exponent:", power(2, 3))
```

Default exponent: 16
Custom exponent: 8

Function without return: These perform an action but don't return a value.

```
def display_message():  
    print("This is a message.")  
  
display_message()
```

This is a message.

Python Lambda Functions

Lambda functions are small, anonymous functions you define in a single line.

Why use Lambda functions?

- Quick, throwaway functions that don't need names.
- Often used with `map()`, `filter()`, or `sorted()`.
- Useful when you only need the function once.

Syntax:

```
lambda arguments: expression
```

```
<function __main__.<lambda>(arguments)>
```

Example: Basic Lambda:

```
square = lambda x: x * x
print(square(5))
```

25

Example: Lambda with multiple arguments:

```
add = lambda a, b: a + b
print(add(3, 4))
```

7

Example: Using Lambda in `sorted()`:

```
words = ["banana", "cherry", "apple"]
words_sorted = sorted(words, key=lambda word: len(word))
print(words_sorted)
```

['apple', 'banana', 'cherry']

Difference between `def` and `lambda`:

Feature	<code>def</code> function	<code>lambda</code> function
Name	Named	Anonymous
Multiple statements	Allowed	Not allowed
Readability	Clear for complex logic	Best for simple expressions

Python Arrays

Arrays are containers for elements of the *same data type*, stored more efficiently in memory compared to lists.

Why use Arrays instead of Lists?

- Lists can hold mixed types (e.g., `[1, "two", 3.0]`), arrays cannot.
- Arrays are faster and consume less memory when you only store one type.
- Useful for numerical computations when you don't need full NumPy.

Syntax:

```
array.array(typecode, initializer)
```

Common Type Codes:

Type Code	C Type	Python Type	Size (bytes)
'i'	signed int	int	usually 4
'f'	float	float	usually 4
'd'	double float	float	usually 8
'b'	signed char	int	1

Example: Creating an array of integers

```
import array

nums = array.array("i", [1, 2, 3, 4])
nums.append(5)
print(nums)
```

```
array('i', [1, 2, 3, 4, 5])
```

Accessing elements:

```
print("First element:", nums[0])  
print("Slice:", nums[1:4])
```

```
First element: 1  
Slice: array('i', [2, 3, 4])
```

When to use Lists vs Arrays?

- Use **list** when:
 - You need mixed types.
 - You don't care about memory footprint.
- Use **array.array** when:
 - All elements are numeric and of the same type.
 - You want better performance and less memory usage.
 - You don't want the dependency on NumPy.

Python Imports

What is `import`?

The `import` statement allows you to **bring in external code (modules) into your program**, so you can reuse functions, classes, and variables without writing them yourself.

Why is `import` powerful?

- Gives you access to thousands of **standard libraries** (like `math`, `datetime`, `random`).
- Makes your code **modular and organized** by splitting functionality into multiple files.
- Allows you to use **third-party packages** (like `numpy`, `pandas`) that dramatically extend Python's capabilities.
- Promotes **code reuse** and saves development time.

Types of Modules:

- **Standard Library Modules:** Built into Python (e.g., `os`, `sys`, `json`).
- **Third-Party Modules:** Installed via `pip` (e.g., `requests`, `flask`).
- **User-Defined Modules:** Your own `.py` files with reusable code.

Basic Syntax:

```
import module_name
```

Import specific parts:

```
from module_name import something
```

Import with alias:

```
import module_name as alias
```

Examples:

```
import math
print(math.sqrt(16))

from datetime import datetime
print(datetime.now())

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

```
4.0
2025-09-29 09:03:55.967607
10
```

Tips:

- Always install third-party modules with `pip install`.
- Use `help(module_name)` or `dir(module_name)` to explore a module.
- Keep imports **at the top of your file** for clarity.

In short: `import` is one of Python's greatest strengths—it lets you build on a massive ecosystem instead of reinventing the wheel.

Python Dates and Times

Python handles **dates and times** with the `datetime` module.

Why work with dates?

- Store timestamps (e.g., when something happened).
- Calculate differences between dates or times.
- Format dates for display in a readable way.
- Schedule tasks or handle logs.

Importing datetime:

```
from datetime import datetime, date, time, timedelta
```

Getting current date and time

```
now = datetime.now()
print("Now:", now)
print("Date only:", now.date())
print("Time only:", now.time())
```

Now: 2025-09-29 09:03:55.975036

Date only: 2025-09-29

Time only: 09:03:55.975036

Formatting dates with `strftime()`

Common format codes:

- `%Y` → Year (2025)
- `%m` → Month (01-12)
- `%d` → Day (01-31)

- `%H` → Hour (00-23)
- `%M` → Minute (00-59)
- `%S` → Second (00-59)
- `%A` → Weekday name
- `%B` → Month name

```
print(now.strftime("%Y-%m-%d %H:%M:%S"))
print(now.strftime("%A, %B %d, %Y"))
```

2025-09-29 09:03:55
Monday, September 29, 2025

Creating specific dates and times

```
d = date(2025, 1, 15)
t = time(14, 30, 0)
dt = datetime(2025, 1, 15, 14, 30)

print("Date:", d)
print("Time:", t)
print("Datetime:", dt)
```

Date: 2025-01-15
Time: 14:30:00
Datetime: 2025-01-15 14:30:00

Calculating differences with `timedelta`

```
start = date(2025, 1, 1)
end = date(2025, 12, 31)
delta = end - start
print("Days between:", delta.days)

# Adding and subtracting time
future = now + timedelta(days=7)
past = now - timedelta(hours=5)
print("One week from now:", future)
print("Five hours ago:", past)
```

Days between: 364

One week from now: 2025-10-06 09:03:55.975036

Five hours ago: 2025-09-29 04:03:55.975036

Parsing strings into dates with `strptime()`

- Opposite of `strftime()`: converts a string into a `datetime` object.

```
date_str = "2025-09-28 14:45:00"
parsed = datetime.strptime(date_str, "%Y-%m-%d %H:%M:%S")
print("Parsed datetime:", parsed)
print("Year:", parsed.year, "Month:", parsed.month, "Day:", parsed.day)
```

Parsed datetime: 2025-09-28 14:45:00

Year: 2025 Month: 9 Day: 28

Python Math

Python includes the `math` module for advanced mathematical operations.

Why use `math`?

- For precise calculations.
- For trigonometry, logarithms, powers, etc.

Importing math module:

```
import math
```

Common functions:

Function	Description
<code>math.sqrt(x)</code>	Square root of x
<code>math.pow(x, y)</code>	x raised to the power y
<code>math.floor(x)</code>	Floor (round down)
<code>math.ceil(x)</code>	Ceiling (round up)
<code>math.pi</code>	π constant
<code>math.sin(x)</code>	Sine of x (radians)

Example usage:

```
print("Square root:", math.sqrt(16))
print("Pi:", math.pi)
print("Power:", math.pow(2, 3))
```

```
Square root: 4.0
Pi: 3.141592653589793
Power: 8.0
```

Python String Formatting

String formatting lets you create strings dynamically.

Why format strings?

- To include variables in text.
- To control how values are displayed.

Old style (`%` operator):

```
name = "Alice"
print("Hello, %s" % name)
```

Hello, Alice

`str.format()` method:

```
age = 30
print("Age: {}".format(age))
```

Age: 30

f-strings (Python 3.6+):

```
score = 95.5
print(f"Score: {score}")
```

Score: 95.5

Formatting numbers:

```
value = 1234.56789
print(f"Value rounded: {value:.2f}")
```

Value rounded: 1234.57

Python User Input

User input allows your program to interact with people.

Reading input:

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

Converting input:

- All input is a string by default.





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

Tips:

- Use `int()` or `float()` to convert.
- Always validate input in real applications.

Error handling in Python is done using `try...except` blocks.

Why use `try...except`?

-  To handle exceptions **gracefully** without crashing the program.
-  To provide fallback behavior when something goes wrong.
-  To give clear error messages to users.

Syntax

```
try:
    # Code that might fail
except ExceptionType:
    # Code to handle the specific error
```

Example with Specific Errors

```
try:
    num = int("abc")
    print(100 / num)
except ZeroDivisionError:
    print(" Cannot divide by zero!")
except ValueError:
    print(" Invalid number format!")
```

- If input is not a number → `ValueError` is caught.
- If input is `0` → `ZeroDivisionError` is caught.

Catching Multiple Exceptions in One Block

```
try:
    num = int("abc")
    print(10 / num)
except (ValueError, ZeroDivisionError) as e:
    print(" Error:", e)
```

- Combines multiple related errors into a single block.

The `else` Clause

- Runs **only if no exception occurs**.

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print(" Cannot divide by zero!")
else:
    print(" Success! Result =", result)
```

The `finally` Clause

- Runs **always**, whether there was an error or not.

```
try:
    file = open("data.txt")
    print(file.read())
except FileNotFoundError:
    print(" File not found!")
finally:
    print(" Closing resources (if open).")
```

Catching All Errors (Not Recommended)

```
try:
    risky_code()
except Exception as e:
    print("⚠️ Something went wrong:", e)
```

- Advantage: prevents crashes if you don't know what errors may occur.
- Disadvantage: hides bugs and makes debugging harder.
- Best practice: **catch specific exceptions** whenever possible.

Bare `except:` (Very Dangerous)

```
try:
    risky_code()
except:
    print("⚠️ An unknown error occurred.")
```

- Catches **everything**, even `KeyboardInterrupt` (when user presses Ctrl+C).
- Should be avoided unless in very controlled scenarios.

Best Practices

- 🎯 Catch **specific exceptions** instead of using `except:`.
- 📖 Read the traceback first before writing error handling.
- 🛡️ Use `finally` for cleanup (e.g., closing files, releasing resources).
- 🚫 Don't silence errors unless absolutely necessary.

Summary: `try...except` makes your code more robust. Use **specific exceptions** for clarity, `else` for success-only code, and `finally` for guaranteed cleanup. Avoid broad `except:` unless you have a very good reason.

Errors are not failures—they are signals that something needs your attention.

When Python encounters a problem, it produces a **traceback**, which tells you:

- 📍 Where the error occurred (file, function, line number)
- 🏷️ What type of error it was (e.g., `ValueError`)
- 💬 A message describing why it failed

Common Error Types

1. `SyntaxError`

- Happens when code is not valid Python.
- Example:

```
if True print('Hi')    # Missing colon
```

- Python cannot even start running this code.

2. `IndentationError`

- Happens when indentation (spaces/tabs) is wrong.
- Example:

```
def hello():  
print('Hi')    # Not indented properly
```

- Python enforces indentation strictly.

3. `NameError`

- Using a variable that doesn't exist.
- Example:

```
print(age)    # age not defined yet
```

4. `TypeError`

- Wrong data type for an operation.
- Example:

```
'5' + 3    # Cannot add string and int
```

5. `ValueError`

- Correct type but invalid value.
- Example:

```
int('abc')    # Cannot convert letters to int
```

6. `ZeroDivisionError`

- Division by zero.
- Example:

```
10 / 0
```

7. `IndexError`

- Accessing list index out of range.
- Example:

```
nums = [1, 2]  
print(nums[5])
```

8. `KeyError`

- Using a dictionary key that doesn't exist.

- Example:

```
person = {'name': 'Ali'}  
print(person['age'])
```

9. `AttributeError`

- Accessing an attribute/method that does not exist.
- Example:

```
text = 'hello'  
text.reverse() # str has no method 'reverse'
```

10. `ImportError`

- Python cannot import the requested module.
- Example:

```
import not_a_module
```

Handling Errors Gracefully

Use `try...except` blocks:

```
def safe_divide(x, y):  
    try:  
        return x / y  
    except ZeroDivisionError:  
        return "Cannot divide by zero!"  
  
print(safe_divide(10, 2)) # Works fine  
print(safe_divide(5, 0)) # Handled gracefully
```

Errors vs Warnings

● Error:

- Stops program execution immediately.
- Must be fixed or handled.
- Example: `ZeroDivisionError`.

● Warning:

- Program still runs, but Python is telling you something might be wrong.
- Example:

```
import warnings
warnings.warn("This feature is deprecated.")
```

- Output shows a warning, but code continues.

Summary: Errors crash your code unless handled. Warnings do not crash the program but alert you to potential issues.