The Little Book of Python

Version 0.1.0

Duc-Tam Nguyen

2025-09-15

Table of contents

The Little Book of Python

Chapter 1. Basics of Python

1. What is Python?

Python is a high-level, general-purpose programming language that emphasizes simplicity and readability. Created by Guido van Rossum in the early 1990s, Python was designed to make programming more accessible by using a clean, English-like syntax. Unlike lower-level languages such as C or assembly, Python abstracts away many technical details, allowing developers to focus on solving problems rather than managing memory or dealing with system-level operations.

Python is both interpreted and dynamically typed. Being interpreted means that Python executes code line by line without requiring compilation into machine code beforehand. This makes it very beginner-friendly, as you can test and run your code quickly without extra steps. Being dynamically typed means you do not need to declare variable types explicitly—Python determines them at runtime, which speeds up development.

The language is also cross-platform. A Python program written on macOS can usually run on Linux or Windows with little to no changes, as long as the environment has Python installed. Combined with its vast ecosystem of libraries and frameworks, Python has become one of the most popular languages worldwide, used in areas ranging from web development to artificial intelligence.

Python's design philosophy emphasizes readability. For example, instead of curly braces (`{}`) to mark blocks of code, Python uses indentation (spaces or tabs). This enforces clean code structure and makes programs easier to read and maintain.

Deep Dive

- Versatility: Python is sometimes called a "glue language" because it can integrate with other systems and languages easily. You can call C or C++ libraries, run shell commands, or embed Python into other applications.

- Community and ecosystem: With millions of developers worldwide, Python has a massive community. This means a wealth of tutorials, open-source projects, and support forums are available for learners and professionals.
- Libraries and frameworks: Python has specialized libraries for nearly every domain:
  - Data Science & AI: NumPy, Pandas, TensorFlow, PyTorch.
  - Web Development: Django, Flask, FastAPI.
  - Automation & Scripting: Selenium, BeautifulSoup, `os` and `shutil` modules.
  - Systems Programming: `subprocess`, `asyncio`, threading tools.
- Design Philosophy: The "Zen of Python" (accessible by running `import this` in a Python shell) summarizes guiding principles, such as "Simple is better than complex" and "Readability counts."

Python's balance of simplicity and power makes it an excellent first language for beginners, yet powerful enough for advanced engineers building production-grade systems.

Tiny Code

```
# A simple Python program
print("Hello, World!")

# Variables don't require type declarations
x = 10       # integer
y = 3.14     # float
name = "Ada" # string

# Control flow example
if x > 5:
    print(f"{name}, x is greater than 5!")
```

Why it Matters

Python matters because it lowers the barrier to entry into programming. Its readability and straightforward syntax make it an ideal starting point for newcomers, while its depth and ecosystem allow professionals to tackle complex problems in machine learning, finance, cybersecurity, and more. Learning Python often serves as a gateway to the broader world of computer science and software engineering.

Try It Yourself

1. Open a terminal or Python shell and type `print("Hello, Python!")`.
2. Assign a number to a variable and print it. Example:

```
age = 25
print("I am", age, "years old.")
```

3. Run `import this` in the Python shell and read through the Zen of Python. Which line resonates with you most, and why?

This exercise introduces you to Python's core design philosophy while letting you experience the simplicity of writing and running your first code.

2. Installing Python & Running Scripts

Python is available for almost every operating system, and installing it is the first step before you can write and execute your own programs. Most modern computers already come with Python preinstalled, but often it is not the latest version. For development, it is generally recommended to use the most recent stable release (for example, Python 3.12).

Deep Dive

Download and Install:

- On Windows, download the installer from the official website python.org. During installation, make sure to check the box *"Add Python to PATH"* so you can run Python from the command line.
- On macOS, you can use Homebrew (`brew install python`) or download from python.org.
- On Linux, Python is usually preinstalled. If not, use your package manager (`sudo apt install python3` on Ubuntu/Debian, `sudo dnf install python3` on Fedora).

After installation, open your terminal (or command prompt) and type:

```
python3 --version
```

This should display something like `Python 3.14.0`. If it doesn't, the installation or PATH configuration may need adjustment.

Running the Interpreter (REPL):

You can enter interactive mode by typing `python` or `python3` in your terminal. This launches the Read-Eval-Print Loop (REPL), where you can execute code line by line:

```
>>> 2 + 3
5
>>> print("Hello, Python!")
Hello, Python!
```

Running Scripts:

While the REPL is good for quick experiments, most real programs are saved in files with a `.py` extension. You can create a file `hello.py` containing:

```
print("Hello from a script!")
```

Then run it from your terminal:

```
python3 hello.py
```

IDEs and Editors:

Beginners often start with editors like IDLE (which comes with Python) or more advanced ones like VS Code or PyCharm, which provide syntax highlighting, debugging tools, and project management.

Environment Management:

Installing libraries for one project can affect others. To avoid conflicts, Python provides virtual environments (`venv`). This isolates project dependencies:

```
python3 -m venv myenv
source myenv/bin/activate   # On Linux/macOS
myenv\Scripts\activate      # On Windows
```

Tiny Code

```
# File: hello.py
name = "Ada"
print("Hello,", name)
```

To run:

```
python3 hello.py
```

Why it Matters

Understanding how to install Python and run scripts is fundamental because it gives you control over your development environment. Without mastering this, you can't progress to building real applications. Installing properly also ensures you have access to the latest features and security updates.

Try It Yourself

1. Install the latest version of Python on your computer.
2. Verify your installation with `python3 --version`.
3. Open the REPL and try basic arithmetic (`5 * 7`, `10 / 2`).
4. Write a script called `greeting.py` that prints your name and favorite color.
5. Run the script from your terminal.

This exercise ensures you can not only experiment interactively but also save and execute complete programs.

3. Python Syntax & Indentation

Python's syntax is designed to be simple and human-readable. Unlike many other programming languages that use braces `{}` or keywords to define code blocks, Python uses indentation (spaces or tabs). This is not optional—correct indentation is part of Python's grammar. The focus on clean and consistent code is one of the reasons why Python is popular both in education and professional development.

Deep Dive

- Indentation Instead of Braces: In languages like C, C++, or Java, you often see:

  ```
  if (x > 0) {
      printf("Positive\n");
  }
  ```

  In Python, the same block is defined by indentation:

  ```
  if x > 0:
      print("Positive")
  ```

The colon (`:`) signals the start of a new block, and the indented lines that follow belong to that block.

- Consistency Matters: Python requires consistency in indentation. You cannot mix tabs and spaces within the same block. The most common convention is 4 spaces per indentation level.
- Nested Indentation: Blocks can be nested by increasing indentation further:

```
if x > 0:
    if x % 2 == 0:
        print("Positive and even")
    else:
        print("Positive and odd")
```

- Syntax Simplicity: Python syntax avoids clutter. For example:
  - No need for semicolons (`;`) at the end of lines (though allowed).
  - Parentheses are optional in control statements unless needed for clarity.
  - Whitespace and line breaks matter, which encourages writing readable code.
- Line Continuation: Long lines can be split with `\` or by wrapping expressions inside parentheses:

```
total = (100 + 200 + 300 +
         400 + 500)
```

- Comments: Python uses `#` for single-line comments and triple quotes (`""" ... """`) for docstrings or multi-line comments.

Tiny Code

```
# Proper indentation example
score = 85

if score >= 60:
    print("Pass")
    if score >= 90:
        print("Excellent")
    else:
        print("Good job")
else:
    print("Fail")
```

Why it Matters

Indentation rules enforce consistency across all Python code. This reduces errors caused by messy formatting and makes programs easier to read, especially when working in teams. Python's syntax philosophy ensures beginners learn clean habits from the start and professionals maintain readability in large projects.

Try It Yourself

1. Write a program that checks if a number is positive, negative, or zero using proper indentation.
2. Experiment by removing indentation or mixing spaces and tabs—notice how Python raises an `IndentationError`.
3. Write nested `if` statements to check whether a number is divisible by both 2 and 3.

This will help you experience firsthand why Python enforces indentation and how it guides you to write clean, structured code.

4. Variables & Assignment

In Python, a variable is like a box with a name where you can store information. You can put numbers, text, or other kinds of data inside that box, and later use the name of the box to get the value back.

Unlike some languages, you don't need to say what kind of data will go inside the box—Python figures it out for you automatically.

Deep Dive

- Creating a Variable: You just choose a name and use the equals sign `=` to assign a value:

```
age = 20
name = "Alice"
height = 1.75
```

- Reassigning a Variable: You can change the value at any time:

```
age = 21   # overwrites the old value
```

- Naming Rules:
  - Names can include letters, numbers, and underscores (_).
  - They cannot start with a number.

  - They are case-sensitive: `Age` and `age` are different.
  - Use meaningful names, like `temperature`, instead of `t`.

- Dynamic Typing: Python does not require you to declare the type. The same variable can hold different types of data at different times:

```
x = 10      # integer
x = "hello" # now it's a string
```

- Multiple Assignments: You can assign several variables in one line:

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

- Swapping Values: Python makes it easy to swap values without a temporary variable:

```
a, b = b, a
```

Tiny Code

```
# Assign variables
name = "Ada"
age = 25

# Print them
print("My name is", name)
print("I am", age, "years old")
```

Why it Matters

Variables let you store and reuse information in your programs. Without variables, you would have to repeat values everywhere, making your code harder to read and change. They are the foundation of all programming.

Try It Yourself

1. Create a variable called `color` and assign your favorite color as text.
2. Make a variable `number` and assign it any number you like.
3. Print both values in a sentence, like:

   ```
   My favorite color is blue and my number is 7
   ```

4. Try changing the values and run the program again.

This will show you how variables make your code flexible and easy to update.

4. Variables & Assignment

In Python, a variable is like a box with a name where you can store information. You can put numbers, text, or other kinds of data inside that box, and later use the name of the box to get the value back.

Unlike some languages, you don't need to say what kind of data will go inside the box—Python figures it out for you automatically.

Deep Dive

To create a variable, you simply choose a name and use the equals sign `=` to assign a value. For example:

```
age = 20
name = "Alice"
height = 1.75
```

You can also change the value at any time. For instance:

```
age = 21   # overwrites the old value
```

Variable names have a few rules. They can include letters, numbers, and underscores (`_`), but they cannot start with a number. They are also case-sensitive, so `Age` and `age` are considered different. It's a good habit to use meaningful names, like `temperature` instead of just `t`.

Python uses dynamic typing, which means you don't have to declare the type of data in advance. A single variable can hold different types of data at different times:

```
x = 10      # integer
x = "hello" # now it's a string
```

You can even assign several variables in one line, like this:

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

And if you ever need to swap the values of two variables, Python makes it very easy without needing a temporary helper:

```
a, b = b, a
```

Tiny Code

```
# Assign variables
name = "Ada"
age = 25

# Print them
print("My name is", name)
print("I am", age, "years old")
```

Why it Matters

Variables let you store and reuse information in your programs. Without variables, you would have to repeat values everywhere, making your code harder to read and change. They are the foundation of all programming.

Try It Yourself

1. Create a variable called `color` and assign your favorite color as text.
2. Make a variable `number` and assign it any number you like.
3. Print both values in a sentence, like:

   ```
   My favorite color is blue and my number is 7
   ```

4. Try changing the values and run the program again.

This will show you how variables make your code flexible and easy to update.

5. Data Types Overview

Every piece of information in Python has a data type. A data type tells Python what kind of thing the value is—whether it's a number, text, a list of items, or something else. Understanding data types is important because it helps you know what you can and cannot do with a value.

Deep Dive

Python has several basic data types you'll use all the time.

Numbers are used for math. Python has three main kinds of numbers: integers (`int`) for whole numbers, floating-point numbers (`float`) for decimals, and complex numbers (`complex`) which are used less often, mostly in math and engineering.

Strings (`str`) represent text. Anything inside quotes, either single (`'hello'`) or double (`"hello"`), is treated as a string. Strings can hold words, sentences, or even whole paragraphs.

Booleans (`bool`) represent truth values—either `True` or `False`. These are useful for decision making in programs, like checking if a condition is met.

Collections let you store multiple values in a single variable. Lists (`list`) are ordered, changeable collections of items, like `[1, 2, 3]`. Tuples (`tuple`) are like lists but cannot be changed after creation, such as `(1, 2, 3)`. Sets (`set`) are collections of unique, unordered items. Dictionaries (`dict`) store data as key–value pairs, like `{"name": "Alice", "age": 25}`.

There are also special types like `NoneType`, which only has the value `None`. This represents "nothing" or "no value."

Python figures out the type of a variable automatically. If you want to check a variable's type, you can use the built-in `type()` function:

```
x = 42
print(type(x))  # <class 'int'>
```

Tiny Code

```
# Examples of different data types
number = 10          # int
pi = 3.14            # float
name = "Ada"         # str
is_student = True    # bool
items = [1, 2, 3]    # list
point = (2, 3)       # tuple
unique = {1, 2, 3}   # set
person = {"name": "Ada", "age": 25}  # dict
nothing = None       # NoneType

print(type(name))    # check type
```

Why it Matters

Data types are the foundation of programming logic. Knowing the type of data tells you what operations you can perform. For example, you can add two numbers but not a number and a string without converting one of them. This prevents errors and helps you design programs correctly.

Try It Yourself

1. Create a variable `city` with the name of your city.
2. Make a list called `colors` with three of your favorite colors.
3. Create a dictionary `book` with keys `title` and `author`.
4. Print out the type of each variable using `type()`.
5. Try combining different types (like adding a string and a number) and see what error appears.

This will give you a feel for how Python handles different data and why types matter.

6. Numbers (int, float, complex)

Numbers are one of the most basic building blocks in Python. They allow you to do math, represent quantities, and calculate results in your programs. Python has three main types of numbers: integers (`int`), floating-point numbers (`float`), and complex numbers (`complex`).

Deep Dive

Number Types in Python

Type	Example	Description
`int`	`-3`, `0`, `42`	Whole numbers, no decimal part. Can be very large (only limited by memory).
`float`	`3.14`, `-0.5`	Numbers with decimal points, often used for measurements or precision math.
`complex`	`2 + 3j`	Numbers with real and imaginary parts, useful in math, physics, engineering.

Common Arithmetic Operators

Operator	Example	Result	Meaning
`+`	`5 + 2`	`7`	Addition
`-`	`5 - 2`	`3`	Subtraction
`*`	`5 * 2`	`10`	Multiplication
`/`	`5 / 2`	`2.5`	Division (always float)
`//`	`5 // 2`	`2`	Floor division (whole number part only)
`%`	`5 % 2`	`1`	Modulo (remainder)
‘|2 3|8‘	Exponent (raise to a power)		

Type Conversion

Function	Example	Result
`int()`	`int(3.9)`	`3`
`float()`	`float(7)`	`7.0`
`complex()`	`complex(2, 3)`	`2+3j`

You can check the type of any number with the `type()` function:

```
x = 42
print(type(x))  # <class 'int'>
```

Tiny Code

```
# Integers
a = 10
b = -3

# Floats
pi = 3.14
g = 9.81

# Complex
z = 2 + 3j

# Operations
print(a + b)    # 7
```

```
print(a / 2)    # 5.0
print(a // 2)   # 5
print(a % 3)    # 1
print(2  3)   # 8

# Type checking
print(type(pi)) # <class 'float'>
print(type(z))  # <class 'complex'>
```

Why it Matters

Numbers are essential for everything from simple calculations to complex algorithms. Understanding the different numeric types and how they behave allows you to choose the right one for each situation. Use integers for counting, floats for precise measurements, and complex numbers for specialized scientific work.

Try It Yourself

1. Create two integers and try all the arithmetic operators (`+`, `-`, `*`, `/`, `//`, `%`, “).
2. Make a float variable for your height (like `1.75`) and multiply it by 2.
3. Experiment with `int()`, `float()`, and `complex()` to convert between number types.
4. Write a complex number and print both its real and imaginary parts using `.real` and `.imag`.

This will help you see how Python handles different numeric types in practice.

7. Strings (creation & basics)

A string in Python is a sequence of characters—letters, numbers, symbols, or even spaces—enclosed in quotes. Strings are used whenever you want to work with text, such as names, sentences, or file paths.

Deep Dive

Creating Strings

You can create strings using either single quotes or double quotes:

```
name = 'Alice'
greeting = "Hello, world!"
```

For multi-line text, you can use triple quotes:

```
paragraph = """This is a
multi-line string."""
```

Basic String Operations

Operation	Example	Result
Concatenation	`"Hello" + " " + "Bob"`	`"Hello Bob"`
Repetition	`"ha" * 3`	`"hahaha"`
Indexing	`"Python"[0]`	`'P'`
Negative Indexing	`"Python"[-1]`	`'n'`
Slicing	`"Python"[0:4]`	`"Pyth"`
Length	`len("Python")`	`6`

Escape Characters

Sometimes you need special characters inside a string:

Escape Code	Meaning	Example	Result
`\n`	New line	`"Hello\nWorld"`	`HelloWorld`
`\t`	Tab	`"A\tB"`	`A  B`
`\'`	Single quote	`'It\'s fine'`	`It's fine`
`\"`	Double quote	`"He said \"Hi\""`	`He said "Hi"`
`\\`	Backslash	`"C:\\Users\\Alice"`	`C:\Users\Alice`

Tiny Code

```
# Creating strings
word = "Python"
sentence = 'I love coding'
multiline = """This is
a string that spans
multiple lines."""

# Operations
print(word[0])        # 'P'
print(word[-1])       # 'n'
print(word[0:3])      # 'Pyt'
```

```
print(word + " 3.12") # 'Python 3.12'
print("ha" * 4)       # 'hahaha'

# Escape characters
path = "C:\\Users\\Alice"
print(path)
```

Why it Matters

Strings are everywhere—whether you're printing messages, reading files, sending data across the internet, or handling user input. Mastering how to create and manipulate strings is essential for building real-world Python programs.

Try It Yourself

1. Create a string with your full name and print the first letter and the last letter.
2. Write a sentence and use slicing to print only the first 5 characters.
3. Use string concatenation to join `"Hello"` and your name with a space in between.
4. Make a string with an escape sequence, like `"Line1\nLine2"`, and print it.

This practice will help you understand how Python treats text as data you can store, manipulate, and display.

8. Booleans and Truth Values

Booleans are the simplest type of data in Python. They represent only two values: `True` or `False`. Booleans are often the result of comparisons or conditions in a program, and they control the flow of logic, such as deciding which branch of an `if` statement should run.

Deep Dive

Boolean Values

In Python, the boolean type is `bool`. There are only two possible values:

```
is_sunny = True
is_raining = False
```

Notice that `True` and `False` are capitalized—writing `true` or `false` will cause an error.

Comparisons That Produce Booleans

Expression	Example	Result
Equal	`5 == 5`	`True`
Not equal	`5 != 3`	`True`
Greater than	`7 > 10`	`False`
Less than	`2 < 5`	`True`
Greater/Equal	`3 >= 3`	`True`
Less/Equal	`4 <= 2`	`False`

Boolean Logic

Python also supports logical operators that combine boolean values:

Operator	Example	Result
`and`	`True and False`	`False`
`or`	`True or False`	`True`
`not`	`not True`	`False`

Truthiness in Python

Not just `True` and `False` are considered booleans. Many values in Python have an implicit boolean value:

Value Type	Considered as
`0`, `0.0`, `0j`	`False`
Empty string `""`	`False`
Empty list `[]`	`False`
Empty dict `{}`	`False`
`None`	`False`
Everything else	`True`

You can test this with the `bool()` function:

```
print(bool(0))     # False
print(bool("hi"))  # True
```

Tiny Code

```
x = 10
y = 20

print(x < y)          # True
print(x == y)         # False
print((x < y) and (y > 5))  # True
print(not (x > y))    # True

# Truthiness
print(bool(""))       # False
print(bool("Python")) # True
```

Why it Matters

Booleans are the foundation of decision-making in programming. They let you write programs that can react differently depending on conditions—like checking if a user is logged in, if there is enough money in a bank account, or if a file exists. Without booleans, all programs would just run straight through without making choices.

Try It Yourself

1. Assign a boolean variable `is_python_fun = True` and print it.
2. Compare two numbers (like `5 > 3`) and store the result in a variable. Print the variable.
3. Test the truthiness of an empty list `[]` and a non-empty list `[1, 2, 3]` with `bool()`.
4. Write an expression using `and`, `or`, and `not` together.

This practice will help you see how conditions and logic form the backbone of Python programs.

9. Comments in Python

Comments are notes you add to your code that Python ignores when running the program. They're meant for humans, not the computer. Comments explain what your code does, why you wrote it a certain way, or leave reminders for yourself and others.

Deep Dive

Single-Line Comments In Python, the # symbol marks the start of a comment. Everything after it on the same line is ignored by Python:

```
# This is a single-line comment
x = 10  # You can also put a comment after code
```

Multi-Line Comments (Docstrings) Python doesn't have a special syntax just for multi-line comments, but programmers often use triple quotes (""" or '''). These are usually used for docstrings (documentation strings), but they can serve as block comments if not assigned to a variable:

```
"""
This is a multi-line comment.
You can use triple quotes
to write long explanations.
"""
```

Docstrings for Functions and Classes Triple quotes are more commonly used as docstrings to document functions, classes, or modules. They are placed right after the definition line:

```
def greet(name):
    """
    This function takes a name
    and prints a greeting.
    """
    print("Hello,", name)
```

You can read docstrings later using the `help()` function.

Why Comments Are Useful

Purpose	Example
Explain code logic	`# Loop through items in the list`
Clarify tricky parts	`# Using floor division to ignore decimals`
Leave reminders (TODOs, FIXMEs)	`# TODO: handle negative numbers`
Provide documentation	Docstrings that explain functions, classes, or entire files

Good comments don't just repeat the code; they explain the why, not just the what.

Tiny Code

```
# Store a user's age
age = 25

# Check if age is greater than 18
if age > 18:
    print("Adult")

def square(x):
    """Return the square of a number."""
    return x * x

print(square(4))  # prints 16
```

Why it Matters

Comments make your code easier to understand for both yourself and others. Six months from now, you might forget why you wrote something. Clear comments act like a guidebook. In teams, comments and docstrings are essential for collaboration, as they make the codebase easier to maintain.

Try It Yourself

1. Write a small program that calculates the area of a rectangle. Add comments explaining what each step does.
2. Use a triple-quoted docstring to describe what the whole program does at the top of your file.
3. Add a `TODO` comment to remind yourself to improve the program later (for example, adding user input).

This will show you how comments make programs not just for computers, but for people too.

10. Printing Output (`print` function)

The `print()` function is one of the most commonly used tools in Python. It lets you display information on the screen so you can see the result of your program, check values, or interact with users.

Deep Dive

Basic Printing The simplest use of `print()` is to show text:

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

Printing Variables You can print variables directly by passing them to `print()`:

```
name = "Ada"
age = 25
print(name)
print(age)
```

Printing Multiple Values `print()` can take multiple arguments separated by commas. Python will add spaces between them automatically:

```
print("Name:", name, "Age:", age)
```

String Formatting There are several ways to make your output more readable:

Method	Example	Output
f-strings (modern)	`print(f"{name} is {age} years old")`	`Ada is 25 years old`
`.format()` method	`print("{} is {}".format(name, age))`	`Ada is 25`
Old `%` style	`print("%s is %d" % (name, age))`	`Ada is 25`

End and Separator Options By default, `print()` ends with a new line (`\n`). You can change this using the `end` parameter:

```
print("Hello", end=" ")
print("World")
# Output: Hello World
```

You can also change the separator between multiple items using `sep`:

```
print("apple", "banana", "cherry", sep=", ")
# Output: apple, banana, cherry
```

Printing Special Characters You can print new lines or tabs with escape sequences:

```
print("Line1\nLine2")
print("A\tB")
```

Tiny Code

```
name = "Grace"
language = "Python"
year = 1991

print("Hello, world!")
print("My name is", name)
print(f"{name} created {language} in {year}?")
print("apple", "orange", "grape", sep=" | ")
```

Why it Matters

Printing is the most direct way to see what your program is doing. It helps you understand results, debug mistakes, and communicate with users. Even professional developers rely heavily on `print()` when testing and exploring code quickly.

Try It Yourself

1. Print your name and your favorite hobby in one sentence.
2. Create two numbers and print their sum with a clear message.
3. Use `sep` to print three words separated by dashes (`-`).
4. Use `end` to print two words on the same line without spaces.

This will show you how flexible `print()` is for displaying information in Python.

Chapter 2. Control Flow

11. Comparison Operators

Comparison operators let you compare two values and return a boolean result (`True` or `False`). They are the foundation for making decisions in Python programs—without them, you couldn't check conditions like "Is this number bigger than that number?" or "Are these two things equal?"

Deep Dive

Comparison operators work on numbers, strings, and many other types. They allow you to check equality, inequality, and order.

Basic Comparison Operators

Operator	Example	Meaning	Result
`==`	`5 == 5`	Equal to	`True`
`!=`	`5 != 3`	Not equal to	`True`
`>`	`7 > 3`	Greater than	`True`
`<`	`2 < 5`	Less than	`True`
`>=`	`3 >= 3`	Greater than or equal to	`True`
`<=`	`4 <= 2`	Less than or equal to	`False`

Comparisons always return `True` or `False`, which can be stored in variables or used directly inside control flow statements (`if`, `while`).

Chained Comparisons Python allows chaining comparisons for readability:

```
x = 5
print(1 < x < 10)  # True
print(10 < x < 20) # False
```

This is equivalent to writing `(1 < x) and (x < 10)`.

Comparisons with Strings Strings are compared alphabetically (lexicographically), based on Unicode values:

```
print("apple" == "apple")  # True
print("apple" < "banana")  # True
print("Zebra" < "apple")   # True (uppercase letters come first)
```

Tiny Code

```
x = 10
y = 20

print(x == y)   # False
print(x != y)   # True
```

```
print(x > y)    # False
print(x <= y)   # True

# Chain comparisons
print(5 < x < 15)  # True
```

Why it Matters

Without comparisons, programs couldn't make choices. They are the basis for decisions like checking passwords, validating input, controlling loops, or comparing values in data. Every real-world Python program relies on comparison operators to "decide what to do next."

Try It Yourself

1. Write a program that compares two numbers (`a = 7`, `b = 12`) and prints whether `a` is less than, greater than, or equal to `b`.
2. Create two strings and check if they are equal.
3. Use a chained comparison to check if a number `n = 15` is between 10 and 20.
4. Experiment with `<` and `>` on strings like `"cat"` and `"dog"` to see how Python compares text.

12. Logical Operators

Logical operators combine boolean values (`True` or `False`) to form more complex conditions. They are essential when you want to check multiple things at once, like "Is the number positive and even?" or "Is this user an admin or a guest?"

Deep Dive

Python has three main logical operators:

Operator	Example	Result	Meaning
`and`	`True and False`	`False`	True only if both sides are True
`or`	`True or False`	`True`	True if at least one side is True
`not`	`not True`	`False`	Flips the truth value (True → False)

Truth Tables

`and` operator:

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

`or` operator:

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

`not` operator:

A	not A
True	False
False	True

Combining Conditions Logical operators are often used in `if` statements:

```
age = 20
is_student = True

if age > 18 and is_student:
    print("Eligible for student discount")
```

Short-Circuiting Python stops evaluating as soon as the result is known:

- For `and`, if the first condition is False, Python won't check the second.
- For `or`, if the first condition is True, Python won't check the second.

Tiny Code

```
x = 10
y = 5

print(x > 0 and y > 0)   # True
print(x > 0 or y < 0)    # True
print(not (x == y))      # True

# Short-circuit example
print(False and (10/0))  # False, no error (second part skipped)
print(True or (10/0))    # True, no error (second part skipped)
```

Why it Matters

Logical operators allow your programs to make more complex decisions by combining multiple conditions. They're at the heart of all real-world logic, from validating form inputs to controlling access in applications.

Try It Yourself

1. Write a condition that checks if a number is both positive and less than 100.
2. Check if a variable `name` is either `"Alice"` or `"Bob"`.
3. Use `not` to test if a list is empty (`not my_list`).
4. Experiment with short-circuiting by combining `and` or `or` with expressions that would normally cause an error.

13. `if` Statements

An `if` statement lets your program make decisions. It checks a condition, and if that condition is `True`, it runs a block of code. If the condition is `False`, the block is skipped. This is the most basic form of control flow in Python.

Deep Dive

Basic Structure

```
if condition:
    # code runs only if condition is True
```

The colon (`:`) signals the start of the block, and indentation shows which lines belong to the `if`.

Example

```
x = 10
if x > 5:
    print("x is greater than 5")
```

Since `x > 5` is `True`, the message is printed.

Condition Can Be Any Boolean Expression The expression inside `if` must evaluate to `True` or `False`. This can come from comparisons, logical operators, or truthy/falsy values:

```
if "hello":     # non-empty string is True
    print("This runs")
```

Indentation is Required All code inside the `if` block must be indented the same amount. Without correct indentation, Python will raise an `IndentationError`.

Tiny Code

```
temperature = 30

if temperature > 25:
    print("It's a hot day!")

if temperature < 0:
    print("It's freezing!")
```

Why it Matters

Without `if` statements, programs would always run the same way. Conditions make programs dynamic and responsive—whether it's checking user input, validating data, or making choices in games, `if` is the starting point for logic in Python.

Try It Yourself

1. Write an `if` statement that prints `"Positive"` if a number is greater than zero.
2. Test what happens if the number is zero—does the code run or not?
3. Use an `if` statement to check if a string is empty, and print `"Empty string"` when it is.
4. Change the indentation in your code incorrectly and observe Python's error message.

14. `if...else`

The `if...else` structure lets your program choose between two paths. If the condition is `True`, the `if` block runs. If the condition is `False`, the `else` block runs instead. This ensures that one of the two blocks always executes.

Deep Dive

Basic Structure

```
if condition:
    # code runs if condition is True
else:
    # code runs if condition is False
```

Example

```
age = 16

if age >= 18:
    print("You can vote")
else:
    print("You are too young to vote")
```

Here, if `age` is 18 or more, the first message is printed. Otherwise, the second one runs.

`if...else` with Variables You can use the result of conditions to assign values:

```
x = 10
y = 20

bigger = x if x > y else y
print(bigger)   # 20
```

This is called a ternary expression (or conditional expression).

Only One `else` An `if` statement can have at most one `else`, and it always comes last.

Tiny Code

```
score = 75

if score >= 60:
    print("Pass")
else:
    print("Fail")
```

Why it Matters

The `if...else` structure makes programs capable of handling two outcomes: one when a condition is met, and another when it isn't. It's essential for branching logic—without it, you could only run code when conditions are true, not handle the "otherwise" case.

Try It Yourself

1. Write a program that checks if a number is even or odd using `if...else`.
2. Create a variable `temperature` and print `"Warm"` if it's 20 or above, otherwise `"Cold"`.
3. Use a conditional expression to set `status = "adult"` if `age >= 18`, else `"minor"`.
4. Change the condition to test different inputs and see how the output changes.

15. `if...elif...else`

The `if...elif...else` structure lets you check multiple conditions in order. The program will run the first block where the condition is `True`, and then skip the rest. If none of the conditions are true, the `else` block runs.

Deep Dive

Basic Structure

```
if condition1:
    # runs if condition1 is True
elif condition2:
    # runs if condition1 is False AND condition2 is True
elif condition3:
    # runs if above are False AND condition3 is True
else:
    # runs if none of the above are True
```

Example

```
score = 85

if score >= 90:
    print("Excellent")
elif score >= 75:
    print("Good")
elif score >= 60:
    print("Pass")
else:
    print("Fail")
```

Here, Python checks each condition in order. Since `score >= 75` is true, it prints `"Good"` and skips the rest.

Order Matters Conditions are checked from top to bottom. As soon as one is `True`, Python stops checking further. For example:

```
x = 100
if x > 50:
    print("Bigger than 50")
elif x > 10:
    print("Bigger than 10")
```

Only `"Bigger than 50"` is printed, even though `x > 10` is also true.

Optional Parts

- The `elif` can appear as many times as needed.
- The `else` is optional—you don't need it if you only want to handle some cases.

Tiny Code

```
day = "Wednesday"

if day == "Monday":
    print("Start of the week")
elif day == "Friday":
    print("Almost weekend")
elif day == "Saturday" or day == "Sunday":
    print("Weekend!")
else:
    print("Midweek day")
```

Why it Matters

Most real-life decisions aren't just yes-or-no. The `if...elif...else` chain lets you handle multiple possibilities in an organized way, making your code more flexible and readable.

Try It Yourself

1. Write a program that checks a number and prints `"Positive"`, `"Negative"`, or `"Zero"`.
2. Create a grading system: `90+ = A`, `75-89 = B`, `60-74 = C`, below `60 = F`.
3. Write code that prints which day of the week it is, based on a variable `day`.
4. Experiment by changing the order of conditions and observe how the output changes.

16. Nested Conditions

A nested condition means putting one `if` statement inside another. This allows your program to make more specific decisions by checking an additional condition only when the first one is true.

Deep Dive

Basic Structure

```
if condition1:
    if condition2:
        # runs if both condition1 and condition2 are True
    else:
        # runs if condition1 is True but condition2 is False
else:
    # runs if condition1 is False
```

Example

```
age = 20
is_student = True

if age >= 18:
    if is_student:
        print("Adult student")
    else:
        print("Adult, not a student")
else:
    print("Minor")
```

Here, the second check (`is_student`) only happens if the first check (`age >= 18`) is true.

Why Nesting is Useful Nested conditions let you handle cases that depend on multiple layers of logic. However, too much nesting can make code hard to read. In such cases, logical operators (`and`, `or`) are often better:

```
if age >= 18 and is_student:
    print("Adult student")
```

Best Practice

- Use nesting when the second condition should only be checked if the first one is true.
- For readability, avoid deep nesting—prefer combining conditions with logical operators when possible.

Tiny Code

```python
x = 15

if x > 0:
    if x % 2 == 0:
        print("Positive even number")
    else:
        print("Positive odd number")
else:
    print("Zero or negative number")
```

Why it Matters

Nested conditions add depth to decision-making. They let you structure logic in layers, which is closer to how we reason in real life—for example, "If the shop is open, then check if I have enough money."

Try It Yourself

1. Write a program that checks if a number is positive. If it is, then check if it's greater than 100.
2. Make a program that checks if someone is eligible to drive: first check if `age >= 18`, then check if `has_license == True`.
3. Rewrite one of your nested conditions using `and` instead, and compare which version is easier to read.

17. `while` Loop

A `while` loop lets your program repeat a block of code as long as a condition is `True`. It's useful when you don't know in advance how many times you need to loop—for example, waiting for user input or running until some condition changes.

Deep Dive

Basic Structure

```python
while condition:
    # code runs as long as condition is True
```

Example

```
count = 1
while count <= 5:
    print("Count is:", count)
    count += 1
```

This loop prints numbers from 1 to 5. Each time, `count` increases by 1 until the condition `count <= 5` is no longer true.

Infinite Loops If the condition never becomes `False`, the loop will run forever. For example:

```
while True:
    print("This never ends!")
```

You must stop such loops manually (Ctrl+C in most terminals).

Using `break` to Stop Early You can break out of a `while` loop when needed:

```
x = 0
while x < 10:
    if x == 5:
        break
    print(x)
    x += 1
```

Using `continue` to Skip The `continue` keyword skips to the next iteration without finishing the rest of the loop body.

Common Use Cases

- Waiting for user input until valid
- Repeating a task until a condition is met
- Infinite background tasks with break conditions

Tiny Code

```
# Print even numbers less than 10
num = 0
while num < 10:
    num += 1
    if num % 2 == 1:
        continue
    print(num)
```

Why it Matters

The `while` loop gives your program flexibility to keep running until something changes. It's a powerful way to model "keep doing this until…" logic that often appears in real-world problems.

Try It Yourself

1. Write a loop that counts down from 10 to 1.
2. Create a loop that keeps asking the user for a password until the correct one is entered.
3. Use `while True` with a `break` to simulate a simple menu system (e.g., type `q` to quit).
4. Experiment with `continue` to skip printing odd numbers.

18. `for` Loop (range)

A `for` loop in Python is used to repeat a block of code a specific number of times. Unlike the `while` loop, which runs as long as a condition is true, the `for` loop usually goes through a sequence of values—often created with the built-in `range()` function.

Deep Dive

Basic Structure

```
for variable in sequence:
    # code runs for each item in the sequence
```

Using `range()` The `range()` function generates a sequence of numbers.

- `range(stop)` → numbers from `0` up to `stop - 1`
- `range(start, stop)` → numbers from `start` up to `stop - 1`
- `range(start, stop, step)` → numbers from `start` up to `stop - 1`, moving by `step`

Examples:

```
for i in range(5):
    print(i)       # 0, 1, 2, 3, 4

for i in range(2, 6):
    print(i)       # 2, 3, 4, 5
```

```
for i in range(0, 10, 2):
    print(i)       # 0, 2, 4, 6, 8
```

Looping with `else` A `for` loop can have an optional `else` block that runs if the loop finishes normally (not stopped by `break`).

```
for i in range(3):
    print(i)
else:
    print("Loop finished")
```

Common Patterns

- Counting a fixed number of times
- Iterating over list indexes
- Generating sequences for calculations

Tiny Code

```
# Print squares of numbers from 1 to 5
for n in range(1, 6):
    print(n, "squared is", n * n)
```

Why it Matters

The `for` loop is the most common way to repeat actions in Python when you know how many times to loop. It's simpler and clearer than a `while` loop for counting and iterating over ranges.

Try It Yourself

1. Write a loop that prints numbers 1 through 10.
2. Use `range()` with a step of 2 to print even numbers up to 20.
3. Write a loop that prints `"Python"` five times.
4. Create a loop with `range(10, 0, -1)` to count down from 10 to 1.

19. Loop Control (`break`, `continue`)

Sometimes you need more control over loops. Python provides two special keywords—`break` and `continue`—to change how a loop behaves. These allow you to stop a loop early or skip parts of it.

Deep Dive

`break` — Stop the Loop The `break` statement ends the loop immediately, even if the loop condition or range still has more values.

```
for i in range(10):
    if i == 5:
        break
    print(i)
# Output: 0, 1, 2, 3, 4
```

`continue` — Skip to Next Iteration The `continue` statement skips the rest of the loop body and moves to the next iteration.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
# Output: 0, 1, 3, 4
```

Using with `while` Loops Both `break` and `continue` work the same way in `while` loops.

```
x = 0
while x < 5:
    x += 1
    if x == 3:
        continue
    if x == 5:
        break
    print(x)
# Output: 1, 2, 4
```

When to Use

- `break` is useful when you find what you're looking for and don't need to continue looping.
- `continue` is useful when you want to skip over certain cases but still keep looping.

Tiny Code

```
# Find first multiple of 7
for n in range(1, 20):
    if n % 7 == 0:
        print("Found:", n)
        break

# Print only odd numbers
for n in range(1, 10):
    if n % 2 == 0:
        continue
    print(n)
```

Why it Matters

Without loop control, you would have to add extra complicated logic or duplicate code. `break` and `continue` give you fine-grained control, making loops cleaner, more efficient, and easier to understand.

Try It Yourself

1. Write a loop that prints numbers from 1 to 100, but stops when it reaches 42.
2. Write a loop that prints numbers from 1 to 10, but skips multiples of 3.
3. Combine both: loop through numbers 1 to 20, skip evens, and stop completely if you find 15.

20. Loop with `else`

In Python, a `for` or `while` loop can have an optional `else` block. The `else` part runs only if the loop finishes normally—that is, it isn't stopped early by a `break`. This feature is unique to Python and is often used when searching for something.

Deep Dive

Basic Structure

```
for item in sequence:
    # loop body
else:
    # runs if loop finishes without break
```

Example with `for`

```
for i in range(5):
    print(i)
else:
    print("Loop finished")
```

This prints numbers 0 through 4, then prints `"Loop finished"`.

Using with `break` If the loop ends because of `break`, the `else` block is skipped:

```
for i in range(5):
    if i == 3:
        break
    print(i)
else:
    print("Finished without break")
# Output: 0, 1, 2
```

Example with `while`

```
x = 0
while x < 3:
    print(x)
    x += 1
else:
    print("While loop ended normally")
```

Practical Use Case: Searching The `else` block is handy when searching for an item. If you find the item, `break` ends the loop; if not, the `else` runs.

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

for n in numbers:
    if n == 7:
        print("Found 7!")
```

```
        break
else:
    print("7 not found")
```

Tiny Code

```
for char in "Python":
    if char == "x":
        print("Found x!")
        break
else:
    print("No x in string")
```

Why it Matters

The `else` clause on loops lets you handle the "nothing found" case cleanly without needing extra flags or checks. It makes code shorter and easier to understand when searching or checking conditions across a loop.

Try It Yourself

1. Write a loop that searches for the number `10` in a list of numbers. If found, print `"Found"`. If not, let the `else` print `"Not found"`.
2. Create a `while` loop that counts from 1 to 5 and uses an `else` block to print `"Done counting"`.
3. Experiment with adding `break` inside your loop to see how it changes whether the `else` runs.

Chapter 3. Data Structures

21. Lists (creation & basics)

A list in Python is an ordered collection of items. Think of it like a container where you can store multiple values in a single variable—numbers, strings, or even other lists. Lists are one of the most commonly used data structures in Python because they're flexible and easy to work with.

Deep Dive

Creating Lists You create a list by putting values inside square brackets `[]`, separated by commas:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]
```

Lists can also be empty:

```
empty = []
```

Lists Are Ordered The items keep the order you put them in. If you create a list `[10, 20, 30]`, Python remembers that order unless you change it.

Lists Can Be Changed (Mutable) Unlike strings or tuples, lists can be modified after creation—you can add, remove, or replace elements.

Accessing Elements Each item in a list has an index (position), starting at 0:

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])   # "apple"
print(fruits[2])   # "cherry"
```

Length of a List You can find out how many items a list has with `len()`:

```
print(len(fruits))  # 3
```

Quick Summary Table

Operation	Example	Result
Create a list	`nums = [1, 2, 3]`	`[1, 2, 3]`
Empty list	`empty = []`	`[]`
Access by index	`nums[0]`	`1`
Last element	`nums[-1]`	`3`
Length of list	`len(nums)`	`3`

Tiny Code

```
colors = ["red", "green", "blue"]

print(colors)        # ['red', 'green', 'blue']
print(colors[1])     # 'green'
print(len(colors))   # 3
```

Why it Matters

Lists let you store and organize multiple values in one place. Without lists, you'd need a separate variable for each value, which quickly becomes messy. Lists are the foundation for handling collections of data in Python.

Try It Yourself

1. Create a list of five animals and print the whole list.
2. Print the first and last element of your list using indexes.
3. Make an empty list called `shopping_cart` and check its length.
4. Try storing mixed types in one list (like a number, string, and boolean) and print it.

22. List Indexing & Slicing

Lists in Python are ordered, which means each item has a position (index). You can use indexes to get specific elements, or slices to get parts of the list.

Deep Dive

Indexing Basics Indexes start at 0 for the first element:

```
fruits = ["apple", "banana", "cherry", "date"]
print(fruits[0])   # "apple"
print(fruits[2])   # "cherry"
```

Negative indexes count from the end:

```
print(fruits[-1])  # "date"
print(fruits[-2])  # "cherry"
```

Slicing Basics Slicing lets you grab a portion of a list. The syntax is:

```
list[start:stop]
```

It includes `start` but stops just before `stop`.

```
print(fruits[1:3])   # ['banana', 'cherry']
```

If you leave out `start`, Python begins at the start of the list:

```
print(fruits[:2])    # ['apple', 'banana']
```

If you leave out `stop`, Python goes to the end:

```
print(fruits[2:])    # ['cherry', 'date']
```

Slicing with Step You can add a third number for step size:

```
numbers = [0, 1, 2, 3, 4, 5]
print(numbers[::2])   # [0, 2, 4]
print(numbers[1::2])  # [1, 3, 5]
```

Reversing a list is easy with step `-1`:

```
print(numbers[::-1])  # [5, 4, 3, 2, 1, 0]
```

Quick Summary Table

Operation	Example	Result
First element	`fruits[0]`	`"apple"`
Last element	`fruits[-1]`	`"date"`
Slice (index 1–2)	`fruits[1:3]`	`['banana', 'cherry']`
From start to 2	`fruits[:2]`	`['apple', 'banana']`
From 2 to end	`fruits[2:]`	`['cherry', 'date']`
Every 2nd element	`numbers[::2]`	`[0, 2, 4]`
Reverse list	`numbers[::-1]`	`[5, 4, 3, 2, 1, 0]`

Tiny Code

```
colors = ["red", "green", "blue", "yellow"]

print(colors[0])     # red
print(colors[-1])    # yellow
print(colors[1:3])   # ['green', 'blue']
print(colors[::-1])  # ['yellow', 'blue', 'green', 'red']
```

Why it Matters

Indexing and slicing make it easy to get exactly the parts of a list you need. Whether you're grabbing one item, a range of items, or reversing the list, these tools are essential for working with collections of data.

Try It Yourself

1. Make a list of 6 numbers and print the first, third, and last elements.
2. Slice your list to get the middle three elements.
3. Use slicing with a step of 2 to get every other number.
4. Reverse the list using slicing and print the result.

23. List Methods (`append`, `extend`, etc.)

Lists in Python come with built-in methods that make it easy to add, remove, and modify items. These methods are powerful tools for managing collections of data.

Deep Dive

Adding Items

- `append(x)` → adds a single item to the end of the list.
- `extend(iterable)` → adds multiple items from another list (or any iterable).
- `insert(i, x)` → inserts an item at a specific position.

```
fruits = ["apple", "banana"]
fruits.append("cherry")     # ['apple', 'banana', 'cherry']
fruits.extend(["date", "fig"])  # ['apple', 'banana', 'cherry', 'date', 'fig']
fruits.insert(1, "kiwi")    # ['apple', 'kiwi', 'banana', 'cherry', 'date', 'fig']
```

Removing Items

- `remove(x)` → removes the first occurrence of `x`.
- `pop(i)` → removes and returns the item at index `i` (defaults to last).
- `clear()` → removes all items.

```
fruits.remove("banana")   # ['apple', 'kiwi', 'cherry', 'date', 'fig']
fruits.pop(2)             # removes 'cherry'
fruits.clear()            # []
```

Finding and Counting

- `index(x)` → returns the position of the first occurrence of `x`.
- `count(x)` → returns how many times `x` appears.

```
nums = [1, 2, 2, 3]
print(nums.index(2))  # 1
print(nums.count(2))  # 2
```

Sorting and Reversing

- `sort()` → sorts the list in place.
- `reverse()` → reverses the order of items in place.
- `sorted(list)` → returns a new sorted list without changing the original.

```
letters = ["b", "a", "d", "c"]
letters.sort()       # ['a', 'b', 'c', 'd']
letters.reverse()    # ['d', 'c', 'b', 'a']
```

Quick Summary Table

Method	Purpose	Example
`append(x)`	Add one item at the end	`lst.append(5)`
`extend()`	Add many items	`lst.extend([6,7])`
`insert()`	Insert at a position	`lst.insert(1, "hi")`
`remove(x)`	Remove first matching value	`lst.remove("hi")`
`pop(i)`	Remove by index (or last by default)	`lst.pop(0)`
`clear()`	Empty the list	`lst.clear()`
`index(x)`	Find index of first match	`lst.index(2)`
`count(x)`	Count how many times value appears	`lst.count(2)`
`sort()`	Sort list in place	`lst.sort()`
`reverse()`	Reverse order in place	`lst.reverse()`

Tiny Code

```
colors = ["red", "blue"]

colors.append("green")
colors.extend(["yellow", "purple"])
colors.insert(2, "orange")

print(colors)  # ['red', 'blue', 'orange', 'green', 'yellow', 'purple']

colors.remove("blue")
last = colors.pop()
print(last)    # 'purple'

print(colors.count("red"))   # 1
colors.sort()
print(colors)  # ['green', 'orange', 'red', 'yellow']
```

Why it Matters

List methods are essential for real-world programming, where data is always changing. Being able to add, remove, and reorder items makes lists versatile tools for tasks like managing to-do lists, processing datasets, or handling user inputs.

Try It Yourself

1. Start with a list of three numbers. Add two more using `append()` and `extend()`.
2. Insert a number at the beginning of the list.
3. Remove one number using `remove()`, then use `pop()` to remove the last one.
4. Sort your list and then reverse it. Print the result at each step.

24. Tuples

A tuple is an ordered collection of items, just like a list, but with one big difference: tuples are immutable. This means once you create a tuple, you cannot change its contents—no adding, removing, or modifying items. Tuples are useful when you want to store data that should not be altered.

Deep Dive

Creating Tuples You create a tuple using parentheses `()` instead of square brackets:

```
numbers = (1, 2, 3)
fruits = ("apple", "banana", "cherry")
```

Tuples can hold mixed data types just like lists:

```
mixed = (1, "hello", 3.14, True)
```

For a single-item tuple, you must include a trailing comma:

```
single = (5,)
print(type(single))   # <class 'tuple'>
```

Accessing Elements Tuples use the same indexing and slicing as lists:

```
print(fruits[0])     # "apple"
print(fruits[-1])    # "cherry"
print(fruits[0:2])   # ("apple", "banana")
```

Immutability You cannot modify a tuple after it's created:

```
fruits[0] = "pear"   #  Error: TypeError
```

Tuple Packing and Unpacking You can pack multiple values into a tuple and unpack them into variables:

```
point = (3, 4)
x, y = point
print(x, y)   # 3 4
```

Use Cases

- Returning multiple values from a function.
- Fixed collections of data (e.g., coordinates, RGB colors).
- Keys in dictionaries (since tuples are hashable, lists are not).

Quick Summary Table

Feature	List	Tuple
Syntax	`[1, 2, 3]`	`(1, 2, 3)`
Mutability	Mutable (can change)	Immutable (cannot)
Methods	Many (`append`, etc.)	Few (`count`, `index`)
Performance	Slower	Faster (lightweight)

Tiny Code

```
colors = ("red", "green", "blue")

print(colors[1])       # green
print(len(colors))     # 3

# Unpacking
r, g, b = colors
print(r, b)            # red blue

# Methods
print(colors.index("blue"))  # 2
print(colors.count("red"))   # 1
```

Why it Matters

Tuples give you a safe way to group data that should not be changed, protecting against accidental modifications. They are also slightly faster than lists, making them useful when performance matters and immutability is desired.

Try It Yourself

1. Create a tuple with three of your favorite foods and print the second one.
2. Try changing one element—observe the error.
3. Use unpacking to assign a tuple `(10, 20, 30)` into variables `a, b, c`.
4. Create a dictionary where the key is a tuple of coordinates `(x, y)` and the value is a place name.

25. Sets

A set in Python is an unordered collection of unique items. Sets are useful when you need to store data without duplicates or when you want to perform mathematical operations like union and intersection.

Deep Dive

Creating Sets You can create a set using curly braces `{}` or the `set()` function:

```
fruits = {"apple", "banana", "cherry"}
numbers = set([1, 2, 3, 2, 1])  # duplicates removed
print(numbers)  # {1, 2, 3}
```

No Duplicates If you try to add duplicates, Python automatically ignores them:

```
colors = {"red", "blue", "red"}
print(colors)  # {'red', 'blue'}
```

Unordered Sets do not preserve order. You cannot access elements by index (`set[0]`).

Adding and Removing Items

- `add(x)` → adds an item.
- `update(iterable)` → adds multiple items.
- `remove(x)` → removes an item (error if not found).
- `discard(x)` → removes an item (no error if not found).
- `pop()` → removes and returns a random item.
- `clear()` → removes all items.

```
s = {1, 2}
s.add(3)           # {1, 2, 3}
s.update([4, 5])   # {1, 2, 3, 4, 5}
s.remove(2)        # {1, 3, 4, 5}
s.discard(10)      # no error
```

Membership Test Checking if an item exists is fast:

```
print("apple" in fruits)  # True
```

Set Operations Sets are great for math-like operations:

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b)   # union → {1, 2, 3, 4, 5}
print(a & b)   # intersection → {3}
print(a - b)   # difference → {1, 2}
print(a ^ b)   # symmetric difference → {1, 2, 4, 5}
```

Quick Summary Table

Operation	Example	Result	
Create set	`{1, 2, 3}`	`{1, 2, 3}`	
Add item	`s.add(4)`	`{1, 2, 3, 4}`	
Remove item	`s.remove(2)`	error if not found	
Discard item	`s.discard(2)`	safe remove	
Union	‘a	b‘	combine sets
Intersection	`a & b`	common items	
Difference	`a - b`	items only in `a`	
Symmetric difference	`a ^ b`	items in `a` or `b`, not both	

Tiny Code

```
numbers = {1, 2, 3, 3, 2}
print(numbers)   # {1, 2, 3}

numbers.add(4)
numbers.discard(1)
print(numbers)   # {2, 3, 4}

odds = {1, 3, 5}
evens = {2, 4, 6}
print(odds | evens)   # {1, 2, 3, 4, 5, 6}
```

Why it Matters

Sets make it easy to eliminate duplicates and perform operations like union or intersection, which are common in data analysis, algorithms, and everyday programming tasks. They are also optimized for fast membership testing.

Try It Yourself

1. Create a set of your favorite fruits and add a new one.
2. Try adding the same fruit again—see how duplicates are ignored.
3. Make two sets of numbers and print their union, intersection, and difference.
4. Use `in` to check if an element is in the set.

26. Set Operations (union, intersection)

Sets in Python shine when you use them for mathematical-style operations. They let you combine, compare, and filter items in powerful ways. These operations are very fast and are often used in data processing, searching, and analysis.

Deep Dive

Union (`|` or `.union()`) The union of two sets contains all unique items from both.

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)           # {1, 2, 3, 4, 5}
print(a.union(b))      # {1, 2, 3, 4, 5}
```

Intersection (`&` or `.intersection()`) The intersection contains only items present in both sets.

```
print(a & b)           # {3}
print(a.intersection(b))  # {3}
```

Difference (`-` or `.difference()`) The difference contains items in the first set but not the second.

```
print(a - b)           # {1, 2}
print(b - a)           # {4, 5}
```

Symmetric Difference (`^` or `.symmetric_difference()`) The symmetric difference contains items in either set, but not both.

```
print(a ^ b)           # {1, 2, 4, 5}
print(a.symmetric_difference(b))  # {1, 2, 4, 5}
```

Subset and Superset Checks

- `a <= b` → checks if `a` is a subset of `b`.
- `a >= b` → checks if `a` is a superset of `b`.

```
x = {1, 2}
y = {1, 2, 3}
print(x <= y)   # True (x is subset of y)
print(y >= x)   # True (y is superset of x)
```

Quick Summary Table

Operation	Sym-bol	Exam-ple	Result		
Union	‘	‘	‘a	b‘	all unique items
Intersection	`&`	`a & b`	common items		
Difference	`-`	`a - b`	in `a` not `b`		
Symmetric difference	`^`	`a ^ b`	in `a` or `b`, not both		
Subset	`<=`	`a <= b`	True/False		
Superset	`>=`	`a >= b`	True/False		

Tiny Code

```
a = {1, 2, 3}
b = {3, 4, 5}

print("Union:", a | b)              # {1, 2, 3, 4, 5}
print("Intersection:", a & b)       # {3}
print("Difference:", a - b)         # {1, 2}
print("SymDiff:", a ^ b)            # {1, 2, 4, 5}

print("Subset?", {1, 2} <= a)       # True
print("Superset?", a >= {2, 3})     # True
```

Why it Matters

Set operations allow you to quickly solve problems like finding common elements, removing duplicates, or checking membership across collections. They map directly to real-world logic such as “all users,” “users in both groups,” or “items missing from one list.”

Try It Yourself

1. Make two sets of numbers: `{1, 2, 3, 4}` and `{3, 4, 5, 6}`. Find their union, intersection, and difference.
2. Create a set of vowels and a set of letters in the word `"python"`. Find the intersection to see which vowels appear.
3. Check if `{1, 2}` is a subset of `{1, 2, 3, 4}`.
4. Try symmetric difference between `{a, b, c}` and `{b, c, d}`.

27. Dictionaries (creation & basics)

A dictionary in Python is a collection of key–value pairs. Instead of accessing items by index like lists, you access them by their keys. This makes dictionaries very powerful for storing and retrieving data when you want to associate labels with values.

Deep Dive

Creating Dictionaries You create a dictionary using curly braces `{}` with keys and values separated by colons:

```
person = {"name": "Alice", "age": 25, "city": "Paris"}
```

Accessing Values You get values by their keys, not by position:

```
print(person["name"])   # "Alice"
print(person["age"])    # 25
```

Adding and Updating Dictionaries are mutable—you can add new key–value pairs or update existing ones:

```
person["job"] = "Engineer"
person["age"] = 26
```

Keys Must Be Unique If you repeat a key, the latest value will overwrite the earlier one:

```
data = {"a": 1, "a": 2}
print(data)   # {"a": 2}
```

Dictionary Keys and Values

- Keys must be immutable types (strings, numbers, tuples).
- Values can be any type: strings, numbers, lists, or even other dictionaries.

Empty Dictionary You can start with an empty dictionary:

```
empty = {}
```

Quick Summary Table

Operation	Example	Result
Create dictionary	`{"a": 1, "b": 2}`	`{'a': 1, 'b': 2}`
Access by key	`person["name"]`	`"Alice"`
Add / update	`person["age"] = 30`	changes value for `"age"`
Empty dictionary	`{}`	`{}`
Mixed values allowed	`{"id": 1, "tags": ["x", "y"], "active": True}`	valid dictionary

Tiny Code

```
car = {"brand": "Toyota", "model": "Corolla", "year": 2020}

print(car["brand"])      # Toyota
car["year"] = 2021       # update value
car["color"] = "blue"    # add new key
print(car)
```

Why it Matters

Dictionaries give you a natural way to organize and retrieve data by name instead of position. They are essential for representing structured data, like database records, configurations, or JSON data from APIs.

Try It Yourself

1. Create a dictionary called `student` with keys `"name"`, `"age"`, and `"grade"`.
2. Access and print the `"grade"`.
3. Update the `"age"` to a new number.
4. Add a new key `"passed"` with the value `True`.
5. Print the whole dictionary to see the changes.

28. Dictionary Methods

Dictionaries come with built-in methods that make it easy to work with their keys and values. These methods let you add, remove, and inspect data in a structured way.

Deep Dive

Accessing Keys, Values, and Items

- `dict.keys()` → returns all keys.
- `dict.values()` → returns all values.
- `dict.items()` → returns pairs of `(key, value)`.

```
person = {"name": "Alice", "age": 25}

print(person.keys())    # dict_keys(['name', 'age'])
print(person.values())  # dict_values(['Alice', 25])
print(person.items())   # dict_items([('name', 'Alice'), ('age', 25)])
```

Adding and Updating

- `update(other_dict)` → adds or updates key–value pairs.

```
person.update({"age": 26, "city": "Paris"})
```

Removing Items

- `pop(key)` → removes and returns the value for a key.
- `popitem()` → removes and returns the last inserted pair.
- `del dict[key]` → deletes a key–value pair.
- `clear()` → empties the dictionary.

```
print(person.pop("age"))      # 26
print(person.popitem())       # ('city', 'Paris')
del person["name"]            # removes "name"
person.clear()                # {}
```

Get with Default

- `get(key, default)` → safely gets a value; returns `default` if the key doesn't exist.

```
person = {"name": "Alice"}
print(person.get("age", "Not found"))  # "Not found"
```

From Keys

- `dict.fromkeys(keys, value)` → creates a dictionary with given keys and default value.

```
keys = ["a", "b", "c"]
print(dict.fromkeys(keys, 0))   # {'a': 0, 'b': 0, 'c': 0}
```

Quick Summary Table

Method	Purpose	Example
`keys()`	Get all keys	`person.keys()`
`values()`	Get all values	`person.values()`
`items()`	Get all pairs	`person.items()`
`update()`	Add/update multiple pairs	`person.update({"age": 26})`
`pop(key)`	Remove by key, return value	`person.pop("name")`
`popitem()`	Remove last inserted pair	`person.popitem()`
`get()`	Safe value access with default	`person.get("city", "Unknown")`
`clear()`	Remove all pairs	`person.clear()`
`fromkeys()`	Create new dict from keys	`dict.fromkeys(["x", "y"], 1)`

Tiny Code

```
student = {"name": "Bob", "age": 20, "grade": "A"}

print(student.keys())       # dict_keys(['name', 'age', 'grade'])
print(student.get("city", "N/A"))  # N/A

student.update({"age": 21})
print(student)

student.pop("grade")
print(student)
```

Why it Matters

Dictionary methods let you manipulate structured data efficiently. Whether you're cleaning data, merging information, or safely handling missing values, these methods are essential for working with real-world datasets and configurations.

Try It Yourself

1. Create a dictionary `book` with `"title"`, `"author"`, and `"year"`.
2. Use `keys()`, `values()`, and `items()` to inspect it.
3. Update the `"year"` to the current year using `update()`.
4. Use `get()` to safely access a missing `"publisher"` key with a default value.
5. Clear the dictionary with `clear()`.

30. Nested Structures

A nested structure means putting one data structure inside another—for example, a list of lists, a dictionary containing lists, or even a list of dictionaries. Nested structures are common when representing more complex, real-world data.

Deep Dive

Lists Inside Lists You can create multi-dimensional lists:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix[0][1])   # 2
```

Dictionaries with Lists Values in a dictionary can be lists:

```
student = {
    "name": "Alice",
    "grades": [85, 90, 92]
}
print(student["grades"][1])   # 90
```

Lists of Dictionaries A list can contain multiple dictionaries, useful for structured records:

```
people = [
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 30}
]
print(people[1]["name"])   # Bob
```

Dictionaries of Dictionaries Dictionaries can be nested, too:

```
users = {
    "alice": {"age": 25, "city": "Paris"},
    "bob": {"age": 30, "city": "London"}
}
print(users["bob"]["city"])   # London
```

Iteration Through Nested Structures You can use loops inside loops to navigate deeper levels:

```
for row in matrix:
    for val in row:
        print(val, end=" ")
```

Quick Summary Table

Nested Type	Example	Access Example
List of lists	`[[1,2],[3,4]]`	`x[0][1] → 2`
Dict with list	`{"scores":[10,20]}`	`d["scores"][0] → 10`
List of dicts	`[{"n":"a"},{"n":"b"}]`	`lst[1]["n"] → "b"`
Dict of dicts	`{"a":{"x":1}, "b":{"x":2}}`	`d["b"]["x"] → 2`

Tiny Code

```
classrooms = {
    "A": ["Alice", "Bob"],
    "B": ["Charlie", "Diana"]
}

for room, students in classrooms.items():
    print("Room:", room)
    for student in students:
        print("-", student)
```

Why it Matters

Real-world data is rarely flat—it's often hierarchical or structured in layers (like JSON from APIs, database rows with embedded fields, or spreadsheets). Nested structures let you represent and work with this complexity directly in Python.

Try It Yourself

1. Create a list of lists to represent a 3×3 grid and print the center value.
2. Make a dictionary with a key `"friends"` pointing to a list of three names. Print the second name.
3. Create a list of dictionaries, each with `"title"` and `"year"`, for your favorite movies. Print the title of the last one.
4. Build a dictionary of dictionaries representing two countries with their capital cities, then print one capital.

Chapter 4. Functions

31. Defining a Function (`def`)

A function is a reusable block of code that performs a specific task. Functions let you avoid repetition, organize your code, and make programs easier to understand. In Python, you define a function using the `def` keyword.

Deep Dive

Basic Function Definition

```
def greet():
    print("Hello!")
```

Calling `greet()` runs the code inside.

Functions with Parameters You can pass data into functions using parameters:

```
def greet(name):
    print("Hello,", name)

greet("Alice")   # Hello, Alice
```

Return Values Functions can return data with `return`:

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

result = add(3, 4)
print(result)   # 7
```

Default Behavior

- If a function doesn't explicitly `return`, it returns `None`.
- Functions can be defined before or after other code, but must be defined before they are called.

Why Use Functions?

- Reusability: write once, use many times.
- Readability: group code into meaningful chunks.
- Maintainability: easier to test and fix.

Quick Summary Table

Feature	Example	Notes
Define function	`def f():`	Code block indented
Call function	`f()`	Executes the block
With parameter	`def f(x):`	Pass value when calling
With return	`def f(x): return x+1`	Gives back a value
Implicit return	function without `return`	Returns `None`

Tiny Code

```
def square(n):
    return n * n

print(square(5))   # 25

def welcome(name):
    print("Welcome,", name)

welcome("Bob")     # Welcome, Bob
```

Why it Matters

Functions are the building blocks of programs. They let you break down complex problems into smaller pieces, reuse code efficiently, and make your programs easier to maintain and understand.

Try It Yourself

1. Write a function `hello()` that prints `"Hello, Python!"`.
2. Write a function `double(x)` that returns twice the number given.
3. Define a function `say_name(name)` that prints `"My name is ..."` with the input name.
4. Call your functions multiple times to see the benefits of reuse.

32. Function Arguments

Functions can take arguments (also called parameters) so you can pass information into them. Arguments make functions flexible because they can work with different inputs instead of being hardcoded.

Deep Dive

Positional Arguments The most common type—values are matched to parameters in order.

```
def greet(name, age):
    print("Hello,", name, "you are", age, "years old")

greet("Alice", 25)
```

Keyword Arguments You can pass values by naming the parameters. This makes the call clearer and order doesn't matter.

```
greet(age=30, name="Bob")
```

Default Arguments You can give parameters default values, making them optional when calling the function.

```
def greet(name, age=18):
    print("Hello,", name, "you are", age)

greet("Charlie")      # uses default age = 18
greet("Diana", 22)    # overrides default
```

Mixing Arguments When mixing, positional arguments come first, then keyword arguments.

```
def student(name, grade="A"):
    print(name, "got grade", grade)

student("Eva")            # Eva got grade A
student("Frank", grade="B")
```

Wrong Usage Causes Errors

```
greet(25, "Alice")   # order matters for positional
```

Quick Summary Table

Type	Example call	Notes
Positional	`f(1, 2)`	Order matters
Keyword	`f(b=2, a=1)`	Order doesn't matter
Default value	`f(1)` when defined as `f(a, b=2)`	Uses default if missing
Mixed	`f(1, b=3)`	Positional first, keyword next

Tiny Code

```
def introduce(name, country="Unknown"):
    print("I am", name, "from", country)

introduce("Alice")                # I am Alice from Unknown
introduce("Bob", "France")        # I am Bob from France
introduce(name="Charlie", country="Japan")
```

Why it Matters

Arguments let you write one function that works in many situations. Instead of duplicating code, you can pass in different values and reuse the same function. This is one of the core ideas of programming.

Try It Yourself

1. Write a function `add(a, b)` that prints the sum of two numbers.
2. Call it with both positional (`add(3, 4)`) and keyword (`add(b=4, a=3)`) arguments.
3. Create a function `greet(name="Friend")` that has a default value for `name`. Call it with and without providing the argument.
4. Write a function `power(base, exponent=2)` that returns `base` raised to `exponent`. Call it with one and two arguments.

33. Default & Keyword Arguments

Python functions can define default values for parameters and accept keyword arguments when called. These features make functions flexible and easier to use by reducing how much you need to type and improving readability.

Deep Dive

Default Arguments When defining a function, you can give a parameter a default value. If the caller doesn't provide it, Python uses the default.

```
def greet(name="Friend"):
    print("Hello,", name)

greet()              # Hello, Friend
greet("Alice")       # Hello, Alice
```

Multiple Defaults You can set defaults for more than one parameter.

```
def connect(host="localhost", port=8080):
    print("Connecting to", host, "on port", port)

connect()                      # localhost, 8080
connect("example.com")         # example.com, 8080
connect(port=5000)             # localhost, 5000
```

Keyword Arguments When calling a function, you can use parameter names. This makes it clear what each value means, and order doesn't matter.

```
def introduce(name, age):
    print(name, "is", age, "years old")

introduce(age=30, name="Bob")  # Bob is 30 years old
```

Mixing Positional and Keyword Arguments You can mix both, but positional arguments must come first.

```
def describe(animal, sound="unknown"):
    print(animal, "goes", sound)

describe("Dog")                    # Dog goes unknown
describe("Cat", sound="Meow")      # Cat goes Meow
```

Important Rule Default arguments are evaluated once, when the function is defined. Be careful with mutable defaults like lists or dictionaries—they can persist changes between calls.

```
def add_item(item, container=[]):
    container.append(item)
    return container

print(add_item(1))   # [1]
print(add_item(2))   # [1, 2]  ← reused same list!
```

The safe way is:

```
def add_item(item, container=None):
    if container is None:
        container = []
    container.append(item)
    return container
```

Quick Summary Table

Feature	Example	Benefit
Default parameter	`def f(x=10)`	Optional arguments
Keyword argument call	`f(y=2, x=1)`	Clear meaning, order-free

Feature	Example	Benefit
Mixing positional+keyword	`f(1, y=2)`	Flexible calls
Mutable default trap	`def f(lst=[])`	Avoid with `None` as default

Tiny Code

```
def greet(name="Guest", lang="en"):
    if lang == "en":
        print("Hello,", name)
    elif lang == "fr":
        print("Bonjour,", name)
    else:
        print("Hi,", name)

greet()
greet("Alice")
greet("Bob", lang="fr")
```

Why it Matters

Default and keyword arguments make functions more user-friendly. They reduce repetitive code, prevent errors from missing values, and improve readability when functions have many parameters.

Try It Yourself

1. Write a function `multiply(a, b=2)` that returns `a * b`. Call it with one argument and with two.
2. Create a function `profile(name, age=18, city="Unknown")` and call it using keyword arguments in any order.
3. Test the mutable default trap by defining a function with `list=[]`. See how it behaves after multiple calls.
4. Rewrite it using `None` as the default and verify the issue is fixed.

34. Return Values

Functions don't just perform actions—they can also send results back using the `return` statement. This makes functions powerful, because you can store their output, use it in calculations, or pass it into other functions.

Deep Dive

Basic Return

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

result = add(3, 4)
print(result)   # 7
```

When Python hits `return`, the function stops and sends the value back.

Returning Multiple Values Python functions can return more than one value by returning a tuple:

```
def get_stats(numbers):
    return min(numbers), max(numbers), sum(numbers) / len(numbers)

low, high, avg = get_stats([10, 20, 30])
print(low, high, avg)   # 10 30 20.0
```

No Return = `None` If a function doesn't have a `return`, it automatically returns `None`.

```
def say_hello():
    print("Hello")

result = say_hello()
print(result)   # None
```

Return vs Print

- `print()` shows something on the screen.
- `return` gives a value back to the program.

```
def square(x):
    return x * x

print(square(5))   # 25 (returned value printed)
```

Without `return`, you can't reuse the result later.

Early Return You can use `return` to exit a function early.

```
def safe_divide(a, b):
    if b == 0:
        return "Cannot divide by zero"
    return a / b
```

Quick Summary Table

Behavior	Example	Result
Single return	`return x + y`	one value
Multiple return	`return a, b`	tuple of values
No return	no `return`	`None`
Return vs print	`return` gives data, `print` shows data	difference in purpose

Tiny Code

```
def cube(n):
    return n  3

def min_max(nums):
    return min(nums), max(nums)

print(cube(4))             # 64
low, high = min_max([3, 7, 2, 9])
print(low, high)           # 2 9
```

Why it Matters

Return values make functions reusable building blocks. Instead of just displaying results, functions can calculate and hand back values, letting you compose larger programs from smaller pieces.

Try It Yourself

1. Write a function `square(n)` that returns the square of a number.
2. Create a function `divide(a, b)` that returns the result, but if `b` is 0, return `"Error"`.
3. Write a function `circle_area(radius)` that returns the area using `3.14 * r * r`.
4. Make a function that returns both the smallest and largest number from a list.

35. Variable Scope (local vs global)

In Python, scope refers to where a variable can be accessed in your code. Variables created inside a function exist only there, while variables created outside are available globally. Understanding scope helps avoid bugs and keeps code organized.

Deep Dive

Local Variables A variable created inside a function is local to that function. It only exists while the function runs.

```
def greet():
    message = "Hello"   # local variable
    print(message)

greet()
# print(message)  Error: message not defined
```

Global Variables A variable created outside functions is global and can be used anywhere.

```
name = "Alice"   # global variable

def say_name():
    print("My name is", name)

say_name()       # works fine
```

Local vs Global Priority If a local variable has the same name as a global one, Python uses the local one inside the function.

```
x = 10   # global

def show():
    x = 5   # local
    print(x)

show()      # 5
print(x)    # 10
```

Using `global` Keyword If you want to modify a global variable inside a function, use `global`.

```
count = 0

def increase():
    global count
    count += 1

increase()
print(count)   # 1
```

Best Practice

- Use local variables whenever possible—they are safer and easier to manage.
- Avoid modifying global variables inside functions unless absolutely necessary.

Quick Summary Table

Variable Type	Defined Where	Accessible Where
Local	Inside a function	Only inside that function
Global	Outside functions	Anywhere in the program
Shadowing	Local overrides global	Local used inside function
`global`	Marks variable as global	Allows modification in function

Tiny Code

```
x = 100   # global

def test():
    x = 50   # local
```

```
    print("Inside function:", x)

test()
print("Outside function:", x)
```

Why it Matters

Scope controls variable visibility and prevents accidental overwriting of values. By understanding local vs global variables, you can write cleaner, more reliable code that avoids confusing bugs.

Try It Yourself

1. Create a global variable `city = "Paris"` and write a function that prints it.
2. Define a function with a local variable `city = "London"` and see which value prints inside vs outside.
3. Make a counter using a global variable and a function that increases it with the `global` keyword.
4. Write two functions that each define their own local variable with the same name, and confirm they don't affect each other.

36. `*args` and `kwargs`

In Python, functions can accept a flexible number of arguments using `*args` and `kwargs`. These let you handle situations where you don't know in advance how many inputs the user will provide.

Deep Dive

`*args` → Variable Positional Arguments

- Collects extra positional arguments into a tuple.

```
def add_all(*args):
    print(args)

add_all(1, 2, 3)   # (1, 2, 3)
```

You can loop through `args` to process them:

```
def add_all(*args):
    return sum(args)

print(add_all(1, 2, 3, 4))   # 10
```

`kwargs` → Variable Keyword Arguments

- Collects extra keyword arguments into a dictionary.

```
def show_info(kwargs):
    print(kwargs)

show_info(name="Alice", age=25)
# {'name': 'Alice', 'age': 25}
```

You can access values like a normal dictionary:

```
def show_info(kwargs):
    for key, value in kwargs.items():
        print(key, "=", value)

show_info(city="Paris", country="France")
```

Combining `*args` and `kwargs` You can use both in the same function, but `*args` must come before `kwargs`.

```
def demo(a, *args, kwargs):
    print("a:", a)
    print("args:", args)
    print("kwargs:", kwargs)

demo(1, 2, 3, x=10, y=20)
# a: 1
# args: (2, 3)
# kwargs: {'x': 10, 'y': 20}
```

Unpacking with `*` and `You can also use `*` and` to unpack lists/tuples and dictionaries into arguments.

```
nums = [1, 2, 3]
print(add_all(*nums))   # 6

options = {"city": "Tokyo", "year": 2025}
show_info(options)
```

Quick Summary Table

Feature	Collects Into	Example Call	Example Result
`*args`	Tuple	`f(1,2,3)`	`(1,2,3)`
`kwargs`	Dictionary	`f(a=1, b=2)`	`{'a':1,'b':2}`
Both combined	args + kwargs	`f(1,2, x=10)`	`args=(2,), kwargs={'x':10}`
Unpacking `*`	Splits list	`f(*[1,2])`	like `f(1,2)`
Unpacking ‘| Splits dict |f({‘a’:1})| likef(a=1)‘			

Tiny Code

```
def greet(*names, options):
    for name in names:
        print("Hello,", name)
    if "lang" in options:
        print("Language:", options["lang"])

greet("Alice", "Bob", lang="English")
```

Why it Matters

`*args` and `kwargs` make functions more flexible and reusable. They let you handle unknown numbers of inputs, write cleaner APIs, and pass around configurations easily.

Try It Yourself

1. Write a function `multiply_all(*nums)` that multiplies any number of values.
2. Create a function `print_info(data)` that prints each key–value pair.
3. Combine them: `f(x, *args, kwargs)` and test with mixed inputs.
4. Experiment with unpacking a list into `*args` and a dictionary into `kwargs`.

37. Lambda Functions

A lambda function is a small, anonymous function defined with the keyword `lambda`. Unlike normal functions defined with `def`, lambda functions are written in a single line and don't need a name unless you assign them to a variable. They're often used for quick, throwaway functions.

Deep Dive

Basic Syntax

```
lambda arguments: expression
```

- `arguments` → input parameters.
- `expression` → a single expression that is evaluated and returned.

Example:

```
square = lambda x: x * x
print(square(5))   # 25
```

Multiple Arguments

```
add = lambda a, b: a + b
print(add(3, 4))   # 7
```

No Arguments

```
hello = lambda: "Hello!"
print(hello())     # Hello!
```

Use with Built-in Functions Lambdas are often used with `map()`, `filter()`, and `sorted()`.

- With `map()` to apply a function to all items:

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x * x, nums))
print(squares)   # [1, 4, 9, 16]
```

- With `filter()` to keep items that match a condition:

```
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)   # [2, 4]
```

- With `sorted()` to customize sorting:

```
words = ["banana", "apple", "cherry"]
words.sort(key=lambda w: len(w))
print(words)   # ['apple', 'banana', 'cherry']
```

Limitations

- Only one expression (no multiple lines).
- Can't contain statements like `print`, `return`, or loops (though you can call functions inside).
- Best for short, simple tasks.

Quick Summary Table

Feature	Example	Output
Single argument	`lambda x: x + 1`	Adds 1 to x
Multiple args	`lambda a, b: a * b`	Multiplies a and b
No args	`lambda: "hi"`	Returns "hi"
With `map()`	`map(lambda x: x*x, [1,2])`	[1, 4]
With `filter()`	`filter(lambda x: x>2, [1,2,3])`	[3]
With `sorted()`	`sorted(words, key=lambda w:len(w))`	Sorted by length

Tiny Code

```
nums = [5, 10, 15]

# Double numbers using lambda + map
doubles = list(map(lambda n: n * 2, nums))
print(doubles)   # [10, 20, 30]

# Filter numbers greater than 7
greater = list(filter(lambda n: n > 7, nums))
print(greater)   # [10, 15]
```

Why it Matters

Lambda functions let you write short, inline functions without cluttering your code. They're especially handy for quick data transformations, sorting, and filtering when defining a full function would be unnecessary.

Try It Yourself

1. Write a lambda function that adds 10 to a number.
2. Use a lambda with `filter()` to keep only odd numbers from a list.
3. Sort a list of names by their last letter using `sorted()` with a lambda key.
4. Use `map()` with a lambda to convert a list of Celsius temperatures into Fahrenheit.

38. Docstrings

A docstring (documentation string) is a special string placed inside functions, classes, or modules to explain what they do. Unlike comments, docstrings are stored at runtime and can be accessed with tools like `help()`. They are a key part of writing clean, professional Python code.

Deep Dive

Basic Function Docstring Docstrings are written using triple quotes (`""" ... """` or `''' ... '''`) right below the function definition:

```
def greet(name):
    """Return a greeting message for the given name."""
    return "Hello, " + name
```

Accessing Docstrings You can retrieve the docstring with:

```
print(greet.__doc__)
help(greet)
```

Multi-Line Docstrings For more complex functions, use multiple lines:

```
def add(a, b):
    """
    Add two numbers and return the result.

    Parameters:
        a (int or float): First number.
        b (int or float): Second number.

    Returns:
        int or float: The sum of a and b.
    """
    return a + b
```

Docstrings for Classes and Modules

- For classes:

```
class Person:
    """A simple class representing a person."""
    def __init__(self, name):
        self.name = name
```

- For modules (at the very top of a file):

```
"""
This module provides math helper functions
like factorial and Fibonacci.
"""
```

PEP 257 Conventions Python has conventions for docstrings:

1. Start with a short summary in one line.
2. Leave a blank line after the summary if you add more detail.
3. Use triple quotes even for one-liners.

Quick Summary Table

Where Used	Example Placement	Purpose
Function	Inside function body	Explain what it does/returns
Class	Inside class definition	Describe the class purpose
Module	At top of file	Overview of the whole module
Accessing	`obj.__doc__`, `help()`	See documentation

Tiny Code

```
def factorial(n):
    """Calculate the factorial of n using recursion."""
    return 1 if n == 0 else n * factorial(n - 1)

print(factorial.__doc__)
```

Why it Matters

Docstrings turn your code into self-documenting programs. They help others (and your future self) understand how functions, classes, and modules should be used without reading all the code. Tools like Sphinx and IDEs also use docstrings to generate documentation automatically.

Try It Yourself

1. Write a function `square(n)` with a one-line docstring explaining what it does.
2. Create a function `divide(a, b)` with a multi-line docstring that explains parameters and return value.
3. Add a class `Car` with a docstring describing its purpose.
4. Use `help()` on your function or class to see the docstring displayed.

39. Recursive Functions

A recursive function is a function that calls itself in order to solve a problem. Recursion is useful when a problem can be broken down into smaller, similar subproblems—like calculating factorials, traversing trees, or solving puzzles.

Deep Dive

Basic Structure A recursive function always has two parts:

1. Base case → the condition that stops the recursion.
2. Recursive case → the function calls itself with a smaller/simpler problem.

```
def countdown(n):
    if n == 0:              # base case
        print("Done!")
    else:
        print(n)
        countdown(n - 1)    # recursive case
```

Example 1: Factorial The factorial of `n` is `n * (n-1) * (n-2) * ... * 1`.

```
def factorial(n):
    if n == 0:      # base case
        return 1
    return n * factorial(n - 1)   # recursive case

print(factorial(5))   # 120
```

Example 2: Fibonacci Sequence Each Fibonacci number is the sum of the previous two.

```
def fib(n):
    if n <= 1:   # base case
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(6))   # 8
```

Potential Issues

- Infinite recursion: forgetting a base case causes the function to call itself forever, leading to an error (`RecursionError`).
- Performance: recursion can be slower and use more memory than loops for large inputs.

Quick Summary Table

Term	Meaning	Example
Base case	Condition that stops recursion	`if n == 0: return 1`
Recursive case	Function calls itself with smaller input	`return n * f(n-1)`
Infinite recursion	Missing/incorrect base case	Error: never ends
Use cases	Factorial, Fibonacci, tree traversal	Many algorithmic problems

Tiny Code

```
def sum_list(numbers):
    if not numbers:       # base case
        return 0
    return numbers[0] + sum_list(numbers[1:])  # recursive case

print(sum_list([1, 2, 3, 4]))   # 10
```

Why it Matters

Recursive functions let you write elegant, natural solutions to problems that involve repetition with smaller pieces—like mathematical sequences, hierarchical data, or divide-and-conquer algorithms.

Try It Yourself

1. Write a recursive function `countdown(n)` that prints numbers down to 0.
2. Create a recursive function `factorial(n)` and test it with `n=5`.
3. Write a recursive function `fib(n)` to compute Fibonacci numbers.
4. Challenge: Write a recursive function that calculates the sum of all numbers in a list.

40. Higher-Order Functions

A higher-order function is a function that either takes another function as an argument, returns a function, or both. This makes Python very powerful for writing flexible and reusable code.

Deep Dive

Functions as Arguments Since functions are objects in Python, you can pass them around like variables.

```
def apply_twice(func, x):
    return func(func(x))

def square(n):
    return n * n

print(apply_twice(square, 2))   # 16
```

Functions Returning Functions A function can also create and return another function.

```
def make_multiplier(n):
    def multiplier(x):
        return x * n
    return multiplier

double = make_multiplier(2)
print(double(5))   # 10
```

Built-in Higher-Order Functions Python provides many built-in higher-order functions:

- `map(func, iterable)` → applies a function to each item.

```
nums = [1, 2, 3]
squares = list(map(lambda x: x * x, nums))
print(squares)   # [1, 4, 9]
```

- `filter(func, iterable)` → keeps only items where the function returns `True`.

```
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)   # [2]
```

- `sorted(iterable, key=func)` → sorts by a custom key.

```
words = ["banana", "apple", "cherry"]
print(sorted(words, key=len))   # ['apple', 'banana', 'cherry']
```

- `reduce(func, iterable)` from `functools` → applies a rolling computation.

```
from functools import reduce
product = reduce(lambda a, b: a * b, [1, 2, 3, 4])
print(product)   # 24
```

Why Use Higher-Order Functions?

- They allow abstraction: write logic once and reuse it.
- They make code shorter and cleaner.
- They are the foundation of functional programming.

Quick Summary Table

Feature	Example	Purpose
Function as argument	`apply_twice(square, 2)`	Pass function in
Function as return value	`make_multiplier(3)`	Generate new function
`map()`	`map(lambda x:x+1, [1,2])`	Apply function to items
`filter()`	`filter(lambda x:x>2, [1,2,3])`	Keep items meeting condition
`sorted(..., key=func)`	`sorted(words, key=len)`	Custom sorting
`reduce()`	`reduce(lambda a,b:a*b, nums)`	Accumulate values

Tiny Code

```
def shout(text):
    return text.upper()

def whisper(text):
    return text.lower()

def speak(func, message):
    print(func(message))

speak(shout, "Hello")
speak(whisper, "Hello")
```

Why it Matters

Higher-order functions let you treat behavior as data. Instead of hardcoding actions, you can pass in functions to customize behavior. This leads to more flexible, reusable, and expressive programs.

Try It Yourself

1. Write a function `apply(func, values)` that applies `func` to every item in `values` (like your own `map`).
2. Use `filter()` with a lambda to keep only numbers greater than 10 from a list.
3. Write a `make_adder(n)` function that returns a new function adding `n` to its input.
4. Use `reduce()` to calculate the sum of a list of numbers.

Chapter 5. Modules and Packages

41. Importing Modules

A module in Python is a file containing Python code (functions, classes, variables) that you can reuse in other programs. Importing a module lets you use its code without rewriting it.

Deep Dive

Basic Import Use the `import` keyword followed by the module name:

```
import math

print(math.sqrt(16))   # 4.0
```

Here, `math` is a built-in module that provides mathematical functions.

Importing Multiple Modules You can import more than one module in one line:

```
import math, random

print(random.randint(1, 6))   # random number between 1 and 6
```

Accessing Module Contents To use something from a module, write `module_name.item`.

```
print(math.pi)      # 3.14159...
print(math.factorial(5))   # 120
```

Import Once Only A module is loaded once per program run, even if imported multiple times.

Where Python Looks for Modules

1. The current working directory.
2. Installed packages (like built-ins).
3. Paths defined in `sys.path`.

You can check where modules are loaded from:

```
import sys
print(sys.path)
```

Quick Summary Table

Statement	Meaning
`import math`	Import the whole module
`math.sqrt(25)`	Access function using `module.function`
`import a, b`	Import multiple modules at once
`sys.path`	Shows module search paths

Tiny Code

```
import math

radius = 3
area = math.pi * (radius  2)
print("Circle area:", area)
```

Why it Matters

Modules let you reuse existing solutions instead of reinventing the wheel. With imports, you can access thousands of built-in and third-party libraries that extend Python's power for math, networking, data science, and more.

Try It Yourself

1. Import the `math` module and calculate the square root of 49.
2. Import the `random` module and generate a random integer between 1 and 100.
3. Use `math.pi` to compute the area of a circle with radius 10.
4. Print out the list of paths from `sys.path` and check where Python looks for modules.

42. Built-in Modules (`math`, `random`)

Python comes with many built-in modules that provide ready-to-use functionality. Two of the most commonly used are `math` (for mathematical operations) and `random` (for random number generation).

Deep Dive

The `math` Module Provides advanced mathematical functions.

Commonly used functions and constants:

```
import math

print(math.sqrt(25))     # 5.0
print(math.pow(2, 3))    # 8.0
print(math.factorial(5)) # 120
print(math.pi)           # 3.141592653589793
print(math.e)            # 2.718281828459045
```

Other useful functions:

- `math.ceil(x)` → round up.
- `math.floor(x)` → round down.
- `math.log(x, base)` → logarithm.
- `math.sin(x)`, `math.cos(x)` → trigonometry.

The `random` Module Used for randomness in numbers, selections, and shuffling.

Examples:

```
import random

print(random.random())        # random float [0, 1)
print(random.randint(1, 6))   # random integer between 1 and 6
print(random.choice(["red", "blue", "green"]))  # random choice
```

Other useful functions:

- `random.shuffle(list)` → shuffle a list in place.
- `random.uniform(a, b)` → random float between `a` and `b`.
- `random.sample(population, k)` → pick `k` unique items.

Quick Summary Table

Module	Function	Example	Result
math	`math.sqrt(16)`	square root	`4.0`
math	`math.ceil(2.3)`	round up	`3`
math	`math.pi`	constant	`3.14159...`

Module	Function	Example	Result
random	`random.random()`	float 0–1	e.g. `0.732`
random	`random.randint(1,10)`	random int	between 1 and 10
random	`random.choice(seq)`	random element	one from list
random	`random.shuffle(seq)`	shuffle list	reorders in place

Tiny Code

```
import math, random

# math example
print("Cos(0):", math.cos(0))

# random example
colors = ["red", "green", "blue"]
random.shuffle(colors)
print("Shuffled colors:", colors)
```

Why it Matters

Built-in modules like `math` and `random` save you from writing code from scratch. They provide reliable, optimized tools for tasks you'll use frequently, from calculating areas to simulating dice rolls.

Try It Yourself

1. Use `math.factorial(6)` to calculate `6!`.
2. Generate a random float between 5 and 10 using `random.uniform()`.
3. Create a list of 5 numbers, shuffle it, and print the result.
4. Use `random.sample(range(1, 50), 6)` to simulate lottery numbers.

43. Aliasing Imports (`import ... as ...`)

Sometimes module names are long, or you want a shorter name for convenience. Python allows you to alias a module (or part of it) using `as`. This doesn't change the module—it just gives it a nickname in your code.

Deep Dive

Basic Aliasing

```
import math as m

print(m.sqrt(16))   # 4.0
print(m.pi)         # 3.14159...
```

Here, instead of typing `math` every time, you can use `m`.

Aliasing Specific Functions You can alias a single function too:

```
from math import factorial as fact

print(fact(5))   # 120
```

Common Conventions Some libraries have standard aliases that are widely used in the Python community:

- `import numpy as np`
- `import pandas as pd`
- `import matplotlib.pyplot as plt`

These conventions make code more readable because most developers recognize them instantly.

Why Use Aliases?

1. Shorter code → no need to write long names.
2. Avoid conflicts → if two modules have the same function name, aliasing prevents confusion.
3. Readability → follow community conventions.

Quick Summary Table

Statement	Meaning
`import module as alias`	give module a short name
`from module import f as alias`	give function a short name
`import numpy as np`	community standard alias

Tiny Code

```
import random as r

print(r.randint(1, 10))

from math import sqrt as root
print(root(81))   # 9.0
```

Why it Matters

Aliasing helps keep code neat, prevents naming conflicts, and improves readability—especially when using popular libraries with well-known abbreviations.

Try It Yourself

1. Import the `math` module as `m` and compute `m.sin(0)`.
2. Import `random.randint` as `dice` and use it to simulate rolling a dice.
3. Import `math.log` as `logarithm` and compute `logarithm(100, 10)`.
4. Think about why `import pandas as pd` is preferred in community codebases.

44. Importing Specific Functions

Instead of importing an entire module, you can import only the functions or variables you need. This makes code shorter and sometimes clearer.

Deep Dive

Basic Syntax

```
from math import sqrt, pi

print(sqrt(25))   # 5.0
print(pi)         # 3.14159...
```

Here, we can use `sqrt` and `pi` directly without prefixing them with `math.`.

Import with Aliases You can also alias imported items:

```
from math import factorial as fact

print(fact(5))   # 120
```

Importing Everything (Not Recommended) Using * imports all names from a module:

```
from math import *
print(sin(0))   # 0.0
```

This works, but it's discouraged because:

1. It clutters your namespace with too many names.
2. You might overwrite existing variables/functions by accident.

When to Import Specific Functions

- When you only need a small part of a large module.
- When you want shorter code without repeating the module name.
- When clarity matters more than knowing the source module.

Quick Summary Table

Statement	Meaning
`from math import sqrt`	Import only `sqrt`
`from math import sqrt, pi`	Import multiple names
`from math import factorial as f`	Import with alias
`from math import *`	Import all (not recommended)

Tiny Code

```
from random import choice, randint

colors = ["red", "green", "blue"]
print(choice(colors))       # random color
print(randint(1, 6))        # random number 1-6
```

Why it Matters

Importing specific functions makes code more concise and sometimes faster to read. It's especially useful when you're using only a few tools from a module instead of the whole thing.

Try It Yourself

1. Import only `sqrt` and `pow` from `math` and use them to calculate `sqrt(16)` and `2^5`.
2. Import `randint` from `random` and simulate rolling two dice.
3. Import `pi` from `math` and compute the circumference of a circle with radius 7.
4. Try using `from math import *`—then explain why this could cause confusion in larger programs.

45. `dir()` and `help()`

Python provides built-in functions like `dir()` and `help()` to let you explore modules, objects, and their available functionality. These are extremely useful when you're learning or working with unfamiliar code.

Deep Dive

`dir()` → List Attributes `dir(object)` returns a list of all attributes (functions, variables, classes) that an object has.

Example with a module:

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

This will show a list like:

```
['acos', 'asin', 'atan', 'ceil', 'cos', 'e', 'pi', 'sqrt', ...]
```

Example with a list:

```
nums = [1, 2, 3]
print(dir(nums))
```

This shows available list methods such as `append`, `extend`, `sort`.

`help()` → Documentation `help(object)` gives a detailed explanation, including docstrings, arguments, and usage.

Example with a module:

```
import random
help(random.randint)
```

This will display documentation:

```
randint(a, b)
    Return a random integer N such that a <= N <= b.
```

Combining Both

1. Use `dir()` to discover what functions exist.
2. Use `help()` to learn how a specific one works.

Quick Summary Table

Function	Purpose	Example
`dir(obj)`	Lists all attributes/methods	`dir(math)`
`help(obj)`	Shows documentation of an object	`help(str.upper)`

Tiny Code

```
import math

print("Attributes in math:", dir(math)[:5])   # show first 5 only
help(math.sqrt)   # show docstring for sqrt
```

Why it Matters

Instead of searching online every time, you can use `dir()` and `help()` inside Python itself. This makes learning, debugging, and exploring modules much faster.

Try It Yourself

1. Use `dir(str)` to see what methods strings have.
2. Pick one (like `.split`) and call `help(str.split)`.
3. Import the `random` module and run `dir(random)`—see how many functions it provides.
4. Use `help(random.choice)` to understand how it works.

46. Creating Your Own Module

A module is just a Python file that you can reuse in other programs. By creating your own module, you can organize code into separate files, making projects easier to maintain and share.

Deep Dive

Step 1: Write a Module Any `.py` file can act as a module. Example — create a file called `mymath.py`:

```
# mymath.py
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

Step 2: Import the Module In another Python file (or interactive shell):

```
import mymath

print(mymath.add(2, 3))       # 5
print(mymath.multiply(4, 5))  # 20
```

Step 3: Import Specific Functions

```
from mymath import add

print(add(10, 20))   # 30
```

Step 4: Module Location Python looks for modules in the current folder first, then in installed libraries (`sys.path`). If your module is in the same directory, you can import it directly.

Special Variable: `__name__` Inside every module, Python sets a special variable `__name__`.

- If the module is run directly: `__name__ == "__main__"`.
- If the module is imported: `__name__ == "module_name"`.

This lets you write code that runs only when the file is executed, not when it's imported.

```python
# mymath.py
def add(a, b):
    return a + b

if __name__ == "__main__":
    print("Testing add:", add(2, 3))
```

Quick Summary Table

Step	Example
Create file	`mymath.py`
Import whole module	`import mymath`
Import specific function	`from mymath import add`
Check module search path	`import sys; print(sys.path)`
Run directly check	`if __name__ == "__main__": ...`

Tiny Code

```python
# File: greetings.py
def hello(name):
    return f"Hello, {name}!"

# File: main.py
import greetings
print(greetings.hello("Alice"))
```

Why it Matters

Creating your own modules lets you structure larger projects, reuse code across different scripts, and share your work with others. It's the foundation for building Python packages and libraries.

Try It Yourself

1. Create a file `calculator.py` with functions `add`, `subtract`, `multiply`, and `divide`.
2. Import it in a separate file and test each function.
3. Add a test block using `if __name__ == "__main__":` that runs some examples when executed directly.

4. Create another module (e.g., `greetings.py`) and practice importing both in a single script.

47. Understanding Packages

A package is a way to organize related modules into a directory. Unlike a single module (a `.py` file), a package is a folder that contains an extra file called `__init__.py`. This tells Python to treat the folder as a package.

Deep Dive

Basic Structure

```
mypackage/
    __init__.py
    math_utils.py
    string_utils.py
```

- `__init__.py` → can be empty, or it can define what gets imported when the package is used.
- `math_utils.py` and `string_utils.py` → normal Python modules.

Importing from a Package

```
import mypackage.math_utils

print(mypackage.math_utils.add(2, 3))
```

Using `from ... import ...`

```
from mypackage import string_utils
print(string_utils.reverse("hello"))
```

Importing Functions Directly

```
from mypackage.math_utils import add
print(add(5, 6))
```

`__init__.py` Role If `__init__.py` includes imports, you can simplify usage:

```python
# mypackage/__init__.py
from .math_utils import add
from .string_utils import reverse
```

Now you can do:

```python
from mypackage import add, reverse
```

Nested Packages Packages can contain sub-packages:

```
mypackage/
    __init__.py
    utils/
        __init__.py
        file_utils.py
```

Access with:

```python
import mypackage.utils.file_utils
```

Quick Summary Table

Term	Meaning
Module	Single `.py` file
Package	Directory with `__init__.py` + modules
Sub-package	Package inside another package
Import	`import mypackage.module`
Simplify import	Define exports in `__init__.py`

Tiny Code

```
mypackage/
    __init__.py
    greetings.py
```

```python
# greetings.py
def hello(name):
    return f"Hello, {name}!"

# main.py
from mypackage import greetings
print(greetings.hello("Alice"))
```

Why it Matters

Packages make it easy to organize large projects into smaller, logical parts. They allow you to group related modules together, keep code clean, and make it reusable for others.

Try It Yourself

1. Create a folder `shapes/` with `__init__.py` and a module `circle.py` that has `area(r)`.
2. Import `circle` in another file and test the function.
3. Add another module `square.py` with `area(s)` and import both.
4. Modify `__init__.py` so you can do `from shapes import area` for both circle and square.

48. Using `pip` to Install Packages

While Python's standard library is powerful, you'll often need third-party packages. Python uses pip (Python Package Installer) to download and manage these packages from the Python Package Index (PyPI).

Deep Dive

Check if `pip` is Installed Most modern Python versions include it by default. You can check with:

```
pip --version
```

Installing a Package

```
pip install requests
```

This downloads and installs the popular `requests` library for making HTTP requests.

Using the Installed Package

```
import requests

response = requests.get("https://api.github.com")
print(response.status_code)   # 200
```

Upgrading a Package

```
pip install --upgrade requests
```

Uninstalling a Package

```
pip uninstall requests
```

Listing Installed Packages

```
pip list
```

Search for Packages

```
pip search numpy
```

Requirements File You can save dependencies in a file (`requirements.txt`) so others can install them easily:

```
requests==2.31.0
numpy>=1.25
```

Install everything at once:

```
pip install -r requirements.txt
```

Quick Summary Table

Command	Purpose
`pip install package`	Install a package

Command	Purpose
`pip install --upgrade package`	Update a package
`pip uninstall package`	Remove a package
`pip list`	Show installed packages
`pip freeze > requirements.txt`	Save current dependencies
`pip install -r requirements.txt`	Install from requirements file

Tiny Code

```
import numpy as np

arr = np.array([1, 2, 3])
print("Array:", arr)
```

Why it Matters

`pip` opens the door to Python's massive ecosystem. Whether you need data analysis (`pandas`), machine learning (`scikit-learn`), or web frameworks (`Flask`, `Django`), you can install them in seconds and start building.

Try It Yourself

1. Run `pip list` to see what's already installed.
2. Install the `requests` package and use it to fetch a webpage.
3. Install `pandas` and create a simple DataFrame.
4. Export your current environment with `pip freeze > requirements.txt` and share it with a friend.

49. Virtual Environments

A virtual environment is a self-contained directory that holds a specific Python version and its installed packages. It allows you to isolate dependencies for different projects so they don't conflict with each other.

Deep Dive

Why Virtual Environments?

- Different projects may need different versions of the same library.
- Prevents conflicts between global and project-specific packages.
- Keeps your system Python clean.

Creating a Virtual Environment Use the built-in `venv` module:

```
python -m venv myenv
```

This creates a folder `myenv/` with its own Python interpreter and libraries.

Activating the Environment

- On Windows:

```
myenv\Scripts\activate
```

- On Mac/Linux:

```
source myenv/bin/activate
```

You'll see `(myenv)` appear in your terminal prompt, showing it's active.

Installing Packages Inside Once activated, use `pip` normally—it only affects this environment:

```
pip install requests
```

Deactivating the Environment

```
deactivate
```

This returns you to the system Python.

Removing the Environment Just delete the folder `myenv/`—it's safe.

Quick Summary Table

Command	Purpose
`python -m venv myenv`	Create a virtual environment
`source myenv/bin/activate`	Activate (Mac/Linux)

Command	Purpose
`myenv\Scripts\activate`	Activate (Windows)
`pip install package`	Install inside environment
`deactivate`	Exit environment

Tiny Code

```
# Create and activate environment
python -m venv env_demo
source env_demo/bin/activate   # Linux/Mac

pip install numpy
python -c "import numpy; print(numpy.__version__)"
```

Why it Matters

Virtual environments are essential for professional Python development. They ensure each project has the right dependencies and prevent "it works on my machine" problems.

Try It Yourself

1. Create a new virtual environment called `project_env`.
2. Activate it and install `pandas`.
3. Verify by importing `pandas` in Python.
4. Deactivate, then delete the folder to remove the environment.

50. Popular Third-Party Packages (Overview)

Beyond the Python standard library, the community has built thousands of powerful third-party packages available through PyPI (Python Package Index). These extend Python's capabilities for web development, data analysis, machine learning, automation, and more.

Deep Dive

Web Development

- Flask → lightweight framework for web apps.
- Django → full-featured framework for large projects.

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return "Hello, Flask!"
```

Data Science & Analysis

- NumPy → arrays and fast math operations.
- Pandas → dataframes for data analysis.
- Matplotlib / Seaborn → visualization and charts.

```
import pandas as pd

data = {"Name": ["Alice", "Bob"], "Age": [25, 30]}
df = pd.DataFrame(data)
print(df)
```

Machine Learning & AI

- scikit-learn → machine learning algorithms.
- TensorFlow / PyTorch → deep learning libraries.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

Networking & APIs

- Requests → simple HTTP requests.
- FastAPI → modern web APIs with async support.

Automation & Scripting

- BeautifulSoup → web scraping.
- openpyxl → Excel file automation.

- schedule → lightweight task scheduler.

Why Use Third-Party Packages?

- Save time → no need to reinvent the wheel.
- Tested & optimized → reliable, community-supported.
- Ecosystem → Python's real power comes from these packages.

Quick Summary Table

Area	Popular Packages	Use Case
Web Development	Flask, Django, FastAPI	Build websites & APIs
Data Analysis	NumPy, Pandas, Matplotlib, Seaborn	Process & visualize data
Machine Learning	scikit-learn, TensorFlow, PyTorch	ML & deep learning
Automation	Requests, BeautifulSoup, openpyxl	HTTP, scraping, Excel automation

Tiny Code

```
import requests

response = requests.get("https://api.github.com")
print("Status:", response.status_code)
```

Why it Matters

Third-party packages are what make Python one of the most popular languages today. Whether you want to build websites, analyze data, or train AI models, there's a package ready to help you.

Try It Yourself

1. Use `pip install requests` and fetch data from any website.
2. Install `pandas` and create a small table of data.
3. Install `matplotlib` and draw a simple line chart.
4. Explore PyPI (https://pypi.org) and find a package that interests you.

Chapter 6. File Handling

51. Opening Files (`open`)

Working with files is a core part of programming. Python's built-in `open()` function lets you read from and write to files easily.

Deep Dive

Basic Syntax

```
file = open("example.txt", "mode")
```

- `"example.txt"` → the file name (with path if needed).
- `"mode"` → tells Python how to open the file.

Common modes:

- `"r"` → read (default).
- `"w"` → write (creates/overwrites file).
- `"a"` → append (adds to file).
- `"b"` → binary mode (e.g., images).
- `"r+"` → read and write.

Example: Opening for Reading

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

Example: Opening for Writing

```
file = open("new.txt", "w")
file.write("Hello, Python!\n")
file.close()
```

File Closing Always close files after use with `file.close()`.

- This frees system resources.
- Ensures data is written properly.

Error Handling If the file doesn't exist in `"r"` mode, Python raises an error:

```
open("missing.txt", "r")  # FileNotFoundError
```

Quick Summary Table

Mode	Meaning	Example
`"r"`	Read (default)	`open("f.txt", "r")`
`"w"`	Write (overwrite)	`open("f.txt", "w")`
`"a"`	Append	`open("f.txt", "a")`
`"b"`	Binary	`open("img.png", "rb")`
`"r+"`	Read + Write	`open("f.txt", "r+")`

Tiny Code

```
# Write a file
f = open("hello.txt", "w")
f.write("Hello, world!")
f.close()

# Read the file
f = open("hello.txt", "r")
print(f.read())
f.close()
```

Why it Matters

Files let you store information permanently. Whether saving logs, configurations, or datasets, file handling is essential for almost every real-world Python project.

Try It Yourself

1. Create a file `notes.txt` and write three lines of text into it.
2. Reopen the file in `"r"` mode and print the contents.
3. Open the same file in `"a"` mode and add another line.
4. Try opening a non-existent file in `"r"` mode and see the error.

52. Reading Files

Once you open a file in read mode, you can extract its contents in different ways depending on your needs: the whole file, line by line, or into a list.

Deep Dive

Read the Entire File

```
f = open("notes.txt", "r")
content = f.read()
print(content)
f.close()
```

- `f.read()` → returns the whole file as a single string.

Read One Line at a Time

```
f = open("notes.txt", "r")
line1 = f.readline()
line2 = f.readline()
print(line1, line2)
f.close()
```

- Each call to `readline()` gets the next line (including the `\n`).

Read All Lines into a List

```
f = open("notes.txt", "r")
lines = f.readlines()
print(lines)
f.close()
```

- `f.readlines()` returns a list where each element is one line.

Iterating Over a File The most common and memory-friendly way:

```
f = open("notes.txt", "r")
for line in f:
    print(line.strip())
f.close()
```

- This reads one line at a time, great for large files.

Quick Summary Table

Method	What it Does	Example
`f.read()`	Reads whole file as a string	`content = f.read()`
`f.readline()`	Reads the next line	`line = f.readline()`
`f.readlines()`	Reads all lines into a list	`lines = f.readlines()`
`for line in f`	Iterates line by line (efficient)	`for l in f: print(l)`

Tiny Code

```
with open("notes.txt", "r") as f:
    for line in f:
        print("Line:", line.strip())
```

Why it Matters

Reading files is fundamental to processing data. Whether you're analyzing logs, reading configurations, or loading datasets, understanding the different read methods helps you handle small and large files efficiently.

Try It Yourself

1. Write three lines into `data.txt`.
2. Read the entire file at once with `f.read()`.
3. Use `f.readline()` twice to print the first two lines separately.
4. Use a loop to print each line from the file without extra spaces.

53. Writing Files

Python lets you write text to files using the `write()` and `writelines()` methods. This is useful for saving logs, results, or any output that needs to be stored permanently.

Deep Dive

Write Text with `write()` Opening a file in `"w"` mode will overwrite it if it already exists, or create it if it doesn't.

```
f = open("output.txt", "w")
f.write("Hello, world!\n")
f.write("This is a new line.\n")
f.close()
```

Append Mode (`"a"`) To keep existing content and add to the end:

```
f = open("output.txt", "a")
f.write("Adding more text here.\n")
f.close()
```

Write Multiple Lines with `writelines()`

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]

f = open("multi.txt", "w")
f.writelines(lines)
f.close()
```

Note: `writelines()` does not add newlines automatically—you must include `\n` yourself.

Best Practice with `with` Automatically closes the file after writing:

```
with open("log.txt", "w") as f:
    f.write("Log entry 1\n")
    f.write("Log entry 2\n")
```

Quick Summary Table

Mode	Behavior	Example
`"w"`	Write (overwrite existing file)	`open("f.txt", "w")`
`"a"`	Append (keep existing, add more)	`open("f.txt", "a")`
`"x"`	Create (error if file exists)	`open("f.txt", "x")`

Tiny Code

```
with open("diary.txt", "w") as f:
    f.write("Day 1: Learned Python file writing.\n")
    f.write("Day 2: Feeling confident!\n")
```

Why it Matters

Being able to write files is crucial for persisting data beyond program execution. Logs, reports, exported data, and notes all rely on writing to files.

Try It Yourself

1. Create a file `journal.txt` and write three lines about your day.
2. Open the file again in `"a"` mode and add two more lines.
3. Use `writelines()` to add a list of tasks into `tasks.txt`.
4. Reopen and read back the contents to confirm everything was saved.

54. File Modes (`r`, `w`, `a`, `b`)

When opening files in Python with `open()`, the mode determines how the file is accessed—read, write, append, or binary. Understanding modes is essential to avoid overwriting or corrupting files.

Deep Dive

Text Modes (default)

- `"r"` → Read (default). File must exist.
- `"w"` → Write. Creates new file or overwrites existing.
- `"a"` → Append. Adds to the end, keeps existing content.
- `"x"` → Create. Errors if the file already exists.

```
open("notes.txt", "r")  # read
open("notes.txt", "w")  # write (erase contents!)
open("notes.txt", "a")  # append
open("newfile.txt", "x")# create only if not exists
```

Binary Modes Add `"b"` to handle non-text files (images, audio, executables).

- `"rb"` → read binary.
- `"wb"` → write binary.
- `"ab"` → append binary.

```
# Reading an image
with open("photo.jpg", "rb") as f:
    data = f.read()

# Writing binary
with open("copy.jpg", "wb") as f:
    f.write(data)
```

Combining Modes You can mix read/write with `"+"`:

- `"r+"` → read & write (file must exist).
- `"w+"` → write & read (overwrites or creates).
- `"a+"` → append & read.

```
with open("data.txt", "r+") as f:
    content = f.read()
    f.write("\nExtra line")
```

Quick Summary Table

Mode	Description	Notes
`"r"`	Read (default)	File must exist
`"w"`	Write	Overwrites file
`"a"`	Append	Adds at end of file
`"x"`	Create new	Error if file exists
`"b"`	Binary	Add to handle non-text data
`"r+"`	Read + Write	No overwrite, must exist
`"w+"`	Write + Read	Overwrites existing file
`"a+"`	Append + Read	File pointer at end

Tiny Code

```
# Write + read
with open("sample.txt", "w+") as f:
    f.write("Hello!\n")
    f.seek(0)
    print(f.read())
```

Why it Matters

Choosing the right mode ensures you don't lose data accidentally (like `"w"` erasing files) and allows you to correctly handle binary files like images or PDFs.

Try It Yourself

1. Open a file in `"w"` mode and write two lines. Reopen it in `"r"` mode and confirm old content was overwritten.
2. Open the same file in `"a"` mode and add another line.
3. Try using `"x"` mode to create a new file. Run it twice and observe the error on the second run.
4. Copy an image using `"rb"` and `"wb"`.

55. Closing Files

When you open a file in Python, the system allocates resources to manage it. To free these resources and ensure all data is written properly, you must close the file once you're done.

Deep Dive

Manual Closing with `close()`

```
f = open("notes.txt", "w")
f.write("Hello, file!")
f.close()
```

- `close()` ensures data is flushed from memory to disk.
- If you forget, data may not be saved properly.

Checking if a File is Closed

```
f = open("notes.txt", "r")
print(f.closed)   # False
f.close()
print(f.closed)   # True
```

Best Practice: `with` Statement Instead of manually calling `close()`, use `with`. It automatically closes the file, even if an error occurs.

```
with open("notes.txt", "r") as f:
    content = f.read()
print(f.closed)   # True
```

Flushing Without Closing If you want to save changes but keep the file open:

```
f = open("data.txt", "w")
f.write("Line 1\n")
f.flush()     # forces write to disk
# file still open
f.close()
```

What Happens if You Don't Close?

- Data might not be saved (especially in write mode).
- Too many open files can exhaust system resources.
- On some systems, files stay locked until closed.

Quick Summary Table

Method	Behavior
`f.close()`	Manually closes the file
`f.closed`	Check if file is closed
`f.flush()`	Force save data without closing
`with open()`	Automatically closes after block

Tiny Code

```
with open("log.txt", "w") as f:
    f.write("Session started.\n")

print("Closed?", f.closed)  # True
```

Why it Matters

Closing files ensures data safety and efficient resource usage. Forgetting to close files can lead to bugs, data loss, or locked files. The `with` statement makes it almost impossible to forget.

Try It Yourself

1. Open a file in write mode, write some text, and check `f.closed` before and after calling `close()`.
2. Use `with open()` to write two lines and verify that the file is closed outside the block.
3. Experiment with `f.flush()`—write text, flush, then write more before closing.
4. Try opening many files in a loop without closing them, then observe system warnings/errors.

56. Using `with` Context Manager

The `with` statement in Python provides a clean and safe way to work with files. It automatically takes care of opening and closing the file, even if errors occur while processing.

Deep Dive

Basic Usage

```
with open("notes.txt", "r") as f:
    content = f.read()
print("File closed?", f.closed)  # True
```

- The file is automatically closed after the `with` block.
- You don't need to call `f.close()` manually.

Writing with `with`

```
with open("output.txt", "w") as f:
    f.write("Hello, Python!\n")
    f.write("Writing with context manager.\n")
```

The file is saved and closed as soon as the block ends.

Why Use `with`?

1. Ensures proper cleanup (file is closed automatically).
2. Handles exceptions safely.
3. Makes code cleaner and shorter.

Multiple Files with One `with` You can work with multiple files in a single `with` statement:

```
with open("input.txt", "r") as infile, open("copy.txt", "w") as outfile:
    for line in infile:
        outfile.write(line)
```

Custom Context Managers The `with` statement isn't just for files—it works with anything that supports the context manager protocol (`__enter__` and `__exit__`).

Example:

```
class MyResource:
    def __enter__(self):
        print("Resource acquired")
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        print("Resource released")

with MyResource():
    print("Using resource")
```

Quick Summary Table

Feature	Example
Auto-close file	`with open("f.txt") as f:`
Write file	`with open("f.txt","w") as f: f.write("x")`
Multiple files	`with open("a.txt") as a, open("b.txt") as b:`
Custom manager	Define `__enter__`, `__exit__`

Tiny Code

```
with open("data.txt", "w") as f:
    f.write("Line 1\n")
    f.write("Line 2\n")

print("Closed?", f.closed)  # True
```

Why it Matters

The `with` statement is the best practice for file handling in Python. It makes code safer, shorter, and more reliable by guaranteeing cleanup.

Try It Yourself

1. Use `with open("log.txt", "w")` to write three lines. Confirm the file is closed afterwards.
2. Copy the contents of one file into another using a `with` block.
3. Experiment by raising an error inside a `with` block—notice the file is still closed.
4. Create a simple class with `__enter__` and `__exit__` to practice writing your own context manager.

57. Working with CSV Files

CSV (Comma-Separated Values) files are widely used for storing tabular data like spreadsheets or databases. Python's built-in `csv` module makes it easy to read and write CSV files.

Deep Dive

Reading a CSV File

```
import csv

with open("data.csv", "r") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

- `csv.reader` → reads file line by line, splitting values by commas.
- Each row is returned as a list of strings.

Writing to a CSV File

```
import csv

rows = [
    ["Name", "Age"],
    ["Alice", 25],
    ["Bob", 30]
]

with open("people.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(rows)
```

- `writerow()` → writes a single row.
- `writerows()` → writes multiple rows.
- `newline=""` avoids blank lines on Windows.

Using Dictionaries with CSV Instead of working with lists, you can use DictReader and DictWriter.

```
import csv

# Writing
with open("people.csv", "w", newline="") as f:
    fieldnames = ["Name", "Age"]
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({"Name": "Charlie", "Age": 35})

# Reading
with open("people.csv", "r") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row["Name"], row["Age"])
```

Quick Summary Table

Class/Function	Purpose
`csv.reader`	Reads CSV into lists
`csv.writer`	Writes CSV from lists
`csv.DictReader`	Reads CSV into dictionaries
`csv.DictWriter`	Writes CSV from dictionaries

Tiny Code

```
import csv

with open("scores.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Name", "Score"])
    writer.writerow(["Alice", 90])
    writer.writerow(["Bob", 85])
```

```
with open("scores.csv", "r") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Why it Matters

CSV is the most common format for sharing data between systems. By mastering the `csv` module, you can process spreadsheets, export reports, and integrate with databases or analytics tools.

Try It Yourself

1. Create a file `students.csv` with three rows (`Name, Age`).
2. Write Python code to read and print all rows.
3. Use `DictWriter` to add a new student to the file.
4. Use `DictReader` to print only the `Name` column.

58. Working with JSON Files

JSON (JavaScript Object Notation) is a lightweight data format often used for APIs, configs, and data exchange. Python has a built-in `json` module that makes it easy to read and write JSON files.

Deep Dive

Importing the Module

```
import json
```

Writing JSON to a File

```
import json

data = {
    "name": "Alice",
    "age": 25,
    "languages": ["Python", "JavaScript"]
}
```

```
with open("data.json", "w") as f:
    json.dump(data, f)
```

- `json.dump(obj, file)` → saves Python object as JSON.
- Automatically converts dicts, lists, strings, numbers, booleans.

Reading JSON from a File

```
with open("data.json", "r") as f:
    loaded = json.load(f)

print(loaded["name"])   # Alice
print(loaded["languages"])  # ['Python', 'JavaScript']
```

Convert Between JSON and String

- `json.dumps(obj)` → convert Python object → JSON string.
- `json.loads(str)` → convert JSON string → Python object.

```
s = json.dumps(data)
print(s)   # '{"name": "Alice", "age": 25, ...}'

obj = json.loads(s)
print(obj["age"])   # 25
```

Pretty Printing JSON

```
print(json.dumps(data, indent=4))
```

Quick Summary Table

Function	Purpose
`json.dump(obj,f)`	Write JSON to a file
`json.load(f)`	Read JSON from a file
`json.dumps(obj)`	Convert object to JSON string
`json.loads(str)`	Convert JSON string to Python object

Tiny Code

```python
import json

user = {"id": 1, "active": True, "roles": ["admin", "editor"]}

with open("user.json", "w") as f:
    json.dump(user, f, indent=2)

with open("user.json", "r") as f:
    print(json.load(f))
```

Why it Matters

JSON is the universal format for modern applications—from web APIs to configuration files. By mastering Python's `json` module, you can easily communicate with APIs, save structured data, and exchange information with other systems.

Try It Yourself

1. Create a dictionary with your name, age, and hobbies, then save it to `me.json`.
2. Reopen `me.json` and print the hobbies.
3. Use `json.dumps()` to print the same dictionary as a formatted JSON string.
4. Convert a JSON string back into a Python dictionary using `json.loads()`.

59. File Exceptions

When working with files, many things can go wrong: the file might not exist, permissions might be missing, or the disk might be full. Python uses exceptions to handle these errors safely.

Deep Dive

Common File Exceptions

- `FileNotFoundError` → trying to open a non-existent file.
- `PermissionError` → trying to open/write without permission.
- `IsADirectoryError` → opening a directory instead of a file.
- `IOError` / `OSError` → general input/output errors (disk, encoding).

Handling File Exceptions

```
try:
    f = open("missing.txt", "r")
    content = f.read()
    f.close()
except FileNotFoundError:
    print("The file does not exist.")
```

Catching Multiple Exceptions

```
try:
    f = open("/protected/data.txt", "r")
except (FileNotFoundError, PermissionError) as e:
    print("Error:", e)
```

Using `finally` for Cleanup

```
try:
    f = open("data.txt", "r")
    print(f.read())
finally:
    f.close()   # ensures file closes even on error
```

Safer with `with` The `with` statement avoids many of these issues automatically, but exceptions can still happen when opening:

```
try:
    with open("notes.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    print("File not found!")
```

Quick Summary Table

Exception	Cause
`FileNotFoundError`	File does not exist
`PermissionError`	No permission to access file
`IsADirectoryError`	Tried to open a directory as a file
`IOError` / `OSError`	General input/output failure

Tiny Code

```
filename = "example.txt"

try:
    with open(filename, "r") as f:
        print(f.read())
except FileNotFoundError:
    print(f"Error: {filename} was not found.")
```

Why it Matters

Errors in file handling are inevitable. Exception handling makes your programs robust, user-friendly, and prevents crashes when dealing with unpredictable files and systems.

Try It Yourself

1. Try opening a file that doesn't exist, catch the `FileNotFoundError`, and print a custom message.
2. Write code that catches both `FileNotFoundError` and `PermissionError`.
3. Use `finally` to always print `"Done"` after attempting to open a file.
4. Combine `with open()` and `try...except` to safely read a file only if it exists.

60. Paths & Directories (`os`, `pathlib`)

Working with files often means dealing with paths and directories. Python provides two main tools for this: the older `os` module and the modern `pathlib` module.

Deep Dive

Getting Current Working Directory

```
import os
print(os.getcwd())   # shows current directory
```

With `pathlib`:

```
from pathlib import Path
print(Path.cwd())
```

Changing Directory

```
os.chdir("/tmp")
```

Listing Files in a Directory

```
print(os.listdir("."))   # list all files/folders
```

With `pathlib`:

```
p = Path(".")
for file in p.iterdir():
    print(file)
```

Joining Paths Instead of manually adding slashes, use:

```
os.path.join("folder", "file.txt")   # "folder/file.txt"
```

With `pathlib`:

```
Path("folder") / "file.txt"
```

Checking File/Folder Existence

```
os.path.exists("notes.txt")   # True/False
```

With `pathlib`:

```
p = Path("notes.txt")
print(p.exists())
print(p.is_file())
print(p.is_dir())
```

Creating Directories

```
os.mkdir("newfolder")
```

With parents:

```
Path("a/b/c").mkdir(parents=True, exist_ok=True)
```

Removing Files and Folders

```
os.remove("file.txt")      # delete file
os.rmdir("empty_folder")   # remove empty folder
```

With `pathlib`:

```
Path("file.txt").unlink()
```

Quick Summary Table

Action	`os` Example	`pathlib` Example
Current dir	`os.getcwd()`	`Path.cwd()`
List dir	`os.listdir(".")`	`Path(".").iterdir()`
Join paths	`os.path.join("a","b")`	`Path("a") / "b"`
Exists?	`os.path.exists("f.txt")`	`Path("f.txt").exists()`
Make dir	`os.mkdir("new")`	`Path("new").mkdir()`
Remove file	`os.remove("f.txt")`	`Path("f.txt").unlink()`

Tiny Code

```
from pathlib import Path

p = Path("demo_folder")
p.mkdir(exist_ok=True)

file = p / "hello.txt"
file.write_text("Hello, pathlib!")

print(file.read_text())
```

Why it Matters

Paths and directories are essential for any project involving files. `pathlib` provides a modern, object-oriented approach, while `os` ensures backward compatibility with older code. Knowing both makes you flexible.

Try It Yourself

1. Print your current working directory with both `os` and `pathlib`.
2. Create a folder called `projects` and inside it, a file `readme.txt` with some text.
3. List all files inside `projects`.
4. Write a script that checks if `archive/` exists, and if not, creates it.

Chapter 7. Object-Oriented Python

61. Classes & Objects

Python is an object-oriented programming (OOP) language. A class is like a blueprint for creating objects, and an object is an instance of that class. Classes define the structure (attributes) and behavior (methods) of objects.

Deep Dive

Defining a Class

```
class Person:
    pass
```

This defines a new class called `Person`.

Creating an Object (Instance)

```
p1 = Person()
print(type(p1))   # <class '__main__.Person'>
```

Here, `p1` is an object of type `Person`.

Adding Attributes

```
class Person:
    def __init__(self, name, age):
        self.name = name    # attribute
        self.age = age

p1 = Person("Alice", 25)
print(p1.name, p1.age)   # Alice 25
```

- `__init__` → constructor method, runs when creating an object.
- `self` → refers to the current object.

Adding Methods

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, my name is {self.name}."

p1 = Person("Bob")
print(p1.greet())   # Hello, my name is Bob.
```

A method is just a function inside a class that operates on its objects.

Quick Summary Table

Concept	Definition	Example
Class	Blueprint for objects	`class Car: ...`
Object	Instance of a class	`c1 = Car()`
Attributes	Data stored in objects	`self.name`, `self.age`
Methods	Functions inside a class	`def drive(self): ...`
`__init__`	Constructor, called when object is created	`def __init__(...)`
`self`	Refers to the current instance	`self.name = name`

Tiny Code

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
```

```
        self.breed = breed

    def bark(self):
        return f"{self.name} says Woof!"

d1 = Dog("Max", "Labrador")
print(d1.bark())
```

Why it Matters

Classes and objects are the foundation of OOP. They let you model real-world things (like cars, users, or bank accounts) in code, organize functionality, and build scalable applications.

Try It Yourself

1. Create a `Car` class with attributes `brand` and `year`.
2. Add a method `drive()` that prints `"The car is driving"`.
3. Make two different `Car` objects and call their `drive()` method.
4. Add another method that prints the car's brand and year.

62. Attributes & Methods

In Python classes, attributes are variables that belong to objects, and methods are functions that belong to objects. Together, they define what an object has (data) and what it does (behavior).

Deep Dive

Attributes (Object Data) Attributes store information about an object.

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

c1 = Car("Toyota", 2020)
print(c1.brand)   # Toyota
print(c1.year)    # 2020
```

Here, `brand` and `year` are attributes of the `Car` object.

Instance Methods (Object Behavior) Methods define actions an object can perform.

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def drive(self):
        return f"{self.brand} is driving."

c1 = Car("Honda", 2019)
print(c1.drive())   # Honda is driving.
```

- `self` allows the method to access the object's attributes.

Updating Attributes Attributes can be changed dynamically:

```
c1.year = 2022
print(c1.year)   # 2022
```

Adding New Attributes at Runtime

```
c1.color = "red"
print(c1.color)   # red
```

(But it's better to define attributes in `__init__` for consistency.)

Class Attributes vs Instance Attributes

- Instance attribute → unique to each object.
- Class attribute → shared by all objects of the class.

```
class Dog:
    species = "Canis lupus familiaris"   # class attribute
    def __init__(self, name):
        self.name = name                 # instance attribute

d1 = Dog("Buddy")
d2 = Dog("Charlie")
print(d1.species, d2.species)   # same for all
print(d1.name, d2.name)         # unique per dog
```

Quick Summary Table

Term	Meaning	Example
Instance attribute	Data unique to each object	`self.brand`, `self.year`
Class attribute	Shared across all objects	`species = ...`
Method	Function inside a class	`def drive(self)`
`self`	Refers to the current object instance	`self.name = name`

Tiny Code

```
class Student:
    school = "Python Academy"   # class attribute

    def __init__(self, name, grade):
        self.name = name
        self.grade = grade      # instance attribute

    def introduce(self):
        return f"I am {self.name}, grade {self.grade}."

s1 = Student("Alice", "A")
s2 = Student("Bob", "B")

print(s1.introduce())
print(s2.introduce())
print("School:", s1.school)
```

Why it Matters

Attributes and methods are the building blocks of object-oriented programming. Attributes give objects state, while methods give them behavior. Together, they let you model real-world entities in code.

Try It Yourself

1. Define a `Book` class with attributes `title` and `author`.
2. Add a method `describe()` that prints `"Title by Author"`.
3. Create two `Book` objects with different details and call `describe()` on both.
4. Add a class attribute `library = "City Library"` and print it from both objects.

63. `__init__` Constructor

In Python, the `__init__` method is a special method that runs automatically when you create a new object. It's often called the constructor because it initializes (sets up) the object's attributes.

Deep Dive

Basic Example

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("Alice", 25)
print(p1.name, p1.age)   # Alice 25
```

- `__init__` is called right after an object is created.
- `self` refers to the new object being initialized.

Default Values You can give parameters default values:

```
class Person:
    def __init__(self, name="Unknown", age=0):
        self.name = name
        self.age = age

p1 = Person()
print(p1.name, p1.age)   # Unknown 0
```

Constructor with Logic You can add checks or calculations during initialization:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height   # auto-calculate

r = Rectangle(4, 5)
print(r.area)   # 20
```

Multiple Objects, Independent Attributes Each object gets its own copy of instance attributes:

```
p1 = Person("Alice", 25)
p2 = Person("Bob", 30)

print(p1.name)   # Alice
print(p2.name)   # Bob
```

Quick Summary Table

Feature	Example	Purpose
Define init	`def __init__(self, ...):`	Runs on object creation
Assign values	`self.attr = value`	Stores attributes in object
Defaults	`def __init__(self, x=0)`	Optional parameters
With logic	Compute or validate values	Setup object cleanly

Tiny Code

```
class Dog:
    def __init__(self, name, breed="Unknown"):
        self.name = name
        self.breed = breed

d1 = Dog("Max", "Beagle")
d2 = Dog("Charlie")

print(d1.name, d1.breed)
print(d2.name, d2.breed)
```

Why it Matters

The `__init__` constructor ensures every object starts in a well-defined state. Without it, you'd have to manually assign attributes after creating objects, which is error-prone and messy.

Try It Yourself

1. Create a `Car` class with attributes `brand`, `model`, and `year` set in `__init__`.
2. Add a method `info()` that prints `"Brand Model (Year)"`.

3. Give `year` a default value if not provided.
4. Create two `Car` objects—one with all values, one with just brand and model—and call `info()` on both.

64. Instance vs Class Variables

In Python classes, variables can belong either to a specific object (instance variables) or to the class itself (class variables). Knowing the difference is key to writing predictable, reusable code.

Deep Dive

Instance Variables

- Defined inside `__init__` using `self`.
- Each object gets its own copy.

```
class Dog:
    def __init__(self, name):
        self.name = name    # instance variable

d1 = Dog("Buddy")
d2 = Dog("Charlie")

print(d1.name)   # Buddy
print(d2.name)   # Charlie
```

Each dog has its own `name`.

Class Variables

- Shared across all objects of the class.
- Defined directly inside the class, outside methods.

```
class Dog:
    species = "Canis lupus familiaris"   # class variable

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

d1 = Dog("Buddy")
d2 = Dog("Charlie")
```

```
print(d1.species)   # Canis lupus familiaris
print(d2.species)   # Canis lupus familiaris
```

Changing it affects all instances:

```
Dog.species = "Dog"
print(d1.species, d2.species)  # Dog Dog
```

Overriding Class Variables per Instance You can assign a new value to a class variable on a specific object, but then it becomes an instance variable for that object only:

```
d1.species = "Wolf"   # overrides for d1 only
print(d1.species)     # Wolf
print(d2.species)     # Dog
```

Quick Summary Table

Variable Type	Defined Where	Belongs To	Example
Instance	Inside `__init__` via `self`	Each object	`self.name = name`
Class	Inside class body	The class	`species = "Dog"`

Tiny Code

```
class Student:
    school = "Python Academy"   # class variable

    def __init__(self, name):
        self.name = name        # instance variable

s1 = Student("Alice")
s2 = Student("Bob")

print(s1.name, "-", s1.school)
print(s2.name, "-", s2.school)

Student.school = "Code Academy"
print(s1.school, s2.school)
```

Why it Matters

- Use instance variables for data unique to each object.
- Use class variables for properties shared across all objects. Mixing them up can cause bugs, so it's important to understand the difference.

Try It Yourself

1. Create a `Car` class with a class variable `wheels = 4`.
2. Add an instance variable `brand` inside `__init__`.
3. Make two cars with different brands, and confirm they both show 4 wheels.
4. Change `Car.wheels = 6` and check how it affects both objects.

65. Inheritance Basics

Inheritance allows one class to take on the attributes and methods of another. This promotes code reuse and models real-world relationships (e.g., a `Dog` is an `Animal`).

Deep Dive

Parent and Child Classes

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):   # Dog inherits from Animal
    def bark(self):
        return f"{self.name} says Woof!"
```

```
a = Animal("Generic")
print(a.speak())     # Generic makes a sound.

d = Dog("Buddy")
print(d.speak())     # Buddy makes a sound. (inherited)
print(d.bark())      # Buddy says Woof! (own method)
```

The `super()` Function `super()` lets the child class call methods from the parent class.

```
class Animal:
    def __init__(self, name):
        self.name = name

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name)   # call parent constructor
        self.color = color
```

```
c = Cat("Luna", "Gray")
print(c.name, c.color)   # Luna Gray
```

Overriding Methods A child can redefine methods from the parent:

```
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"

print(Dog().speak())   # Woof!
```

Inheritance Hierarchy

- A class can inherit from another class.
- You can create chains (e.g., `A → B → C`).
- Python supports multiple inheritance (covered later).

Quick Summary Table

Concept	Meaning	Example
Parent class	Base class being inherited from	`class Animal:`
Child class	Derived class that inherits from parent	`class Dog(Animal):`
Inheritance	Child gets parent's attributes/methods	`Dog` uses `speak()`
`super()`	Call parent methods inside child	`super().__init__(...)`
Overriding	Redefining a parent method in the child	`def speak(self): ...`

Tiny Code

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand
    def drive(self):
        return f"{self.brand} is moving."

class Car(Vehicle):
    def drive(self):
        return f"{self.brand} is driving on the road."

v = Vehicle("Generic Vehicle")
c = Car("Toyota")

print(v.drive())
print(c.drive())
```

Why it Matters

Inheritance reduces duplication and makes code more organized. By building hierarchies, you can model relationships between classes naturally, reusing and extending existing functionality.

Try It Yourself

1. Create a base class `Shape` with a method `area()` that returns 0.
2. Make a child class `Circle` that overrides `area()` to compute r^2.
3. Create a class `Square` that overrides `area()` to compute $side^2$.
4. Use `super().__init__()` to pass shared attributes from parent to child.

66. Method Overriding

Method overriding happens when a child class defines a method with the same name as one in its parent class. The child's version replaces (overrides) the parent's when called on a child object.

Deep Dive

Basic Example

```
class Animal:
    def speak(self):
        return "Some generic sound"

class Dog(Animal):
    def speak(self):   # overrides parent method
        return "Woof!"

a = Animal()
d = Dog()

print(a.speak())   # Some generic sound
print(d.speak())   # Woof!
```

Why Override?

- To provide specialized behavior in a child class.
- Keeps shared structure in the parent but allows customization.

Using `super()` with Overrides You can call the parent's version inside the override:

```
class Vehicle:
    def drive(self):
        return "The vehicle is moving."

class Car(Vehicle):
    def drive(self):
        parent_drive = super().drive()
        return parent_drive + " Specifically, the car is driving."

c = Car()
print(c.drive())
```

Partial Overrides You don't always have to replace the entire method—you can extend it:

```
class Logger:
    def log(self, message):
        print("Log:", message)
```

```
class TimestampLogger(Logger):
    def log(self, message):
        import datetime
        time = datetime.datetime.now()
        super().log(f"{time} - {message}")
```

Quick Summary Table

Concept	Meaning	Example
Overriding	Redefine method in child class	`Dog.speak()` replaces `Animal.speak()`
Specialized	Child provides its own implementation	`Car.drive()` different from `Vehicle.drive()`
`super()` use	Call parent version inside child	`super().log(...)`

Tiny Code

```
class Employee:
    def work(self):
        return "Employee is working."

class Manager(Employee):
    def work(self):
        return "Manager is planning and managing."

e = Employee()
m = Manager()

print(e.work())   # Employee is working.
print(m.work())   # Manager is planning and managing.
```

Why it Matters

Method overriding lets subclasses adapt behavior without rewriting everything from scratch. It's a cornerstone of polymorphism, where different classes can define the same method name but act differently.

Try It Yourself

1. Create a base class `Animal` with `sound()` that returns `"Unknown sound"`.
2. Make `Dog` and `Cat` subclasses that override `sound()` with `"Woof"` and `"Meow"`.
3. Use a loop to call `sound()` on both objects and see polymorphism in action.
4. Extend the base method in one subclass using `super()` to add extra behavior.

67. Multiple Inheritance

Python allows a class to inherit from more than one parent class. This is called multiple inheritance. It can be powerful but must be used carefully to avoid confusion.

Deep Dive

Basic Example

```
class Flyer:
    def fly(self):
        return "I can fly!"

class Swimmer:
    def swim(self):
        return "I can swim!"

class Duck(Flyer, Swimmer):   # inherits from both
    pass

d = Duck()
print(d.fly())   # I can fly!
print(d.swim())  # I can swim!
```

Here, `Duck` inherits methods from both `Flyer` and `Swimmer`.

The Diamond Problem & MRO If multiple parents have methods with the same name, Python uses the Method Resolution Order (MRO) to decide which one to call.

```
class A:
    def hello(self):
        return "Hello from A"

class B(A):
```

```
    def hello(self):
        return "Hello from B"

class C(A):
    def hello(self):
        return "Hello from C"

class D(B, C):
    pass

d = D()
print(d.hello())        # Hello from B
print(D.mro())          # [D, B, C, A, object]
```

- Python searches left to right in the inheritance list (`B` before `C`).
- `mro()` shows the order.

Using `super()` with Multiple Inheritance `super()` respects the MRO, allowing cooperative behavior:

```
class A:
    def action(self):
        print("A action")

class B(A):
    def action(self):
        super().action()
        print("B action")

class C(A):
    def action(self):
        super().action()
        print("C action")

class D(B, C):
    def action(self):
        super().action()
        print("D action")

d = D()
d.action()
```

Output:

```
A action
C action
B action
D action
```

Quick Summary Table

Concept	Meaning
Multiple inheritance	Class inherits from more than one parent
MRO	Defines search order for methods/attributes
Diamond problem	Ambiguity when same method exists in parents
`super()` in MRO	Ensures cooperative method calls

Tiny Code

```
class Writer:
    def write(self):
        return "Writing..."

class Reader:
    def read(self):
        return "Reading..."

class Author(Writer, Reader):
    pass

a = Author()
print(a.write())
print(a.read())
```

Why it Matters

Multiple inheritance allows you to combine behaviors from different classes, making code flexible and modular. But without understanding MRO, it can introduce bugs and unexpected results.

Try It Yourself

1. Create two classes `Walker` and `Runner`, each with a method.
2. Create a class `Athlete` that inherits from both and test all methods.
3. Add the same method `train()` in both parents and see which one `Athlete` uses.
4. Use `ClassName.mro()` to confirm the method resolution order.

68. Encapsulation & Private Members

Encapsulation is the principle of restricting direct access to some parts of an object, protecting its internal state. In Python, this is done through naming conventions rather than strict enforcement.

Deep Dive

Public Members

- Accessible from anywhere.
- Default in Python.

```
class Person:
    def __init__(self, name):
        self.name = name   # public attribute

p = Person("Alice")
print(p.name)   # Alice
```

Protected Members (`_var`)

- Indicated with a single underscore.
- Treated as "internal use only", but still accessible.

```
class Person:
    def __init__(self, name):
        self._secret = "hidden"

p = Person("Alice")
print(p._secret)   # possible, but discouraged
```

Private Members (`__var`)

- Indicated with double underscores.
- Name-mangled to prevent accidental access.

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance   # private

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

acc = BankAccount(100)
acc.deposit(50)
print(acc.get_balance())   # 150
```

Trying to access directly:

```
print(acc.__balance)   # AttributeError
print(acc._BankAccount__balance)   # works (name-mangled)
```

Why Encapsulation?

1. Prevent accidental modification of sensitive data.
2. Provide controlled access via methods (getters/setters).
3. Separate internal logic from public API.

Quick Summary Table

Convention	Syntax	Access Level
Public	`var`	Free to access
Protected	`_var`	Internal use only
Private	`__var`	Strongly restricted (name-mangled)

Tiny Code

```
class Student:
    def __init__(self, name, grade):
        self.name = name            # public
        self._grade = grade         # protected
        self.__id = 12345           # private

    def get_id(self):
        return self.__id

s = Student("Bob", "A")
print(s.name)       # Public
print(s._grade)     # Accessible but discouraged
print(s.get_id())   # Safe access
```

Why it Matters

Encapsulation protects the integrity of your objects. By controlling access, you reduce bugs and make your code safer and more maintainable.

Try It Yourself

1. Create a `BankAccount` class with a private `__balance`.
2. Add `deposit()` and `withdraw()` methods that safely modify it.
3. Add a method `get_balance()` to return the balance.
4. Try accessing `__balance` directly and observe the error.

69. Special Methods (`__str__`, `__len__`, etc.)

Python classes can define special methods (also called *dunder methods*, because they have double underscores). These let objects behave like built-in types and integrate smoothly with Python features.

Deep Dive

`__str__` → String Representation Defines what `print(obj)` shows.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return f"{self.name}, {self.age} years old"

p = Person("Alice", 25)
print(p)   # Alice, 25 years old
```

`__repr__` → Developer-Friendly Representation Used in debugging and interactive shells.

```python
class Person:
    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"
```

`__len__` → Length Lets your object work with `len(obj)`.

```python
class Team:
    def __init__(self, members):
        self.members = members
    def __len__(self):
        return len(self.members)

t = Team(["Alice", "Bob"])
print(len(t))   # 2
```

`__getitem__` and `__setitem__` → Indexing Make objects behave like lists/dicts.

```python
class Notebook:
    def __init__(self):
        self.notes = {}
    def __getitem__(self, key):
        return self.notes[key]
    def __setitem__(self, key, value):
        self.notes[key] = value

n = Notebook()
n["day1"] = "Learn Python"
print(n["day1"])   # Learn Python
```

Other Useful Special Methods

- `__eq__` → equality (`==`)
- `__lt__` → less than (`<`)
- `__add__` → addition (`+`)
- `__call__` → make object callable like a function
- `__iter__` → make object iterable in `for` loops

Quick Summary Table

Method	Purpose	Example Use
`__str__`	User-friendly string	`print(obj)`
`__repr__`	Debug/developer string	`obj` in console
`__len__`	Length	`len(obj)`
`__getitem__`	Indexing	`obj[key]`
`__setitem__`	Assigning by key	`obj[key] = value`
`__eq__`	Equality check	`obj1 == obj2`
`__add__`	Addition	`obj1 + obj2`
`__call__`	Callable object	`obj()`

Tiny Code

```
class Counter:
    def __init__(self, count=0):
        self.count = count

    def __str__(self):
        return f"Counter({self.count})"

    def __add__(self, other):
        return Counter(self.count + other.count)

c1 = Counter(3)
c2 = Counter(7)
print(c1)           # Counter(3)
print(c1 + c2)      # Counter(10)
```

Why it Matters

Special methods let you design objects that feel natural to use, just like built-in types. This makes your classes more powerful, expressive, and Pythonic.

Try It Yourself

1. Create a `Book` class with `title` and `author`, and override `__str__` to print `"Title by Author"`.
2. Add `__len__` to return the length of the title.
3. Implement `__eq__` to compare two books by title and author.
4. Implement `__add__` so that adding two books returns a string joining both titles.

70. Static & Class Methods

In Python, not all methods need to work with a specific object. Sometimes they belong to the class itself. Python provides class methods and static methods for these cases.

Deep Dive

Instance Method (Default)

- The usual method, works with an instance.
- First parameter is always `self`.

```
class Person:
    def greet(self):
        return "Hello!"
```

Class Method (`@classmethod`)

- Works with the class, not an individual object.
- First parameter is `cls` (the class).
- Declared with `@classmethod` decorator.

```
class Person:
    species = "Homo sapiens"

    @classmethod
    def get_species(cls):
        return cls.species

print(Person.get_species())  # Homo sapiens
```

Static Method (`@staticmethod`)

- Does not use `self` or `cls`.

- A regular function inside a class for logical grouping.
- Declared with `@staticmethod`.

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

print(MathUtils.add(5, 7))   # 12
```

When to Use What

- Instance method → operates on object data.
- Class method → operates on class-level data.
- Static method → utility function logically related to the class.

Quick Summary Table

Type	First Arg	Accesses	Use Case
Instance Method	`self`	Object	Work with object attributes
Class Method	`cls`	Class	Work with class attributes
Static Method	None	Nothing	Utility/helper function

Tiny Code

```
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius

    @classmethod
    def from_fahrenheit(cls, f):
        return cls((f - 32) * 5/9)

    @staticmethod
    def is_freezing(temp_c):
        return temp_c <= 0

t = Temperature.from_fahrenheit(32)
print(t.celsius)                  # 0.0
print(Temperature.is_freezing(-5)) # True
```

Why it Matters

Static and class methods give you more flexibility in structuring code. They help keep related functions together inside classes, even if they don't act on specific objects.

Try It Yourself

1. Create a `Circle` class with a class variable `pi = 3.14`. Add a `@classmethod get_pi()` that returns it.
2. Add a `@staticmethod area(radius)` that computes circle area using `pi`.
3. Create a circle and check both methods.
4. Try calling them on both the class and an instance.

Chapter 8. Error Handling and Exceptions

71. What Are Exceptions?

An exception is an error that happens during program execution, interrupting the normal flow. Unlike syntax errors (which stop code before running), exceptions occur at runtime and can be handled so the program doesn't crash.

Deep Dive

Common Examples of Exceptions

```
print(10 / 0)       # ZeroDivisionError
numbers = [1, 2, 3]
print(numbers[5])   # IndexError
int("hello")        # ValueError
open("nofile.txt")  # FileNotFoundError
```

Without handling, these errors stop the program immediately.

Python Exception Hierarchy

- All exceptions inherit from the built-in `Exception` class.
- Examples:
  - `ValueError` → invalid type of value.
  - `TypeError` → wrong data type.

- `KeyError` → missing dictionary key.
- `OSError` → file system-related errors.

Difference Between Errors and Exceptions

- Error: general term for something wrong (syntax or runtime).
- Exception: specific type of runtime error that can be caught and handled.

Quick Summary Table

Exception Type	Example Situation
`ZeroDivisionError`	Dividing by zero
`IndexError`	Accessing list index that doesn't exist
`KeyError`	Accessing missing dict key
`FileNotFoundError`	File does not exist
`ValueError`	Wrong value type
`TypeError`	Wrong operation on data type

Tiny Code

```
try:
    num = int("abc")   # invalid conversion
except ValueError:
    print("Oops! That was not a valid number.")
```

Why it Matters

Exceptions are unavoidable in real-world programs. By understanding them, you can write code that fails gracefully instead of crashing unexpectedly.

Try It Yourself

1. Try dividing a number by zero and observe the exception.
2. Access an element outside a list's range and note the error.
3. Use `int("abc")` and catch the `ValueError`.
4. Try opening a file that doesn't exist to see a `FileNotFoundError`.

72. Common Exceptions (`ValueError`, `TypeError`, etc.)

Python has many built-in exceptions that you will encounter often. Knowing them helps you quickly identify problems and handle them gracefully.

Deep Dive

ValueError Occurs when a function gets the right type of input but an inappropriate value.

```
int("hello")    # ValueError
```

TypeError Occurs when an operation or function is applied to an object of the wrong type.

```
"5" + 3   # TypeError: cannot add str and int
```

IndexError Happens when you try to access an index outside the valid range of a list.

```
nums = [1, 2, 3]
print(nums[5])   # IndexError
```

KeyError Raised when trying to access a dictionary key that doesn't exist.

```
person = {"name": "Alice"}
print(person["age"])   # KeyError
```

FileNotFoundError Occurs when you try to open a file that doesn't exist.

```
open("missing.txt")   # FileNotFoundError
```

ZeroDivisionError Raised when dividing a number by zero.

```
10 / 0   # ZeroDivisionError
```

Quick Summary Table

Exception	Example Trigger
`ValueError`	`int("abc")`
`TypeError`	`"5" + 3`

Exception	Example Trigger
`IndexError`	`[1,2,3][10]`
`KeyError`	`{"a":1}["b"]`
`FileNotFoundError`	`open("nofile.txt")`
`ZeroDivisionError`	`1 / 0`

Tiny Code

```
try:
    nums = [1, 2, 3]
    print(nums[10])
except IndexError:
    print("Oops! That index doesn't exist.")
```

Why it Matters

These exceptions are among the most frequent in Python. Understanding them helps you debug faster and design safer programs by predicting possible errors.

Try It Yourself

1. Trigger a `TypeError` by adding a string and a number.
2. Create a dictionary and access a non-existent key to raise a `KeyError`.
3. Open a file that doesn't exist and catch the `FileNotFoundError`.
4. Write code that divides by zero and catch the `ZeroDivisionError`.

73. `try` and `except` Blocks

Python uses `try` and `except` to handle exceptions gracefully. Instead of crashing, the program jumps to the `except` block when an error occurs.

Deep Dive

Basic Structure

```
try:
    # code that may cause an error
    x = int("abc")
except ValueError:
    print("That was not a number!")
```

- The code inside `try` is executed.
- If an exception occurs, the matching `except` block runs.
- If no error happens, the `except` block is skipped.

Catching Different Exceptions You can handle multiple specific errors separately:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You can't divide by zero.")
except ValueError:
    print("Invalid value.")
```

Catching Any Exception

```
try:
    f = open("nofile.txt")
except Exception as e:
    print("Error occurred:", e)
```

Be careful—catching all exceptions may hide bugs.

Multiple Statements in `try` If one statement fails, control jumps immediately to `except`, skipping the rest of the `try` block.

```
try:
    print("Before error")
    x = 5 / 0
    print("This won't run")
except ZeroDivisionError:
    print("Handled division by zero")
```

Quick Summary Table

Keyword	Purpose
`try`	Wraps code that may cause an error
`except`	Defines how to handle specific exceptions
`as e`	Captures the exception object

Tiny Code

```
try:
    num = int("42a")
    print("Converted:", num)
except ValueError as e:
    print("Error:", e)
```

Why it Matters

`try/except` is the foundation of error handling in Python. It lets you recover from errors, give helpful messages, and keep your program running.

Try It Yourself

1. Write code that divides two numbers but catches `ZeroDivisionError`.
2. Try converting a string to `int`, and catch `ValueError`.
3. Open a non-existent file and catch `FileNotFoundError`.
4. Use `except Exception as e` to print the error message.

74. Catching Multiple Exceptions

Sometimes, different types of errors can occur in the same block of code. Python allows you to handle multiple exceptions separately or together.

Deep Dive

Separate Except Blocks You can write different handlers for each type of exception:

```
try:
    x = int("abc")    # may cause ValueError
    y = 10 / 0        # may cause ZeroDivisionError
except ValueError:
    print("Invalid conversion to int.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Catching Multiple Exceptions in One Block You can group exceptions in a tuple:

```
try:
    data = [1, 2, 3]
    print(data[5])    # IndexError
except (ValueError, IndexError) as e:
    print("Caught an error:", e)
```

Generic Catch-All The `Exception` base class catches everything derived from it:

```
try:
    result = 10 / 0
except Exception as e:
    print("Something went wrong:", e)
```

Order Matters Python matches the first fitting `except`.

```
try:
    10 / 0
except Exception:
    print("General error")     # this will run
except ZeroDivisionError:
    print("Specific error")    # never reached
```

Always put specific exceptions first before generic ones.

Quick Summary Table

Style	Example	Use Case
Separate handlers	`except ValueError: ...`	Different handling per exception
Grouped in tuple	`except (A, B): ...`	Same handling for multiple types

Style	Example	Use Case
General `Exception` catch-all	`except Exception as e:`	Debugging, fallback handling

Tiny Code

```
try:
    num = int("xyz")
    result = 10 / 0
except ValueError:
    print("Conversion failed.")
except ZeroDivisionError:
    print("Math error: division by zero.")
```

Why it Matters

Most real-world code must guard against different failure modes. Being able to catch multiple exceptions lets you handle each case correctly without stopping the whole program.

Try It Yourself

1. Convert `"abc"` to an integer and catch `ValueError`.
2. Divide by zero in the same block, and handle `ZeroDivisionError`.
3. Use one `except (ValueError, ZeroDivisionError)` to handle both at once.
4. Add a final generic `except Exception as e:` to print any unexpected error.

75. `else` in Exception Handling

In Python, you can use an `else` block with `try`/`except`. The `else` block runs only if no exception was raised in the `try` block.

Deep Dive

Basic Structure

```
try:
    x = int("42")   # no error here
except ValueError:
    print("Conversion failed.")
else:
    print("Conversion successful:", x)
```

- If the code in `try` succeeds, the `else` block runs.
- If an exception occurs, the `else` block is skipped.

Why Use `else`?

- Keeps your `try` block focused only on code that might fail.
- Puts the "safe" code in `else`, separating it clearly.

Example:

```
try:
    f = open("data.txt")
except FileNotFoundError:
    print("File not found.")
else:
    print("File opened successfully.")
    f.close()
```

With Multiple Exceptions

```
try:
    num = int("100")
except ValueError:
    print("Invalid number.")
else:
    print("Parsed successfully:", num)
```

Quick Summary Table

Block	Runs When
`try`	Always, until error happens
`except`	If an error of specified type occurs
`else`	If no errors happened in `try`

Tiny Code

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Division failed.")
else:
    print("Division successful:", result)
```

Why it Matters

Using `else` makes exception handling cleaner: risky code in `try`, error handling in `except`, and safe follow-up code in `else`. This improves readability and reduces mistakes.

Try It Yourself

1. Write code that reads a number from a string with `int()`. If it fails, handle `ValueError`. If it succeeds, print `"Valid number"` in `else`.
2. Try dividing two numbers, catching `ZeroDivisionError`, and use `else` to print the result if successful.
3. Open an existing file in `try`, handle `FileNotFoundError`, and confirm success in `else`.

76. `finally` Block

In Python, the `finally` block is used with `try`/`except` to guarantee that certain code always runs — no matter what happens. This is useful for cleanup tasks like closing files or releasing resources.

Deep Dive

Basic Structure

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Division failed.")
finally:
    print("This always runs.")
```

- If no error: `finally` still runs.
- If an error occurs and is caught: `finally` still runs.
- If an error occurs and is not caught: `finally` still runs before the program crashes.

With `else` and `finally` Together

```
try:
    num = int("42")
except ValueError:
    print("Invalid number")
else:
    print("Conversion successful:", num)
finally:
    print("Execution finished")
```

Order of execution here:

1. `try` block
2. `except` (if error) OR `else` (if no error)
3. `finally` (always)

Practical Example: Closing Files

```
try:
    f = open("data.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("File not found.")
finally:
    print("Closing file...")
    try:
        f.close()
    except:
        pass
```

Quick Summary Table

Block	Runs When
`try`	Always, until error happens
`except`	If an error occurs
`else`	If no error occurs
`finally`	Always, regardless of error or success

Tiny Code

```
try:
    print("Opening file...")
    f = open("missing.txt")
except FileNotFoundError:
    print("Error: File not found.")
finally:
    print("Cleanup done.")
```

Why it Matters

The `finally` block ensures important cleanup (like closing files, saving data, disconnecting from databases) always happens — even if the program crashes in the middle.

Try It Yourself

1. Write code that divides two numbers with `try/except`, then add a `finally` block to print `"End of operation"`.
2. Try opening a file in `try`, handle `FileNotFoundError`, and in `finally` print `"Closing resources"`.
3. Combine `try`, `except`, `else`, and `finally` in one program and observe the execution order.

77. Raising Exceptions (`raise`)

Sometimes, instead of waiting for Python to throw an error, you may want to raise an exception yourself when something unexpected happens. This is done with the `raise` keyword.

Deep Dive

Basic Usage

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero!")
    return a / b
```

```
print(divide(10, 2))   # 5.0
print(divide(5, 0))    # Raises ZeroDivisionError
```

Here, we explicitly raise `ZeroDivisionError` when dividing by zero.

Raising Built-in Exceptions You can raise any built-in exception manually:

```
age = -1
if age < 0:
    raise ValueError("Age cannot be negative")
```

Raising Custom Messages Exceptions can carry useful error messages:

```
name = ""
if not name:
    raise Exception("Name must not be empty")
```

Re-raising Exceptions Sometimes you catch an error but still want to pass it upward:

```
try:
    x = int("abc")
except ValueError as e:
    print("Caught an error:", e)
    raise   # re-raises the same exception
```

Quick Summary Table

Keyword	Purpose	Example
`raise`	Manually throw an exception	`raise ValueError("Invalid input")`
Message	Provide details for debugging	`raise Exception("Something went wrong")`
Re-raise	Pass the error up the stack	`raise` inside `except`

Tiny Code

```python
def check_age(age):
    if age < 18:
        raise ValueError("Must be at least 18 years old.")
    return "Access granted."

print(check_age(20))   # Access granted
print(check_age(15))   # Raises ValueError
```

Why it Matters

Raising exceptions gives you control. Instead of letting bad data silently continue, you can stop execution, show a meaningful error, and prevent bigger problems later.

Try It Yourself

1. Write a `withdraw(balance, amount)` function. If `amount > balance`, raise a `ValueError`.
2. Create a `check_name(name)` function that raises an exception if the string is empty.
3. Inside a `try/except`, catch a `ValueError` and then re-raise it to see the traceback.
4. Raise a custom `Exception("Custom error message")` and print it.

78. Creating Custom Exceptions

In addition to Python's built-in exceptions, you can define your own custom exceptions to make error handling more meaningful in your programs.

Deep Dive

Defining a Custom Exception A custom exception is just a class that inherits from Python's built-in `Exception` class.

```python
class NegativeNumberError(Exception):
    """Raised when a number is negative."""
    pass
```

Using the Custom Exception

```
def square_root(x):
    if x < 0:
        raise NegativeNumberError("Cannot take square root of negative number")
    return x  0.5

print(square_root(9))   # 3.0
print(square_root(-4))  # Raises NegativeNumberError
```

Adding Extra Functionality You can extend custom exceptions with attributes.

```
class BalanceError(Exception):
    def __init__(self, balance, message="Insufficient funds"):
        self.balance = balance
        self.message = message
        super().__init__(f"{message}. Balance: {balance}")

def withdraw(balance, amount):
    if amount > balance:
        raise BalanceError(balance)
    return balance - amount

withdraw(100, 200)   # Raises BalanceError
```

Catching Custom Exceptions

```
try:
    square_root(-1)
except NegativeNumberError as e:
    print("Custom error caught:", e)
```

Why Create Custom Exceptions?

1. Make your errors descriptive and domain-specific.
2. Easier debugging since you know exactly what went wrong.
3. Provide structured error handling in larger projects.

Quick Summary Table

Step	Example
Define custom error	`class MyError(Exception): ...`
Raise it	`raise MyError("Something happened")`

Step	Example
Catch it	`except MyError as e:`

Tiny Code

```
class AgeError(Exception):
    pass

def register(age):
    if age < 18:
        raise AgeError("Must be 18 or older to register")
    return "Registered!"

try:
    print(register(16))
except AgeError as e:
    print("Registration failed:", e)
```

Why it Matters

Custom exceptions make your programs more self-explanatory and professional. Instead of generic errors, you provide meaningful messages tailored to your application's domain.

Try It Yourself

1. Create a `PasswordError` class for invalid passwords.
2. Write a function `set_password(pw)` that raises `PasswordError` if the password is less than 8 characters.
3. Create a `TemperatureError` class and raise it if input temperature is below absolute zero (-273°C).
4. Catch your custom exception and print the message.

79. Assertions (`assert`)

An assertion is a quick way to test if a condition in your program is true. If the condition is false, Python raises an `AssertionError`. Assertions are often used for debugging and catching mistakes early.

Deep Dive

Basic Usage

```
x = 5
assert x > 0    # passes, nothing happens
assert x < 0    # fails, raises AssertionError
```

With a Custom Message

```
age = -1
assert age >= 0, "Age cannot be negative"
```

If the condition is false, it raises:

```
AssertionError: Age cannot be negative
```

When to Use Assertions

- To check assumptions during development.
- To catch impossible states in your logic.
- For debugging, not for handling user errors (use exceptions for that).

Turning Off Assertions

- Assertions can be disabled when running Python with the `-O` (optimize) flag.
- Example: `python -O program.py` → all `assert` statements are skipped.

Practical Example

```
def divide(a, b):
    assert b != 0, "Denominator must not be zero"
    return a / b

print(divide(10, 2))   # 5.0
print(divide(5, 0))    # AssertionError
```

Quick Summary Table

Syntax	Behavior
`assert condition`	Raises `AssertionError` if condition false

Syntax	Behavior
`assert condition, msg`	Raises with custom message
Disabled with `-O`	Skips all asserts

Tiny Code

```
score = 95
assert 0 <= score <= 100, "Score must be between 0 and 100"
print("Score is valid!")
```

Why it Matters

Assertions help you detect logic errors early. They make your intentions clear in code and act as built-in sanity checks during development.

Try It Yourself

1. Write an `assert` to check that a number is positive.
2. Add an assertion in a function to make sure a list isn't empty before accessing it.
3. Use `assert` to check that temperature is above -273 (absolute zero).
4. Run your program with `python -O` and see that assertions are skipped.

80. Best Practices for Error Handling

Good error handling makes your programs reliable, readable, and easier to maintain. Instead of letting programs crash or hiding bugs, you should follow certain best practices.

Deep Dive

1. Be Specific in `except` Blocks Catch only the exceptions you expect, not all of them.

```
try:
    num = int("abc")
except ValueError:
    print("Invalid number!")   # good
```

Avoid:

```
except:
    print("Something went wrong")   # too vague
```

2. Use `finally` for Cleanup Always free resources like files, network connections, or databases.

```
try:
    f = open("data.txt")
    content = f.read()
except FileNotFoundError:
    print("File not found.")
finally:
    f.close()
```

3. Keep `try` Blocks Small Put only the risky code inside `try`, not everything.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Math error")
```

Better than wrapping the entire function.

4. Don't Hide Bugs Catching all exceptions with `except Exception` should be a last resort. Otherwise, real bugs get hidden.
5. Raise Exceptions When Needed Instead of returning special values like `-1`, raise meaningful errors.

```
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError("Insufficient funds")
    return balance - amount
```

6. Create Custom Exceptions for Clarity For domain-specific logic, define your own exceptions (e.g., `PasswordTooShortError`).
7. Log Errors Use Python's `logging` module instead of just `print()`.

```
import logging
logging.error("File not found", exc_info=True)
```

Quick Summary Table

Practice	Why It Matters
Catch specific exceptions	Avoids hiding unrelated bugs
Use `finally` for cleanup	Ensures resources are freed
Keep `try` small	Improves readability
Raise exceptions	Signals errors clearly
Custom exceptions	Domain-specific clarity
Logging over printing	Professional error tracking

Tiny Code

```
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        raise ValueError("b must not be zero")

print(safe_divide(10, 2))
print(safe_divide(5, 0))   # raises ValueError
```

Why it Matters

Well-structured error handling prevents small mistakes from becoming big failures. It keeps your programs predictable, professional, and easier to debug.

Try It Yourself

1. Write a function `read_file(filename)` that catches `FileNotFoundError` and raises a new exception with a clearer message.
2. Add a `finally` block to always print `"Operation complete"`.
3. Try logging an error instead of printing it.
4. Refactor a long `try` block so it only wraps the risky line of code.

Chapter 9. Advanced Python Features

81. List Comprehensions

A list comprehension is a concise way to create lists in Python. It lets you generate new lists by applying an expression to each item in an existing sequence (or iterable), often replacing loops with a single readable line.

Deep Dive

Basic Syntax

```
[expression for item in iterable]
```

Example:

```
nums = [1, 2, 3, 4]
squares = [x2 for x in nums]
print(squares)   # [1, 4, 9, 16]
```

With a Condition

```
evens = [x for x in range(10) if x % 2 == 0]
print(evens)   # [0, 2, 4, 6, 8]
```

Nested Loops in Comprehensions

```
pairs = [(x, y) for x in [1, 2] for y in [3, 4]]
print(pairs)   # [(1, 3), (1, 4), (2, 3), (2, 4)]
```

With Functions

```
words = ["hello", "python", "world"]
uppercased = [w.upper() for w in words]
print(uppercased)   # ['HELLO', 'PYTHON', 'WORLD']
```

Replacing Loops Loop version:

```
squares = []
for x in range(5):
    squares.append(x2)
```

Comprehension version:

```
squares = [x2 for x in range(5)]
```

Quick Summary Table

Form	Example
Simple comprehension	`[x*2 for x in range(5)]`
With condition	`[x for x in range(10) if x % 2 == 0]`
Nested loops	`[(x,y) for x in [1,2] for y in [3,4]]`
With function	`[f(x) for x in items]`

Tiny Code

```
nums = [1, 2, 3, 4, 5]
double = [n * 2 for n in nums if n % 2 != 0]
print(double)   # [2, 6, 10]
```

Why it Matters

List comprehensions make your code shorter, faster, and easier to read. They are a hallmark of Pythonic style, turning loops and conditions into expressive one-liners.

Try It Yourself

1. Create a list of squares from 1 to 10 using a list comprehension.
2. Make a list of only the odd numbers between 1 and 20.
3. Use a comprehension to extract the first letter of each word in `["apple", "banana", "cherry"]`.
4. Build a list of coordinate pairs `(x, y)` for `x` in `[1,2,3]` and `y` in `[4,5]`.

82. Dictionary Comprehensions

A dictionary comprehension is a compact way to build dictionaries by combining expressions and loops into a single line. It works like list comprehensions but produces key–value pairs instead of list elements.

Deep Dive

Basic Syntax

```
{key_expression: value_expression for item in iterable}
```

Example:

```
nums = [1, 2, 3, 4]
squares = {x: x2 for x in nums}
print(squares)   # {1: 1, 2: 4, 3: 9, 4: 16}
```

With a Condition

```
even_squares = {x: x2 for x in range(10) if x % 2 == 0}
print(even_squares)   # {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

Swapping Keys and Values

```
fruit = {"a": "apple", "b": "banana", "c": "cherry"}
swap = {v: k for k, v in fruit.items()}
print(swap)   # {'apple': 'a', 'banana': 'b', 'cherry': 'c'}
```

With Functions

```
words = ["hello", "world", "python"]
lengths = {w: len(w) for w in words}
print(lengths)   # {'hello': 5, 'world': 5, 'python': 6}
```

Nested Loops in Dictionary Comprehensions

```
pairs = {(x, y): x*y for x in [1, 2] for y in [3, 4]}
print(pairs)   # {(1, 3): 3, (1, 4): 4, (2, 3): 6, (2, 4): 8}
```

Quick Summary Table

Form	Example
Basic dict comp	`{x: x*2 for x in range(3)}`
With condition	`{x: x2 for x in range(6) if x % 2 == 0}`
Swap keys and values	`{v: k for k, v in dict.items()}`
Using function	`{w: len(w) for w in words}`
Nested loops	`{(x,y): x*y for x in A for y in B}`

Tiny Code

```
students = ["Alice", "Bob", "Charlie"]
grades = {name: "Pass" if len(name) <= 4 else "Review" for name in students}
print(grades)   # {'Alice': 'Review', 'Bob': 'Pass', 'Charlie': 'Review'}
```

Why it Matters

Dictionary comprehensions save time and reduce boilerplate when building mappings from existing data. They make code cleaner, more expressive, and Pythonic.

Try It Yourself

1. Create a dictionary mapping numbers 1–5 to their cubes.
2. Build a dictionary of words and their lengths from `["cat", "elephant", "dog"]`.
3. Flip a dictionary `{"x": 1, "y": 2}` so values become keys.
4. Generate a dictionary mapping `(x, y)` pairs to `x + y` for `x` in `[1,2]` and `y` in `[3,4]`.

83. Set Comprehensions

A set comprehension is similar to a list comprehension, but it produces a set—an unordered collection of unique elements. It's a concise way to build sets with loops and conditions.

Deep Dive

Basic Syntax

```
{expression for item in iterable}
```

Example:

```
nums = [1, 2, 2, 3, 4, 4]
unique_squares = {x2 for x in nums}
print(unique_squares)   # {16, 1, 4, 9}
```

With a Condition

```
evens = {x for x in range(10) if x % 2 == 0}
print(evens)   # {0, 2, 4, 6, 8}
```

From a String

```
letters = {ch for ch in "banana"}
print(letters)   # {'a', 'b', 'n'}
```

With Functions

```
words = ["hello", "world", "python"]
lengths = {len(w) for w in words}
print(lengths)   # {5, 6}
```

Nested Loops in Set Comprehensions

```
pairs = {(x, y) for x in [1, 2] for y in [3, 4]}
print(pairs)   # {(1, 3), (1, 4), (2, 3), (2, 4)}
```

Quick Summary Table

Form	Example
Simple set comp	`{x*2 for x in range(5)}`
With condition	`{x for x in range(10) if x % 2 == 0}`
From string	`{ch for ch in "banana"}`
With function	`{len(w) for w in words}`
Nested loops	`{(x,y) for x in A for y in B}`

Tiny Code

```
nums = [1, 2, 3, 2, 1, 4]
squares = {n2 for n in nums if n % 2 != 0}
print(squares)   # {1, 9}
```

Why it Matters

Set comprehensions provide a quick way to eliminate duplicates and apply transformations at the same time. They're useful for data cleaning, filtering, and fast membership checks.

Try It Yourself

1. Create a set of squares from 1–10.
2. Build a set of all vowels in the word `"programming"`.
3. Make a set of numbers between 1–20 that are divisible by 3.
4. Generate a set of `(x, y)` pairs where `x` in `[1,2,3]` and `y` in `[4,5]`.

84. Generators (`yield`)

A generator is a special type of function that lets you produce a sequence of values lazily, one at a time, using the `yield` keyword. Unlike regular functions, generators don't return everything at once—they pause and resume.

Deep Dive

Basic Generator

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for num in count_up_to(5):
    print(num)
```

Output:

```
1
2
3
4
5
```

Difference Between `return` and `yield`

- `return` → ends the function and gives a single value.
- `yield` → pauses the function, remembers its state, and continues next time.

Using Generators with `next()`

```
gen = count_up_to(3)
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
```

Infinite Generators Generators can produce endless sequences:

```
def even_numbers():
    n = 0
    while True:
        yield n
        n += 2

gen = even_numbers()
for _ in range(5):
    print(next(gen))   # 0 2 4 6 8
```

Generator Expressions Like list comprehensions but with parentheses:

```
squares = (x2 for x in range(5))
for s in squares:
    print(s)
```

Quick Summary Table

Feature	Example	Behavior
`yield` keyword	`yield x`	Produces one value at a time
Pause & resume	Uses `next()`	Continues from last state

Feature	Example	Behavior
Generator function	`def f(): yield ...`	Creates a generator
Generator expr	`(x2 for x in range(5))`	Compact generator syntax

Tiny Code

```
def fibonacci(limit):
    a, b = 0, 1
    while a <= limit:
        yield a
        a, b = b, a + b

for num in fibonacci(20):
    print(num)
```

Why it Matters

Generators are memory-efficient because they don't build the whole list in memory. They're ideal for large datasets, streams of data, or infinite sequences.

Try It Yourself

1. Write a generator `countdown(n)` that yields numbers from `n` down to `1`.
2. Make a generator that yields only odd numbers up to 15.
3. Create a generator expression for cubes of numbers 1–5.
4. Modify the Fibonacci generator to stop after producing 10 numbers.

85. Iterators

An iterator is an object that represents a stream of data. It returns items one at a time when you call `next()` on it, and it remembers its position between calls. Iterators are the foundation of loops, comprehensions, and generators in Python.

Deep Dive

Iterator Protocol An object is an iterator if it implements two methods:

- `__iter__()` → returns the iterator object itself.
- `__next__()` → returns the next value, or raises `StopIteration` when done.

Built-in Iterators

```
nums = [1, 2, 3]
it = iter(nums)   # get iterator

print(next(it))   # 1
print(next(it))   # 2
print(next(it))   # 3
# next(it) now raises StopIteration
```

For Loops Use Iterators Under the Hood

```
for n in [1, 2, 3]:
    print(n)
```

is equivalent to:

```
nums = [1, 2, 3]
it = iter(nums)
while True:
    try:
        print(next(it))
    except StopIteration:
        break
```

Custom Iterator You can build your own iterator by defining `__iter__` and `__next__`:

```
class CountDown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
```

```
        if self.current <= 0:
            raise StopIteration
        self.current -= 1
        return self.current + 1

for num in CountDown(5):
    print(num)
```

Output:

```
5
4
3
2
1
```

Quick Summary Table

Concept	Example	Purpose
`iter(obj)`	`it = iter([1,2,3])`	Get iterator from iterable
`next(it)`	`next(it)`	Get next value
StopIteration	Exception when done	Signals end of iteration
Custom	Define `__iter__`, `__next__`	Create your own sequence

Tiny Code

```
nums = [10, 20, 30]
it = iter(nums)

print(next(it))  # 10
print(next(it))  # 20
print(next(it))  # 30
```

Why it Matters

Understanding iterators explains how loops, generators, and comprehensions actually work in Python. Iterators allow Python to handle large datasets efficiently, consuming one item at a time.

Try It Yourself

1. Use `iter()` and `next()` on a string like `"hello"` to get characters one by one.
2. Build a simple custom iterator that counts from 1 to 5.
3. Write a `for` loop manually using `while True` and `next()` with `StopIteration`.
4. Create a custom iterator `EvenNumbers(n)` that yields even numbers up to `n`.

86. Decorators

A decorator is a special function that takes another function as input, adds extra behavior to it, and returns a new function. In Python, decorators are often used for logging, authentication, caching, and more.

Deep Dive

Basic Decorator

```
def my_decorator(func):
    def wrapper():
        print("Before function runs")
        func()
        print("After function runs")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Before function runs
Hello!
After function runs
```

- `@my_decorator` is shorthand for `say_hello = my_decorator(say_hello)`.

Decorators with Arguments

```
def repeat(func):
    def wrapper():
        for _ in range(3):
            func()
    return wrapper

@repeat
def greet():
    print("Hi!")

greet()
```

Output:

```
Hi!
Hi!
Hi!
```

Passing Arguments to Wrapped Function

```
def log_args(func):
    def wrapper(*args, kwargs):
        print("Arguments:", args, kwargs)
        return func(*args, kwargs)
    return wrapper

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

print(add(3, 5))
```

Using `functools.wraps` Without it, the decorated function loses its original name and docstring.

```
from functools import wraps

def decorator(func):
    @wraps(func)
    def wrapper(*args, kwargs):
        return func(*args, kwargs)
    return wrapper
```

Quick Summary Table

Feature	Example	Purpose
Basic decorator	`@my_decorator`	Add behavior before/after function
With args	`def wrapper(*args,kwargs)`	Works with any function signature
Multiple decorators	`@d1` + `@d2`	Stacks behaviors
`functools.wraps`	`@wraps(func)`	Preserve metadata

Tiny Code

```
def uppercase(func):
    def wrapper():
        result = func()
        return result.upper()
    return wrapper

@uppercase
def message():
    return "hello world"

print(message())   # HELLO WORLD
```

Why it Matters

Decorators are a powerful way to separate what a function does from how it's used. They make code reusable, clean, and Pythonic.

Try It Yourself

1. Write a decorator `@timer` that prints how long a function takes to run.
2. Create a decorator `@authenticate` that prints `"Access denied"` unless a variable `user_logged_in = True`.
3. Combine two decorators on the same function and observe the order of execution.
4. Use `functools.wraps` to keep the function's original `__name__`.

87. Context Managers (Custom)

A context manager is a Python construct that properly manages resources, like opening and closing files. You usually use it with the `with` statement. While Python has built-in context managers (like `open`), you can also create your own.

Deep Dive

Using `with` Built-in

```
with open("data.txt", "r") as f:
    content = f.read()
```

Here, `open` is a context manager: it opens the file, then automatically closes it when done.

Creating a Custom Context Manager with a Class To make your own, define `__enter__` and `__exit__`.

```
class MyResource:
    def __enter__(self):
        print("Resource acquired")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Resource released")

with MyResource() as r:
    print("Using resource")
```

Output:

```
Resource acquired
Using resource
Resource released
```

Handling Errors in `__exit__` `__exit__` can suppress exceptions if it returns `True`.

```python
class SafeDivide:
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        return True   # suppress error

with SafeDivide():
    print(10 / 0)   # No crash!
```

Creating a Context Manager with `contextlib`

```python
from contextlib import contextmanager

@contextmanager
def managed_resource():
    print("Start")
    yield
    print("End")

with managed_resource():
    print("Inside block")
```

Output:

```
Start
Inside block
End
```

Quick Summary Table

Method	How it Works	Example
Class-based	Define `__enter__` and `__exit__`	`with MyClass(): ...`
Function-based	Use `@contextmanager` decorator	`with managed_resource(): ...`
Built-in examples	`open`, `threading.Lock`, `sqlite3`	`with open("f.txt") as f:`

Tiny Code

```python
from contextlib import contextmanager

@contextmanager
def open_upper(filename):
    f = open(filename, "r")
    try:
        yield (line.upper() for line in f)
    finally:
        f.close()

with open_upper("data.txt") as lines:
    for line in lines:
        print(line)
```

Why it Matters

Custom context managers let you manage setup and cleanup tasks automatically. They make code safer, reduce errors, and ensure resources are always released properly.

Try It Yourself

1. Write a context manager class that prints `"Start"` when entering and `"End"` when exiting.
2. Create one that temporarily changes the working directory and restores it afterwards.
3. Use `@contextmanager` to make a timer context that prints how long the block took.
4. Build a safe database connection context that opens, yields, then closes automatically.

88. `with` and Resource Management

The `with` statement in Python is a shortcut for using context managers. It ensures resources (like files, network connections, or locks) are acquired and released properly, even if errors occur.

Deep Dive

File Handling with `with`

```python
with open("notes.txt", "w") as f:
    f.write("Hello, Python!")
```

- File opens automatically.
- File closes automatically after the block, even if an error happens.

Multiple Resources in One `with`

```
with open("input.txt", "r") as infile, open("output.txt", "w") as outfile:
    for line in infile:
        outfile.write(line.upper())
```

Both files are managed safely within the same `with` statement.

Using `with` for Locks (Threading Example)

```
import threading

lock = threading.Lock()
with lock:
    # critical section
    print("Safe access")
```

The lock is automatically acquired and released.

Database Connections Some libraries provide context managers for connections.

```
import sqlite3

with sqlite3.connect("example.db") as conn:
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS users(id INTEGER)")
```

Connection commits and closes automatically at the end.

Custom Resource Management Any class with `__enter__` and `__exit__` can be used in a `with` block.

```
class Resource:
    def __enter__(self):
        print("Acquired resource")
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Released resource")

with Resource():
    print("Using resource")
```

Output:

```
Acquired resource
Using resource
Released resource
```

Quick Summary Table

Resource Type	Example with `with`	Benefit
File	`with open("file.txt") as f:`	Auto-close file
Thread lock	`with lock:`	Auto-release lock
Database connection	`with sqlite3.connect(...) as conn:`	Auto-commit & close
Custom resource	`with MyResource(): ...`	Custom cleanup

Tiny Code

```
with open("demo.txt", "w") as f:
    f.write("Resource managed with 'with'")
```

Why it Matters

Resource management is crucial to avoid memory leaks, file corruption, or dangling connections. The `with` statement makes code safer, cleaner, and more professional.

Try It Yourself

1. Write a `with open("data.txt", "r")` block that prints each line.
2. Use `with` to copy one file into another.
3. Create a threading lock and use it with `with` in a simple program.
4. Write a custom class with `__enter__` and `__exit__` that logs when it starts and stops.

89. Modules `itertools` & `functools`

Python provides `itertools` and `functools` as standard libraries to work with iterators and functional programming tools. They let you process data efficiently and write more expressive code.

Deep Dive

`itertools` – Tools for Iteration

- Infinite Iterators

```
import itertools

counter = itertools.count(start=1, step=2)
print(next(counter))  # 1
print(next(counter))  # 3
```

- Cycling and Repeating

```
colors = itertools.cycle(["red", "green", "blue"])
print(next(colors))  # red
print(next(colors))  # green

repeat_hello = itertools.repeat("hello", 3)
print(list(repeat_hello))  # ['hello', 'hello', 'hello']
```

- Combinatorics

```
from itertools import permutations, combinations

print(list(permutations([1, 2, 3], 2)))
# [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]

print(list(combinations([1, 2, 3], 2)))
# [(1, 2), (1, 3), (2, 3)]
```

- Chaining Iterables

```
from itertools import chain
print(list(chain("ABC", "123")))  # ['A','B','C','1','2','3']
```

`functools` – Tools for Functions

- `reduce` → apply a function cumulatively.

```python
from functools import reduce

nums = [1, 2, 3, 4]
product = reduce(lambda a, b: a * b, nums)
print(product)  # 24
```

- `lru_cache` → memoize function results.

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

print(fib(30))  # fast due to caching
```

- `partial` → fix some arguments of a function.

```python
from functools import partial

def power(base, exponent):
    return base  exponent

square = partial(power, exponent=2)
print(square(5))  # 25
```

Quick Summary Table

Module	Function	Example	Purpose
itertools	`count`	`count(1,2)`	Infinite counter
itertools	`cycle`	`cycle(['A','B'])`	Repeat sequence forever
itertools	`permutations`	`permutations([1,2,3],2)`	All orderings
itertools	`combinations`	`combinations([1,2,3],2)`	All unique pairs
functools	`reduce`	`reduce(lambda x,y: x+y, [1,2,3])`	Cumulative reduction
functools	`lru_cache`	`@lru_cache`	Cache results for speed
functools	`partial`	`partial(func, arg=value)`	Pre-fill arguments

Tiny Code

```
from itertools import accumulate
print(list(accumulate([1, 2, 3, 4])))  # [1, 3, 6, 10]
```

Why it Matters

`itertools` and `functools` give you powerful building blocks for iteration and function manipulation. They make complex tasks simpler, faster, and more memory-efficient.

Try It Yourself

1. Use `itertools.combinations` to list all pairs from `[1, 2, 3, 4]`.
2. Create an infinite counter with `itertools.count()` and print the first 5 values.
3. Use `functools.reduce` to compute the sum of `[10, 20, 30]`.
4. Define a `cube` function using `functools.partial(power, exponent=3)`.

90. Type Hints (`typing` Module)

Type hints let you specify the expected data types of variables, function arguments, and return values. They don't change how the code runs, but they make it easier to read, maintain, and catch errors early with tools like `mypy`.

Deep Dive

Basic Function Hints

```
def greet(name: str) -> str:
    return "Hello, " + name
```

- `name: str` means `name` should be a string.
- `-> str` means the function returns a string.

Variable Hints

```
age: int = 25
pi: float = 3.14159
active: bool = True
```

Using `List`, `Dict`, and `Tuple`

```
from typing import List, Dict, Tuple

numbers: List[int] = [1, 2, 3]
user: Dict[str, int] = {"Alice": 25, "Bob": 30}
point: Tuple[int, int] = (10, 20)
```

Optional Values

```
from typing import Optional

def find_user(id: int) -> Optional[str]:
    if id == 1:
        return "Alice"
    return None
```

Union Types

```
from typing import Union

def add(x: Union[int, float], y: Union[int, float]) -> Union[int, float]:
    return x + y
```

Type Aliases

```
UserID = int
def get_user(id: UserID) -> str:
    return "User" + str(id)
```

Callable (Functions as Arguments)

```
from typing import Callable

def apply(func: Callable[[int, int], int], a: int, b: int) -> int:
    return func(a, b)

print(apply(lambda x, y: x + y, 2, 3))  # 5
```

Quick Summary Table

Type Hint	Example	Meaning
Basic	`x: int`, `def f()->str`	Simple types
List, Dict	`List[int]`, `Dict[str,int]`	Collections with types
Tuple	`Tuple[int,str]`	Fixed-size sequence
Optional	`Optional[str]`	String or None
Union	`Union[int,float]`	One of several types
Callable	`Callable[[int,int],int]`	Function type
Alias	`UserID = int`	Custom type name

Tiny Code

```
from typing import List

def average(values: List[float]) -> float:
    return sum(values) / len(values)

print(average([1.0, 2.0, 3.0]))  # 2.0
```

Why it Matters

Type hints improve clarity and enable better error detection during development. They help teams understand code faster and catch mistakes before running the program.

Try It Yourself

1. Add type hints to a function `def square(x): return x*x`.
2. Write a function `join(names)` that expects a `List[str]` and returns a `str`.
3. Use `Optional[int]` for a function that may return `None`.
4. Create a function `operate` that accepts a `Callable[[int,int],int]` and applies it to two numbers.

Chapter 10. Python in Practices

91. REPL & Interactive Mode

Python comes with an interactive environment called the REPL (Read–Eval–Print Loop). It lets you type Python commands one at a time and see results immediately, making it perfect

for learning, testing, and quick experiments.

Deep Dive

Open a terminal and type:

```
python
```

or sometimes:

```
python3
```

You'll see a prompt like:

```
>>>
```

where you can type Python code directly.

Basic Usage

```
>>> 2 + 3
5
>>> "hello".upper()
'HELLO'
```

The REPL evaluates each expression and prints the result instantly.

Multi-line Input For blocks like loops or functions, use indentation:

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

Exploring Objects You can quickly inspect functions and objects:

```
>>> help(str)
>>> dir(list)
```

Using the Underscore _ The REPL stores the last result in _:

```
>>> 5 * 5
25
>>> _ + 10
35
```

Exiting the REPL

- Press `Ctrl+D` (Linux/Mac) or `Ctrl+Z` + Enter (Windows).
- Or type `exit()` or `quit()`.

Enhanced REPLs

- IPython → advanced REPL with colors, auto-complete, and history.
- Jupyter Notebook → browser-based interactive coding environment.

Quick Summary Table

Feature	Example	Purpose
Run REPL	`python`	Start interactive mode
Expression	`2 + 3` → `5`	Immediate evaluation
Multi-line	`for i in ...`	Supports blocks of code
Inspect object	`dir(obj)`, `help(obj)`	Explore methods & docs
Last result	`_`	Use last computed value

Tiny Code

```
>>> x = 10
>>> y = 20
>>> x + y
30
>>> _
30
>>> _ * 2
60
```

Why it Matters

The REPL makes Python beginner-friendly and powerful for professionals. It's like a live scratchpad where you can test ideas, debug small snippets, or explore libraries interactively.

Try It Yourself

1. Start the Python REPL and calculate `7 * 8`.
2. Use `help(int)` to see details about integers.
3. Assign a variable, then use `_` to reuse its value.
4. Try an enhanced REPL like `ipython` for auto-completion.

92. Debugging (`pdb`)

Python includes a built-in debugger called `pdb`. It allows you to pause execution, step through code line by line, inspect variables, and find bugs interactively.

Deep Dive

Starting the Debugger Insert this line where you want to pause:

```
import pdb; pdb.set_trace()
```

When the program runs, it will stop there and open an interactive debugging session.

Common `pdb` Commands

Command	Meaning
`n`	Next line (step over)
`s`	Step into a function
`c`	Continue until next breakpoint
`l`	List source code around current line
`p var`	Print the value of `var`
`q`	Quit the debugger
`b num`	Set a breakpoint at line number `num`

Example Debugging Session

```
def divide(a, b):
    result = a / b
    return result

x = 10
y = 0
```

```
import pdb; pdb.set_trace()
print(divide(x, y))
```

When run:

```
(Pdb) p x
10
(Pdb) p y
0
(Pdb) n
ZeroDivisionError: division by zero
```

Running a Script with Debug Mode You can also run the debugger directly from the command line:

```
python -m pdb myscript.py
```

Modern Alternatives

- ipdb → improved pdb with colors and better interface.
- debugpy → used in VS Code and IDEs for integrated debugging.

Tiny Code

```
def greet(name):
    message = "Hello " + name
    return message

import pdb; pdb.set_trace()
print(greet("Alice"))
```

Inside pdb, type:

```
(Pdb) p name
(Pdb) n
```

Why it Matters

Debugging with `pdb` helps you see *exactly* what your program is doing step by step. Instead of guessing where things go wrong, you can inspect state directly and fix issues faster.

Try It Yourself

1. Write a function that divides two numbers and insert `pdb.set_trace()` before the division. Step through and print variables.
2. Run a script with `python -m pdb file.py` and use `n` and `s` to move through code.
3. Try setting a breakpoint with `b` and continuing with `c`.
4. Experiment with inspecting variables using `p var` during debugging.

93. Logging (`logging` Module)

The `logging` module in Python is used to record messages about what your program is doing. Unlike `print()`, logging is flexible, configurable, and suitable for real-world applications.

Deep Dive

Basic Logging

```
import logging

logging.basicConfig(level=logging.INFO)
logging.info("Program started")
logging.warning("This is a warning")
logging.error("An error occurred")
```

Output:

```
INFO:root:Program started
WARNING:root:This is a warning
ERROR:root:An error occurred
```

Log Levels Logging has different severity levels:

Level	Function	Meaning
DEBUG	`logging.debug()`	Detailed information for devs
INFO	`logging.info()`	General program information
WARNING	`logging.warning()`	Something unexpected happened
ERROR	`logging.error()`	A serious problem occurred
CRITICAL	`logging.critical()`	Very severe error

Custom Formatting

```
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(message)s",
    level=logging.DEBUG
)

logging.debug("Debugging details")
```

Example output:

```
2025-09-14 19:30:01,234 - DEBUG - Debugging details
```

Logging to a File

```
logging.basicConfig(filename="app.log", level=logging.INFO)
logging.info("This message goes into the log file")
```

Separate Logger Instances

```
logger = logging.getLogger("myapp")
logger.setLevel(logging.DEBUG)

logger.info("App is running")
```

Why Logging Instead of Print?

- `print()` always goes to stdout.
- `logging` lets you choose where messages go: console, file, system log, etc.
- You can control severity and disable logs without changing code.

Quick Summary Table

Feature	Example	Purpose
Basic log	`logging.info("msg")`	Simple logging
Levels	`DEBUG`, `INFO`, `WARNING`, etc.	Control importance
Formatting	`%(asctime)s - %(levelname)s...`	Add timestamps, names
To file	`filename="app.log"`	Persist logs
Custom logger	`getLogger("name")`	Separate log sources

Tiny Code

```
import logging

logging.basicConfig(level=logging.WARNING)
logging.debug("Hidden")
logging.warning("Visible warning")
```

Why it Matters

Logging is essential for debugging, monitoring, and auditing applications. It helps you understand what your code does in production without spamming users with print statements.

Try It Yourself

1. Write a script that logs an INFO message when it starts and an ERROR when something goes wrong.
2. Change log formatting to include the date and time.
3. Configure logging to write output to a file instead of the console.
4. Create two different loggers: one for `db` and one for `api`, with different log levels.

94. Unit Testing (`unittest`)

Python's `unittest` module provides a framework for writing and running automated tests. It helps you verify that your code works as expected and prevents future changes from breaking existing functionality.

Deep Dive

Basic Test Case

```
import unittest

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

class TestMath(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)

if __name__ == "__main__":
    unittest.main()
```

Running the script:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Common Assertions

Method	Usage Example	Purpose
`assertEqual(a, b)`	`assertEqual(x, 10)`	Check equality
`assertNotEqual(a, b)`	`assertNotEqual(x, 5)`	Check inequality
`assertTrue(x)`	`assertTrue(flag)`	Check condition is True
`assertFalse(x)`	`assertFalse(flag)`	Check condition is False
`assertIn(a, b)`	`assertIn(3, [1,2,3])`	Check membership
`assertRaises(error)`	`with self.assertRaises(ValueError):`	Check exception raised

Testing Exceptions

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

class TestDivide(unittest.TestCase):
    def test_zero_division(self):
        with self.assertRaises(ValueError):
            divide(5, 0)
```

Grouping Multiple Tests

```
class TestStrings(unittest.TestCase):
    def test_upper(self):
        self.assertEqual("hello".upper(), "HELLO")

    def test_isupper(self):
        self.assertTrue("HELLO".isupper())
        self.assertFalse("Hello".isupper())
```

Running Tests

- Run directly:

```
python test_file.py
```

- Or use:

```
python -m unittest discover
```

Quick Summary Table

Feature	Example	Purpose
Test class	`class TestX(unittest.TestCase)`	Group related tests
Assertion methods	`assertEqual`, `assertTrue`	Validate expected behavior
Exception testing	`assertRaises`	Check error handling
Discover tests	`unittest discover`	Auto-run all tests

Tiny Code

```
import unittest

class TestBasics(unittest.TestCase):
    def test_sum(self):
        self.assertEqual(sum([1,2,3]), 6)

if __name__ == "__main__":
    unittest.main()
```

Why it Matters

Unit tests catch bugs early, make code safer to change, and provide confidence that your program works correctly. They are a cornerstone of professional software development.

Try It Yourself

1. Write a function `multiply(a, b)` and a test to check `multiply(2, 5) == 10`.
2. Add a test that verifies dividing by zero raises a `ValueError`.
3. Test that `"python".upper()` returns `"PYTHON"`.
4. Run your tests with `python -m unittest`.

95. Virtual Environments Best Practice

A virtual environment is an isolated Python environment that allows you to install packages without affecting the system-wide Python installation. It's the best practice for managing dependencies in projects.

Deep Dive

Why Use Virtual Environments?

- Avoid conflicts between project dependencies.
- Keep each project self-contained.
- Easier to reproduce the same setup on another machine.

Creating a Virtual Environment

```
python -m venv venv
```

This creates a folder `venv/` that holds the environment.

Activating the Virtual Environment

- Linux / macOS:

```
source venv/bin/activate
```

- Windows (cmd):

```
venv\Scripts\activate
```

After activation, your shell prompt changes, e.g.:

```
(venv) $
```

Installing Packages Inside the environment, install packages as usual:

```
pip install requests
```

Only this environment will have it.

Freezing Requirements Save dependencies to a file:

```
pip freeze > requirements.txt
```

Reinstall them elsewhere:

```
pip install -r requirements.txt
```

Deactivating the Environment

```
deactivate
```

Best Practices

1. Always create a virtual environment for new projects.
2. Use `requirements.txt` for reproducibility.
3. Don't commit the `venv/` folder to version control.
4. Consider tools like pipenv or poetry for advanced dependency management.

Quick Summary Table

Command	Purpose
`python -m venv venv`	Create environment
`source venv/bin/activate`	Activate (Linux/macOS)
`venv\Scripts\activate`	Activate (Windows)
`pip install package`	Install inside environment
`pip freeze > requirements.txt`	Save dependencies
`deactivate`	Exit environment

Tiny Code

```
python -m venv venv
source venv/bin/activate
pip install flask
pip freeze > requirements.txt
```

Why it Matters

Virtual environments prevent dependency chaos. They ensure that one project's libraries don't break another's, making projects portable and maintainable.

Try It Yourself

1. Create a virtual environment called `myenv`.
2. Activate it and install the package `requests`.
3. Run `pip freeze` to confirm the installed package.
4. Deactivate the environment, then reactivate it.

96. Writing a Simple Script

Python scripts are just plain text files with `.py` extension. They can contain functions, logic, and be executed directly from the command line.

Deep Dive

Hello World Script Create a file `hello.py`:

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

Run it:

```
python hello.py
```

Using `if __name__ == "__main__":` This ensures some code only runs when the file is executed directly, not when imported as a module.

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

if __name__ == "__main__":
    print(greet("Alice"))
```

Running `python hello.py` prints:

```
Hello, Alice!
```

But if you import it in another file:

```
import hello
print(hello.greet("Bob"))
```

It won't run the main block automatically.

Accepting Command-Line Arguments Use the `sys` module:

```
import sys

name = sys.argv[1] if len(sys.argv) > 1 else "World"
print(f"Hello, {name}!")
```

Run it:

```
python script.py Alice
# Output: Hello, Alice!
```

Making the Script Executable (Linux/macOS) At the top of the file:

```
#!/usr/bin/env python3
```

Then give permission:

```
chmod +x hello.py
./hello.py
```

Quick Summary Table

Concept	Example	Purpose
Simple script	`print("Hello")`	First step in scripting
Main guard	`if __name__ == "__main__":`	Control script vs import
Command-line arguments	`sys.argv`	Pass input via terminal
Executable script (Unix)	`#!/usr/bin/env python3` + `chmod +x`	Run without `python` prefix

Tiny Code

```
import sys

def square(n: int) -> int:
    return n * n

if __name__ == "__main__":
    num = int(sys.argv[1]) if len(sys.argv) > 1 else 5
    print(f"Square of {num} is {square(num)}")
```

Why it Matters

Scripts turn Python into a tool for automation. With just a few lines, you can create utilities, batch jobs, or prototypes that are reusable and shareable.

Try It Yourself

1. Write a script `greet.py` that prints `"Hello, Python learner!"`.
2. Add a function `double(x)` and use `if __name__ == "__main__":` to call it.
3. Modify the script to accept a number from the command line.
4. Make the script executable on Linux/macOS with a shebang line.

97. CLI Arguments (`argparse`)

Python's `argparse` module makes it easy to build user-friendly command-line interfaces (CLI). Instead of manually reading `sys.argv`, you can define arguments, defaults, help text, and parsing rules automatically.

Deep Dive

Basic Example

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("name")
args = parser.parse_args()

print(f"Hello, {args.name}!")
```

Run:

```
python script.py Alice
# Output: Hello, Alice!
```

Optional Arguments

```
parser = argparse.ArgumentParser()
parser.add_argument("--age", type=int, default=18, help="Your age")
args = parser.parse_args()

print(f"Age: {args.age}")
```

Run:

```
python script.py --age 25
# Output: Age: 25
```

Multiple Arguments

```
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int)
parser.add_argument("y", type=int)
args = parser.parse_args()

print(args.x + args.y)
```

Run:

```
python script.py 5 7
# Output: 12
```

Flags (True/False)

```
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", action="store_true")
args = parser.parse_args()

if args.verbose:
    print("Verbose mode on")
```

Run:

```
python script.py --verbose
# Output: Verbose mode on
```

Quick Summary Table

Feature	Example	Purpose
Positional arg	`parser.add_argument("name")`	Required input
Optional arg	`--age 25`	Extra info with defaults
Type conversion	`type=int`	Enforce types
Flags	`action="store_true"`	On/off switches
Help text	`help="description"`	User guidance

Tiny Code

```
import argparse

parser = argparse.ArgumentParser(description="Square a number")
parser.add_argument("num", type=int, help="The number to square")
args = parser.parse_args()

print(args.num  2)
```

Run:

```
python script.py 6
# Output: 36
```

Why it Matters

With `argparse`, your Python scripts behave like real command-line tools. They're more professional, self-documenting, and easier to use.

Try It Yourself

1. Write a script that accepts a `--name` argument and prints a greeting.
2. Add a `--times` argument that repeats the greeting multiple times.
3. Create a flag `--shout` that prints the greeting in uppercase.
4. Run your script with `-h` to see the auto-generated help message.

98. Working with APIs (`requests`)

APIs let programs talk to each other over the web. Python's `requests` library makes sending HTTP requests simple and readable. You can use it to fetch data, send data, or interact with web services.

Deep Dive

Making a GET Request

```python
import requests

response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
print(response.status_code)   # 200 means success
print(response.json())        # Parse response as JSON
```

Query Parameters

```python
url = "https://jsonplaceholder.typicode.com/posts"
params = {"userId": 1}
response = requests.get(url, params=params)
print(response.json())   # All posts from userId=1
```

POST Request (Send Data)

```python
data = {"title": "foo", "body": "bar", "userId": 1}
response = requests.post("https://jsonplaceholder.typicode.com/posts", json=data)
print(response.json())
```

Handling Errors

```python
response = requests.get("https://jsonplaceholder.typicode.com/invalid")
if response.status_code != 200:
    print("Error:", response.status_code)
```

Headers and Authentication

```python
headers = {"Authorization": "Bearer mytoken"}
response = requests.get("https://api.example.com/data", headers=headers)
```

Quick Summary Table

Method	Example	Purpose
GET	`requests.get(url)`	Retrieve data
POST	`requests.post(url, json=data)`	Send data
PUT	`requests.put(url, json=data)`	Update resource
DELETE	`requests.delete(url)`	Remove resource
params arg	`get(url, params={})`	Add query string
headers arg	`get(url, headers={})`	Set custom headers

Tiny Code

```
import requests

r = requests.get("https://api.github.com")
print("Status:", r.status_code)
print("Headers:", r.headers["content-type"])
```

Why it Matters

APIs are everywhere—from weather apps to payment systems. Knowing how to interact with them lets you integrate external services into your projects.

Try It Yourself

1. Use `requests.get` to fetch JSON from `https://jsonplaceholder.typicode.com/todos/1`.
2. Extract and print the `"title"` field from the response.
3. Send a `POST` request with your own JSON data.
4. Experiment with adding query parameters like `userId=2` to filter results.

99. Basics of Web Scraping (`BeautifulSoup`)

Web scraping means extracting information from websites automatically. In Python, this is commonly done using `requests` to fetch the page and BeautifulSoup (`bs4`) to parse the HTML.

Deep Dive

Installing BeautifulSoup

```
pip install requests beautifulsoup4
```

Fetching a Webpage

```
import requests
from bs4 import BeautifulSoup

url = "https://example.com"
response = requests.get(url)
soup = BeautifulSoup(response.text, "html.parser")
```

Extracting Data

- Get the page title:

```
print(soup.title.string)
```

- Find the first paragraph:

```
print(soup.p.text)
```

- Find all links:

```
for link in soup.find_all("a"):
    print(link.get("href"))
```

Searching by CSS Class

```
soup.find_all("div", class_="article")
```

Practical Example Scraping article headlines:

```
url = "https://news.ycombinator.com"
res = requests.get(url)
soup = BeautifulSoup(res.text, "html.parser")

titles = soup.find_all("a", class_="storylink")
for t in titles[:5]:
    print(t.text)
```

Respect Robots.txt and Rules

- Always check if scraping is allowed (`/robots.txt`).
- Don't overload websites with too many requests.

Quick Summary Table

Method	Example	Purpose
`soup.title.string`	Get title	Page metadata
`soup.p.text`	Get first `<p>` text	Paragraphs
`soup.find_all("a")`	Extract all links	Navigation, references
`soup.find_all("div", class_="x")`	Find elements by class	Structured data extraction

Tiny Code

```
import requests
from bs4 import BeautifulSoup

res = requests.get("https://example.com")
soup = BeautifulSoup(res.text, "html.parser")

print("Title:", soup.title.string)
print("First paragraph:", soup.p.text)
```

Why it Matters

Web scraping lets you automate data collection from websites—useful for research, market analysis, or building datasets when APIs aren't available.

Try It Yourself

1. Scrape the title of `https://example.com`.
2. Extract and print all `<h1>` headers from the page.
3. Collect all links (`href`) on the page.
4. Try scraping a news site (like Hacker News) and print the first 10 headlines.

100. Next Steps: Where to Go from Here

Now that you've mastered the Python flashcards, you have the foundation to build almost anything. The next step is to choose a direction and deepen your skills in areas that interest you most.

Deep Dive

1. Data Science & Machine Learning

- Libraries: `numpy`, `pandas`, `matplotlib`, `scikit-learn`
- Learn to analyze datasets, build models, and visualize results.
- Progress into deep learning with `tensorflow` or `pytorch`.

2. Web Development

- Frameworks: `flask`, `django`, `fastapi`
- Learn to build APIs, web apps, and services.
- Explore front-end integration with JavaScript frameworks.

3. Automation & Scripting

- Use Python to automate repetitive tasks (file handling, Excel reports, web scraping).
- Explore `selenium` for browser automation.

4. Systems & DevOps

- Learn about Python in DevOps: `fabric`, `ansible`, or working with Docker/Kubernetes APIs.
- Use Python for cloud services (AWS, GCP, Azure SDKs).

5. Computer Science Foundations

- Study algorithms and data structures with Python.
- Explore competitive programming and problem-solving platforms (LeetCode, HackerRank).

Learning Pathways

- Books: *Fluent Python*, *Automate the Boring Stuff with Python*, *Python Crash Course*.
- Online platforms: Coursera, edX, freeCodeCamp.
- Open-source projects: contribute on GitHub to gain real experience.

Quick Summary Table

Direction	Libraries / Tools	Example Goal
Data Science	`numpy`, `pandas`, `scikit`	Build a recommendation system
Web Development	`flask`, `django`	Create a blog or API
Automation	`requests`, `selenium`	Automate a daily reporting workflow
DevOps & Cloud	`boto3`, `ansible`	Deploy an app to AWS automatically

Direction	Libraries / Tools	Example Goal
CS Foundations	`heapq`, `collections`	Implement algorithms in Python

Tiny Code (Automation Example)

```
import requests

def get_weather(city):
    url = f"https://wttr.in/{city}?format=3"
    res = requests.get(url)
    return res.text

print(get_weather("London"))
```

Why it Matters

Python is not just a language—it's a gateway. Whether you're interested in AI, finance, web apps, or automating your own life, Python is a tool that grows with you.

Try It Yourself

1. Choose one domain (web, data, AI, automation).
2. Install the relevant libraries (`pip install flask pandas torch`, etc.).
3. Build a small project (e.g., a to-do app, data analysis notebook, or web scraper).
4. Share your project on GitHub to start building a portfolio.