

Study Delight Publications

PYTHON

Simplified

From
First Line
of Code
to Real
Possibilities

Learn
Practice
Build
Grow

PART 1

**Getting Started &
Core Fundamentals**



Simple
Explanations



Hands-on
Examples



Real-World
Practice

Small
Steps
Big Future

Code
Learn
Solve
Create



Clear
Concepts



Practical
Examples



Easy
to Follow



Build
Confidence



Foundation
for AI & More

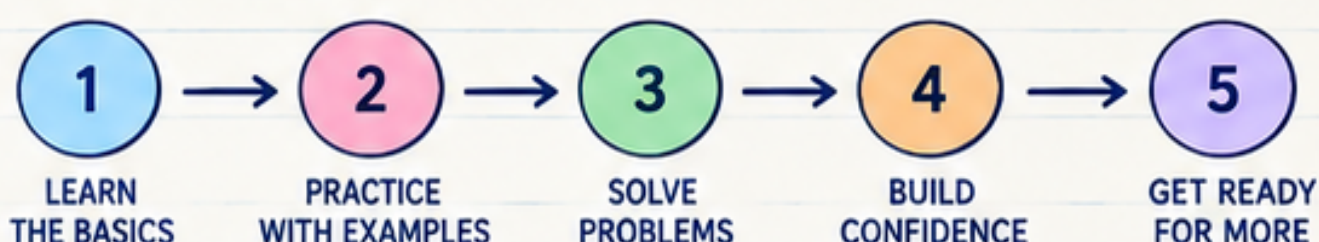
pythonforaiml.com

PYTHON PART 1

Getting Started & Core Fundamentals

Build a strong foundation with Python. Learn the basics, practice with examples, and solve real problems step by step.

20-PAGE ROADMAP



01 What is Python?

02 Why Learn Python?

03 Setting Up Python

04 Your First Python Program

05 Variables and Data Types

06 Type Conversion

07 Input and Output

08 Operators

09 Comments and Code Style

10 Strings

11 String Methods

12 Lists

13 List Methods

14 Tuples

15 Dictionaries

16 Set Basics

17 Conditional Statements

18 Loops

19 Simple Functions

20 Mini Project: A Personal Information App

LEARN • PRACTICE • BUILD • GROW

Use this index as your quick navigation map while studying.

Good
Code
Brighter
Future!



Part 1: Python Foundations

A handwritten, beginner-friendly journey from your first Python statement to functions, collections and a small project.

Goal of Part 1: understand the building blocks well enough to read, write and modify simple Python programs.

Key ideas

- Learn concepts in small pieces.
- Run every example yourself.
- Predict the output before you execute code.
- Practice is more important than memorising syntax.

Remember: Small steps + regular practice = strong foundations.

Quick Check

- What problem does Python solve?
- Name two areas where Python is used.
- Why is readable syntax useful?



Chapter 1: What is Python?

Python is a high-level, general-purpose programming language designed to be readable and productive. It is widely used in automation, web development, data analysis, AI and machine learning.

Key ideas

- Simple and readable syntax
- Large ecosystem of libraries
- Works across major operating systems
- Useful for scripting and automation
- Excellent foundation for AI/ML

Think of Python as a language for expressing instructions clearly. The computer executes those instructions through the Python interpreter.

Practice: Write three reasons you want to learn Python.

Common Mistakes

- Confusing Python with an editor: the interpreter executes code; the editor helps you write it.
- Assuming Python is only for beginners — it is also used for production systems.
- Skipping small experiments and only reading theory.



Chapter 1: Your First Program

The `print()` function displays information on the screen. Start with tiny programs so you can see the relationship between code and output.

Code

```
print("Hello, World!")  
print("Welcome to Python")  
print(10)
```

Output

```
Hello, World!  
Welcome to Python  
10
```

Key ideas

- Text is written inside quotes.
- Numbers can be written without quotes.
- Python is case-sensitive.
- A `print()` call normally ends with a new line.

Remember: Run the program, change one line, and run it again. That feedback loop is how programming becomes easier.

Try Variations

- `print("Your name")`
- `print("Your goal:", "AI")`
- `print(2 + 3 * 4)`



Chapter 1: Comments

Comments are notes for people reading the code. Python ignores comments during execution. Good comments explain intent, decisions or tricky logic.

Code

```
# This is a comment
name = "Asha"
# Display a greeting
print("Hello", name)
```

Output

Hello Asha

Key ideas

- Use # for a single-line comment.
- Keep comments accurate and useful.
- Do not comment every obvious line.

Practice: Add a comment explaining why a variable is needed in one of your programs.

Good Comment vs Bad Comment

- Good: explain why a non-obvious decision was made.
- Weak: repeat exactly what the next line already says.
- Keep comments short enough to stay useful.



Chapter 2: Variables

A variable is a name bound to a value. Assignment creates or updates that name.

Code

```
name = "Asha"  
age = 25  
height = 5.6  
is_student = True  
  
print(name)  
print(age)
```

Output

```
Asha  
25
```

Key ideas

- Use meaningful names: `student_name` is clearer than `x`.
- Names can contain letters, digits and `_`.
- A name cannot start with a digit.
- Use `=` for assignment.

The name is a label; the value is the data associated with that label.

Workflow

- 1. Write a tiny example.
- 2. Run it.
- 3. Observe the output or error.
- 4. Change one thing.
- 5. Run again and explain the result.



Chapter 2: Data Types

A data type tells Python what kind of value it is working with. Common built-in types cover most beginner programs.

Key ideas

- str ☒ text, e.g. 'Python'
- int ☒ whole numbers, e.g. 42
- float ☒ decimal numbers, e.g. 3.14
- bool ☒ True or False
- list ☒ ordered, changeable collection
- tuple ☒ ordered, unchangeable collection
- set ☒ unique values
- dict ☒ key-value pairs

Code

```
name = "Python"
age = 25
print(type(name))
print(type(age))
```

Output

```
<class 'str'>
<class 'int'>
```

Remember: Before choosing an operation, ask: what type of value am I working with?

Naming Checklist

- Prefer total_price over tp.
- Prefer student_count over n when the meaning matters.
- Avoid names such as list, str or input because they shadow useful built-ins.



Chapter 2: Strings

A string is a sequence of characters. Strings can be written with single or double quotes.

Code

```
first = "Python"
second = "Programming"

print(first)
print(first + " " + second)
print(len(first))
```

Output

```
Python
Python Programming
6
```

Key ideas

- + joins strings (concatenation).
- len() counts characters.
- String indexes start at 0.
- Text is case-sensitive.

Practice: Create a message variable containing a complete welcome sentence.

Type Practice

- What is the type of 10?
- What is the type of 10.0?
- What is the type of '10'?
- What is the type of False?



Chapter 2: Indexing & Slicing

Strings are sequences, so individual characters can be accessed with indexes. Python starts counting from zero.

Code

```
word = "PYTHON"

print(word[0])
print(word[2])
print(word[-1])
print(word[1:4])
```

Output

```
P
T
N
YTH
```

Key ideas

- `word[0]` → first character
- `word[-1]` → last character
- `word[start:stop]` → stop is excluded

Remember: Slicing is a quick way to extract a useful part of a sequence.

String Challenge

- Join `first_name` and `last_name` with a space.
- Print a sentence in uppercase.
- Find the length of a word.



Chapter 2: String Methods

String methods solve common text-processing tasks. They return useful results that can be printed or stored.

Code

```
text = " Python Is Great "  
  
print(text.strip())  
print(text.lower())  
print(text.upper())  
print(text.replace("Great", "Powerful"))
```

Output

```
Python Is Great  
python is great  
PYTHON IS GREAT  
Python Is Powerful
```

Key ideas

- `strip()` removes outer whitespace.
- `lower()` makes letters lowercase.
- `upper()` makes letters uppercase.
- `replace(old, new)` replaces matching text.

Remember: Most string methods return a new string rather than changing the original string in place.

Slice Practice

- For 'COMPUTER', what is `word[0]`?
- What is `word[1:5]`?
- What is `word[-1]`?
- Try `word[:4]` and `word[4:]`.



Chapter 2: Numbers & Operators

Python supports arithmetic with integers and floating-point numbers. The main arithmetic operators are shown below.

Code

```
a = 17
b = 5

print(a + b)
print(a - b)
print(a * b)
print(a / b)
print(a // b)
print(a % b)
print(a ** 2)
```

Output

```
22
12
85
3.4
3
2
289
```

Key ideas

- + addition
- - subtraction
- * multiplication
- / division
- // floor division
- % remainder
- ** exponentiation

Practice: Predict `29 % 4` before running it.

Method Match

- | | |
|---|--|
| • Need lowercase? <code>lower()</code> | • Need a replacement? <code>replace()</code> |
| • Need whitespace removed? <code>strip()</code> | • Need character count? <code>len()</code> |



Chapter 2: Input

`input()` reads text entered by the user. The important beginner rule is that `input()` returns a string.

Code

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

Output

```
Enter your name: Asha  
Hello, Asha
```

Code

```
age = int(input("Enter age: "))  
print("Next year:", age + 1)
```

Convert input when you need a number. `int()` and `float()` are common conversions.

Operator Drill

- Calculate $15 + 7$.
- Calculate $15 // 7$.
- Calculate $15 \% 7$.
- Explain why `/` and `//` give different results.



Chapter 2: Type Conversion

Type conversion changes a value from one type to another. It is common when working with input, files and external data.

Code

```
age_text = "25"  
age = int(age_text)  
price = "99.50"  
amount = float(price)  
  
print(age + 5)  
print(amount + 0.50)
```

Output

```
30  
100.0
```

Key ideas

- `int(value)` → integer when valid
- `float(value)` → floating-point number
- `str(value)` → string representation

Watch out: Conversion can fail when the source value is not compatible with the target type.

Practice: Convert the string '42' to an integer and multiply it by 2.

Input Rule

- `input()` gives text.
- Convert numeric input before arithmetic.
- Test your program with both small and large values.



Chapter 2: Comparison Operators

Comparison operators compare values and produce a Boolean result: True or False.

Code

```
age = 20

print(age == 20)
print(age != 18)
print(age > 18)
print(age < 18)
print(age >= 20)
print(age <= 19)
```

Output

```
True
True
True
False
True
False
```

Key ideas

- == equal to
- != not equal to
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

Remember: = assigns a value; == compares values.

Conversion Drill

- int('50') + 10
- float('2.5') * 4
- str(2026)
- Ask yourself which conversions are valid before running them.



Chapter 2: Logical Operators

Logical operators combine Boolean expressions. Python provides `and`, `or` and `not`.

Code

```
age = 25
has_id = True

print(age >= 18 and has_id)
print(age < 18 or has_id)
print(not has_id)
```

Output

```
True
True
False
```

Key ideas

- `and` ☒ both conditions must be true
- `or` ☒ at least one condition is true
- `not` ☒ reverses a Boolean result

Practice: Create a condition that is true only when age is at least 18 AND the user has an ID.

Comparison Drill

- `10 == 10`
- `10 != 9`
- `10 > 12`
- `10 <= 10`
- Predict each Boolean result.



Chapter 3: if — Decisions

Programs often need to choose an action. The if statement runs its indented block only when its condition is true.

Code

```
age = 20

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

Output

Adult

Key ideas

- A colon starts the block.
- Indentation defines the block.
- The condition should evaluate to True or False.

Remember: Indentation is part of Python syntax. Consistent indentation is essential.

Logic Drill

- True and False
- not True
- True or False
- Build a condition for 'adult AND registered'.



Chapter 3: if / else

Use else when there are two main paths: one for a true condition and another for a false condition.

Code

```
marks = 72

if marks >= 40:
    print("Pass")
else:
    print("Try again")
```

Output

Pass

Practice: Change marks to 35. Predict the output before you run the program.

Decision Pattern

- Condition ☒ True? ☒ run the indented block.
- Condition ☒ False? ☒ skip it.
- Use indentation consistently.
- Test both sides of a decision.



Chapter 3: if / elif / else

When several outcomes are possible, use elif between if and else. Python checks the conditions from top to bottom.

Code

```
marks = 84

if marks >= 90:
    grade = "A+"
elif marks >= 75:
    grade = "A"
elif marks >= 60:
    grade = "B"
else:
    grade = "C"

print(grade)
```

Output

A

Key ideas

- The first matching branch runs.
- else is optional.
- Order matters when conditions overlap.

Remember: Put higher-priority or more specific conditions before broader ones.

Decision Pattern

- Two-way decision: if / else.
- Test at least one input for each branch.
- Keep the branches easy to read.



Chapter 3: Nested Conditions

A condition can contain another condition. This is called nesting. Use it when the second decision depends on the first.

Code

```
age = 22
has_ticket = True

if age >= 18:
    if has_ticket:
        print("You may enter")
    else:
        print("Ticket required")
else:
    print("Not eligible")
```

Output

You may enter

If nesting becomes deep, look for a clearer Boolean expression or a function.

Grading Logic

- Start with the highest threshold.
- Check what happens exactly at 90, 75, 60 and 59.
- Remember that only the first matching branch runs.



Chapter 4: for Loops

A for loop repeats a block for each item in an iterable. `range()` is useful when you want a sequence of numbers.

Code

```
for number in range(1, 6):  
    print(number)
```

Output

```
1  
2  
3  
4  
5
```

Key ideas

- `range(1, 6)` produces 1 through 5.
- The stop value is excluded.
- The loop variable receives one value at a time.

Practice: Print the numbers from 10 down to 1.

Simplify It

- Ask whether nested conditions can become one clear condition.
- Use parentheses when they improve readability.
- Move complicated logic into a function when needed.



Chapter 4: while Loops

A while loop repeats as long as its condition remains true. It is useful when the number of repetitions is not known beforehand.

Code

```
count = 1

while count <= 5:
    print(count)
    count += 1
```

Output

```
1
2
3
4
5
```

Watch out: Make sure something changes the condition. Otherwise the loop may never end.

Remember: Trace the condition and the changing variable when debugging a while loop.

Loop Trace

- Before running, write the values you expect.
- Track the loop variable on every iteration.
- Remember that range() excludes its stop value.



Chapter 4: break / continue / pass

These statements change the normal flow of a loop or provide a placeholder.

Code

```
for n in range(1, 8):  
    if n == 3:  
        continue  
    if n == 6:  
        break  
    print(n)
```

Output

```
1  
2  
4  
5
```

Key ideas

- break ☒ stop the loop
- continue ☒ skip the current iteration
- pass ☒ do nothing; useful as a placeholder

Practice: Modify the example so 5 is skipped and the loop stops at 7.

Loop Safety

- Initial value ☒ condition ☒ body ☒ update ☒ condition again.
- Keep loop conditions easy to understand.
- If the update never changes the condition, inspect for an infinite loop.



Chapter 5: Lists

A list stores multiple values in one ordered, changeable collection. Lists are among Python's most useful data structures.

Code

```
fruits = ["apple", "banana", "mango"]  
  
print(fruits[0])  
fruits.append("orange")  
print(fruits)
```

Output

```
apple  
['apple', 'banana', 'mango', 'orange']
```

Key ideas

- Lists use square brackets.
- Indexes start at 0.
- `append()` adds an item at the end.
- Lists are mutable.

Remember: Choose a list when order matters and the collection may change.

Control Flow

- `continue` skips the rest of the current iteration.
- `break` exits the loop.
- `pass` is a placeholder and does not skip an iteration by itself.



Chapter 5: List Operations

Code

```
numbers = [10, 20, 30, 40, 50]

print(len(numbers))
print(numbers[1:4])
print(30 in numbers)
numbers.remove(20)
print(numbers)
```

Output

```
5
[20, 30, 40]
True
[10, 30, 40, 50]
```

Key ideas

- `len(list)` ↗ number of items
- `x in list` ↗ membership test
- `remove(x)` ↗ removes a matching item
- Slicing uses start included, stop excluded.

Practice: Create five cities and print the first and last city.

List Drill

- Create a list of three subjects.
- Append a fourth subject.
- Print the first subject.
- Change one item and print the list again.



Chapter 5: Tuples

A tuple is an ordered collection that cannot be changed after creation. Tuples are useful for fixed groups of related values.

Code

```
point = (10, 20)
print(point[0])
print(point[1])

x, y = point
print(x + y)
```

Output

```
10
20
30
```

Key ideas

- Tuples support indexing and slicing.
- Tuple items cannot be reassigned.
- Unpacking can assign tuple elements to multiple names.

Remember: Use a tuple when the group of values is conceptually fixed.

List Challenge

- Find the middle slice of [10,20,30,40,50].
- Test membership with in.
- Remove one item.
- Explain why lists are useful for collections that change.



Chapter 5: Sets

A set is a collection of unique values. It is useful when you care about membership and removing duplicates.

Code

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

numbers.add(4)
print(2 in numbers)
```

Output

```
{1, 2, 3}
True
```

Key ideas

- Duplicates are removed.
- `add()` inserts an item.
- Membership testing is a common use.
- Sets are not indexed like lists.

Do not rely on the printed order of a set.

Tuple Drill

- Create a tuple for coordinates.
- Access each element.
- Unpack it into `x` and `y`.
- Try changing one element and observe the error.



Chapter 5: Dictionaries

A dictionary stores data as key-value pairs. It is useful when values need meaningful labels rather than numeric positions.

Code

```
student = {  
    "name": "Asha",  
    "age": 21,  
    "course": "Python"  
}  
  
print(student["name"])  
student["age"] = 22  
print(student["age"])
```

Output

```
Asha  
22
```

Key ideas

- Keys identify values.
- Keys are unique within a dictionary.
- Use [] to access a value by key.
- Dictionaries are mutable.

Practice: Create a product dictionary with name, price and category.

Set Drill

- Create a set with duplicate numbers.
- Observe that duplicates disappear.
- Add one new value.
- Test membership with in.



Chapter 6: Functions

A function groups instructions into a reusable unit. Functions reduce repetition and make programs easier to test and maintain.

Code

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

Output

```
Hello, Asha  
Hello, Ravi
```

Key ideas

- def starts a function definition.
- Parameters receive input values.
- The indented block is the function body.
- Call the function using name(...).

Remember: A good function usually has one clear responsibility.

Dictionary Drill

- Create keys: name, age and city.
- Read one value.
- Update age.
- Add a new key called course.



Chapter 6: return & Parameters

A function can return a result instead of printing it. Parameters make the same function useful with different values.

Code

```
def add(a, b):  
    result = a + b  
    return result  
  
total = add(10, 20)  
print(total)
```

Output

30

Key ideas

- `print()` displays a value.
- `return` sends a value back to the caller.
- Parameters make functions reusable.
- Local variables belong to the function's scope.

Practice: Write `square(n)` that returns `n * n`.

Function Design

- Choose a meaningful function name.
- Use parameters instead of hard-coding values.
- Keep one clear responsibility.
- Call the function more than once to test reuse.



Chapter 6: Mini Project — Calculator

Now combine variables, input concepts, conditions and functions into a small calculator. Build one feature at a time and test each branch.

Code

```
def calculate(a, b, op):  
    if op == "+":  
        return a + b  
    elif op == "-":  
        return a - b  
    elif op == "*":  
        return a * b  
    elif op == "/":  
        return a / b  
    else:  
        return "Unknown operator"  
  
print(calculate(10, 5, "*"))
```

Output

50

Key ideas

- Break a problem into small steps.
- Give each function a clear job.
- Test normal cases and edge cases.
- Readable code is easier to debug.

Challenge: Add % and **. Then think about how you would handle division by zero.

Return Drill

- Write add(a,b) with return.
- Store the returned value in total.
- Print total.
- Explain why returning is more reusable than printing inside the function.



Part 1 Review: Python Foundation

You have now covered the core ideas needed to begin writing small Python programs: syntax, variables, types, strings, operators, input, decisions, loops, collections and functions.

Key ideas

- I can read basic Python syntax.
- I can store and convert data.
- I can make decisions with if/elif/else.
- I can repeat work with for and while.
- I can use lists, tuples, sets and dictionaries.
- I can create reusable functions.

Practice: Final practice: ask for a student's name and three marks, calculate the average, choose a grade, store the marks in a list, and create a function for the average.

Remember: Next: Part 2 will go deeper into Python's core tools, data structures, functions and practical problem-solving.

Learn • Practice • Build • Repeat — keep this book beside you while you code.

Project Checklist

- Test +, -, * and /.
- Add a useful message for an unknown operator.
- Think about division by zero.
- Try several inputs, not just one example.

