PYTHON PROGRAMMING

A practical path from your first instruction to useful programs

LEARNSAPPEAL | ORIGINAL COURSE

Learn by writing and improving real code.

Build a strong foundation in Python, then use it to work with files, data, APIs, and reusable projects.

Before you start

A computer and a Python 3 installation are enough. No previous programming experience is required. Type each example, run it, and change one detail at a time. When an error appears, read the last line first and use its message to guide your next check.

Course map: Core building blocks (Modules 1-6), followed by working with data and building projects (Modules 7-12). Includes runnable examples, practice with answers, project briefs, and a glossary.

How to use this course

Move through the modules in order. Create one folder for your experiments and save each program with a .py ending. Run the examples, predict what they will do, then compare your prediction with the result. Make small edits and rerun the program.

By the end, you can: write clear Python programs; use collections and functions; work with errors, files, and data; organize code into modules; test important behavior; and build a small useful application.

A steady rhythm helps. Short, frequent practice is more useful than reading a whole course without running code. Keep your examples and notes so you can revisit and improve them.

Table of contents

• Core building blocks
• 1. Meet Python
• 2. Values, names, and expressions
• 3. Decisions and repetition
• 4. Collections for everyday data
• 5. Functions and reusable steps
• 6. Errors and debugging
• Build and connect
• 7. Files and structured data
• 8. Modules and packages
• 9. Classes and objects
• 10. Work with web APIs
• 11. Testing and clean code
• 12. Capstone projects
• Glossary and next steps

CORE BUILDING BLOCKS

1. Meet Python

Python runs instructions from top to bottom. A program can display a message, calculate a result, or ask for input. Indentation groups code blocks, so spaces are part of Python’s structure. Use four spaces per indentation level and keep code readable.

Try it

```
print('Hello, LearnsAppeal!')
name = input('What is your name? ')
print(f'Welcome, {name}!')
```

Check your understanding

- What does print() do?
- What is input() useful for?
- Why should indentation be consistent?

Suggested answers

1. It displays a value or message.
2. It reads text entered by a person.
3. Indentation marks code blocks and inconsistent spacing can cause errors.

2. Values, names, and expressions

A value is information such as text, a whole number, a decimal number, or True/False. A variable name lets you refer to a value later. Python works out expressions using operators. Use f-strings to place values inside readable messages.

Try it

```
course = 'Python Programming'
lessons = 12
hours_per_lesson = 1.5
total_hours = lessons * hours_per_lesson
print(f'{course}: about {total_hours} hours')
```

Check your understanding

- Name two built-in value types.
- What does the * operator do with numbers?
- What is an f-string for?

Suggested answers

1. Examples: str, int, float, and bool.
2. It multiplies values.
3. It formats text with values inserted into it.

3. Decisions and repetition

Programs use if, elif, and else to choose actions. Comparisons produce True or False. A for loop repeats an action for each item in a sequence; a while loop repeats while a condition remains true. Ensure a while condition eventually changes to avoid an endless loop.

Try it

```
score = 82
if score >= 90:
    result = 'excellent'
elif score >= 70:
    result = 'on track'
else:
    result = 'keep practicing'
print(result)

for number in range(1, 4):
    print(number)
```

Check your understanding

- Which keyword handles a second condition?
- What values does range(1, 4) produce?
- When is a while loop useful?

Suggested answers

1. elif.
2. 1, 2, and 3.
3. When repetition should continue until a condition changes.

4. Collections for everyday data

A list keeps an ordered group of values and can be changed. A tuple is ordered but is usually treated as fixed. A dictionary maps keys to values, which is handy for named details. A set stores unique values. Choose the collection that matches the job.

Try it

```
courses = ['Python', 'Web Development', 'Machine Learning']
courses.append('Data Skills')
learner = {'name': 'Mina', 'completed': 3}
print(learner['name'])
print(len(courses))
```

Check your understanding

- How do you add an item to a list?
- What does a dictionary key help you find?
- What is a set useful for?

Suggested answers

1. Use append().
2. Its associated value.
3. Keeping unique items or checking membership.

5. Functions and reusable steps

A function gives a group of instructions a name. Parameters let it receive input; return sends a result back to the caller. Keep each function focused on one task and choose names that explain what it does. A function can have a default parameter value.

Try it

```
def course_progress(completed, total):
    if total <= 0:
        return 0
    return completed / total * 100

percent = course_progress(3, 12)
print(f'{percent:.0f}% complete')
```

Check your understanding

- What does return do?
- Why use parameters?
- Why check total before dividing?

Suggested answers

1. It sends a result back to the code that called the function.
2. They let the function work with different input values.
3. Division by zero is undefined and causes an error.

6. Errors and debugging

Syntax errors mean Python cannot parse the code. Runtime errors happen while it runs, such as trying to convert invalid text to an integer. A traceback points to where the problem surfaced. Read the error type and message, inspect nearby values, then make one change and run again.

Try it

```
text = '42'
try:
    age = int(text)
except ValueError:
    print('Please enter a whole number.')
else:
    print(f'Next year: {age + 1}')
```

Check your understanding

- What does a traceback help you locate?
- Which exception may int("hello") raise?
- Why change one thing at a time?

Suggested answers

1. The line or call path where an error occurred.
2. ValueError.
3. It helps identify which change fixed or caused the behavior.

BUILD AND CONNECT

7. Files and structured data

Use with open(...) so a file closes automatically, even if something goes wrong. Specify an encoding for text. CSV works well for rows and columns; JSON represents nested data. Handle missing files and invalid data with clear messages rather than letting a confusing failure reach the user.

Try it

```
from pathlib import Path

path = Path('notes.txt')
path.write_text('Practice Python\n', encoding='utf-8')
text = path.read_text(encoding='utf-8')
print(text.strip())
```

Check your understanding

- Why use with when opening a file?
- Why specify UTF-8?
- What kind of data is JSON good at representing?

Suggested answers

1. It ensures the file is closed properly.

2. It makes text encoding predictable across systems.
3. Nested objects, lists, and simple values.

8. Modules and packages

A module is a Python file you can import. Splitting a project into modules keeps related functions together and avoids one enormous file. Put executable startup behavior inside a main function and use the __name__ check so importing the file does not unexpectedly run the program.

Try it

```
# helpers.py
def make_title(topic):
    return f'Learn: {topic}'

# app.py
from helpers import make_title

if __name__ == '__main__':
    print(make_title('Python'))
```

Check your understanding

• What is a Python module?

• Why use an if __name__ == "__main__" check?

• What should a package structure make easy?

Suggested answers

1. A Python file containing code that can be imported.
2. So startup code runs when executed directly, not just imported.
3. Finding related code and understanding the project.

9. Classes and objects

A class describes a kind of object and its behavior. An object stores its own data. Use classes when several related values and actions belong together; for a simple group of fields, a dataclass can reduce repetitive code. Prefer simple functions when no object state is needed.

Try it

```
from dataclasses import dataclass

@dataclass
class Course:
    title: str
    lessons: int

    def summary(self):
        return f'{self.title}: {self.lessons} lessons'

course = Course('Python', 12)
print(course.summary())
```

Check your understanding

• What does a class describe?

• What does @dataclass provide?

• When might a function be simpler than a class?

Suggested answers

1. A kind of object, including its data and behavior.
2. Convenient initialization and common data methods.
3. When the work does not need stored object state.

10. Work with web APIs

An API lets programs exchange information. A client sends a request and receives a response, often in JSON. Check the response status, handle timeouts and network errors, and avoid exposing secret keys in code sent to a browser. Respect API usage rules and avoid requesting data too often.

Try it

This example uses the requests package. Install it in your project environment before running the snippet.

```
# Requires: python -m pip install requests
import requests

response = requests.get(
    'https://api.github.com/repos/python/cpython', timeout=10
)
response.raise_for_status()
repo = response.json()
print(repo['full_name'], repo['stargazers_count'])
```

Check your understanding

- What does JSON provide?
- Why set a timeout?
- What does raise_for_status() do?

Suggested answers

1. A structured format for exchanging data.
2. So a request does not wait forever.
3. It raises an error for unsuccessful HTTP status responses.

11. Testing and clean code

Tests describe behavior that should stay correct. Start with small functions and test normal inputs, edge cases, and invalid values. Clear names, short functions, and consistent formatting make code easier to review. A virtual environment keeps each project's installed packages separate.

Try it

```
def add_tax(price, rate):
    if price < 0 or rate < 0:
        raise ValueError('price and rate must be non-negative')
    return price * (1 + rate)

assert add_tax(100, 0.1) == 110
assert add_tax(0, 0.2) == 0
```

Check your understanding

- What is an edge case?
- Why test invalid inputs?
- What problem does a virtual environment solve?

Suggested answers

1. An unusual boundary input, such as zero or an empty list.
2. To make failure behavior deliberate and understandable.
3. It isolates a project's installed packages from other projects.

12. Capstone projects

Choose one project and build it in small steps. First write down what a person should be able to do. Then implement the simplest working version, test it, and add features. Save a README with setup steps, example usage, and limitations.

Try it

```
# Project idea: study tracker
# 1. Store course names and progress in JSON.
# 2. Add commands to list, add, and update courses.
# 3. Validate progress from 0 to 100.
# 4. Handle a missing or malformed data file.
# 5. Add tests and document how to run it.
```

Check your understanding

- Why build a small version first?
- Name two cases the study tracker should handle.
- What should the README include?

Suggested answers

1. It gives you something working to test and improve safely.
2. Missing data file, invalid percentage, empty course list, or malformed JSON.
3. Setup steps, usage examples, and known limitations.

Project workshop

1. Personal expense summary

Read a CSV file of expenses. Group totals by category, handle blank or invalid amounts, and print a clear summary. Add a small test dataset and document how to run the program.

2. Learner course tracker

Create a command-line tool to add courses, record progress, list incomplete courses, and save data as JSON. Validate input, handle a missing file, and write tests for the progress rules.

3. Public API report

Choose a public API with clear usage terms. Request a small amount of data with a timeout, handle request failures, select useful fields, and display a readable report. Do not place private credentials in source code.

Project checklist
Can another person understand the setup? Are invalid inputs handled? Does the program explain failures? Are important behaviors tested? Are secrets and personal data kept safe?

Quick reference

Loop through a list

```
for item in items:
    print(item)
```

Define and call a function

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

message = greet('Sam')
```

Read JSON

```
import json
from pathlib import Path
data = json.loads(Path('data.json').read_text(encoding='utf-8'))
```

Run tests

```
python -m unittest discover
```

Glossary

Term	Meaning
Argument	A value supplied when calling a function.
Boolean	A True or False value.
Class	A definition for creating objects with related data and behavior.
Exception	An event that interrupts normal execution and can be handled.
Function	A named, reusable group of instructions.
List	An ordered, changeable collection.
Module	A Python file that can be imported.
Object	A value with data and behavior.
Parameter	A named input accepted by a function.
Package	A way to organize related Python modules.
Test	A check that verifies expected program behavior.
Virtual environment	An isolated place for a project's Python packages.

Next steps

Use the official Python tutorial and language documentation as your reference when exploring new features. Practice by improving one project: add a feature, write a test, and update the README.

• Python Software Foundation. *The Python Tutorial.* https://docs.python.org/3/tutorial/

• Python Software Foundation. *Python Standard Library.* https://docs.python.org/3/library/

• Python Packaging Authority. *Python Packaging User Guide.* https://packaging.python.org/

Course completion checklist

I can write a Python program with clear variables and functions; choose useful collections; make decisions and repeat work; handle errors; work with files and APIs; organize modules; test expected behavior; and explain how to run and improve a small project.