How Python Code Runs

Understanding the Interpreter, Execution Process and Python Files

Category: Python Development

haas.dev

https://dev-roast-app.vercel.app

---

1. Before You Start Writing More Python

You now know:

- what Python is
- why Python is used
- how Python fits into software development
- how to install Python
- how to set up VS Code
- how to create a Python project
- how to run a `.py` file from the terminal

But there is an important question underneath all of that:

> **What actually happens when you run a Python file?**

You write:

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

Then you run:

```
python hello.py
```

And somehow the computer produces:

```
Hello, Python!
```

What happened between those two points?

Understanding this process gives you a much stronger foundation for everything that comes later.

You do not need to become a Python interpreter developer.

But you should understand the basic execution model.

---

2. Learning Objectives

By the end of this guide, you should be able to:

- Explain what Python source code is
- Explain what a `.py` file represents
- Understand what happens when Python starts
- Understand the role of the Python interpreter
- Understand the difference between source code and execution
- Understand the basic idea of parsing
- Understand what bytecode is
- Understand the role of the Python Virtual Machine
- Understand what CPython means
- Understand how Python executes a script
- Understand interactive execution vs file execution
- Understand what happens when Python encounters an error
- Understand imports at a basic execution level
- Understand why Python can feel interpreted while also involving compilation
- Debug simple execution problems using the correct mental model

---

3. The Simplest Mental Model

Start with this:

```
Python File
    ↓
Python Runtime
    ↓
Execution
    ↓
Output
```

Suppose you create:

```
hello.py
```

with:

```
print("Hello")
```

Then:

```
python hello.py
```

The operating system starts the Python runtime.

Python receives your source code.

It processes the code.

Then the program executes.

Finally, the result appears in the terminal.

```
hello.py
   ↓
Python
   ↓
Process Code
   ↓
Execute Instructions
   ↓
Output
```

This is the basic model.

Now we can go deeper.

---

4. What Is Source Code?

Source code is the code written by a developer.

For example:

```
name = "Hafsa"
print(name)
```

This is readable by humans who understand Python.

The file might be:

```
main.py
```

The `.py` file contains Python source code.

Source code is the starting point of the execution process.

---

5. Source Code Is Not Machine Code

Your processor does not directly execute:

```
print("Hello")
```

as Python syntax.

A CPU ultimately executes machine-level instructions.

There is therefore a transformation and execution process between:

```
Human-readable Python
```

and:

```
Computer execution
```

A simplified model is:

```
Python Source Code
        ↓
Processing
        ↓
Python Bytecode
        ↓
Python Runtime
        ↓
Machine-level execution
```

This is why saying:

> "The computer directly executes my `.py` file"

is an oversimplification.

The Python runtime is involved.

---

6. What Is the Python Interpreter?

The term **interpreter** is often used to describe the software that executes Python code.

When you run:

```
python main.py
```

you are launching the Python runtime and asking it to execute `main.py`.

Conceptually:

```
Terminal
   ↓
python main.py
   ↓
Python Runtime
   ↓
main.py
```

```
    ↓
Execution
```

The interpreter is therefore not your `.py` file.

It is the software that processes and executes Python code.

---

7. The Python Runtime

The runtime is the environment responsible for running your Python program.

It provides the machinery needed to:

- load Python code
- process it
- manage objects
- execute instructions
- manage memory
- interact with the operating system
- load modules
- handle exceptions

You normally do not think about these mechanisms while writing simple Python programs.

But they are working underneath your code.

---

8. What Happens When You Run a Python File?

Consider:

```
x = 10
y = 20

print(x + y)
```

You save it as:

```
calculator.py
```

Then run:

```
python calculator.py
```

A simplified execution flow is:

```
calculator.py
      ↓
```

```
Python loads source code
      ↓
Python parses the code
      ↓
Python compiles it to bytecode
      ↓
Python runtime executes the bytecode
      ↓
print() produces output
      ↓
30
```

This is a simplified model of what happens in the commonly used CPython implementation.

---

9. Step 1: Python Loads Your File

When you run:

```
python calculator.py
```

Python needs to locate and read the file.

The operating system first launches the Python executable.

Python then receives the path to your file.

Conceptually:

```
Operating System
       ↓
Start Python
       ↓
Give Python calculator.py
       ↓
Python reads calculator.py
```

If the file cannot be found, execution cannot continue.

For example:

```
python: can't open file ...
```

This is not a Python logic problem.

It is a file/path problem.

That distinction matters when debugging.

---

10. Step 2: Python Reads the Source

Python reads the contents of the file.

For example:

```
name = "Hafsa"
print(name)
```

At this stage, Python is dealing with source code.

It needs to understand whether the code follows Python's grammatical and syntactical rules.

That leads us to parsing.

---

11. Step 3: Parsing

A programming language has rules.

For example:

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

follows Python's syntax.

But:

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

is incomplete.

Python needs to analyze the structure of the source code.

This process is called **parsing**.

A parser examines the source and determines whether it follows the language's grammar.

Conceptually:

```
Source Code
    ↓
Parser
    ↓
Valid structure?
```

If the source contains invalid syntax, Python reports an error.

---

12. Syntax Errors

Consider:

```
print("Hello"
```

The parentheses are incomplete.

Python cannot correctly understand the structure.

You may receive a `SyntaxError`.

The important idea is:

```
Source Code
     ↓
Parsing
     ↓
Invalid syntax
     ↓
SyntaxError
```

The program does not reach normal execution.

This is different from a runtime error.

---

13. Syntax Error vs Runtime Error

These two errors are extremely important.

Syntax Error

The code violates Python's syntax rules.

Example:

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

The structure is invalid.

Python cannot properly interpret the program.

Runtime Error

The code is syntactically valid but something goes wrong while the program is running.

Example:

```
number = 10
result = number / 0
```

Python understands the structure.

But division by zero is invalid during execution.

This can produce:

```
ZeroDivisionError
```

The distinction is:

```
Syntax Error
→ Problem before normal execution

Runtime Error
→ Problem during execution
```

You will study errors and exceptions in depth later.

---

14. Step 4: Compilation to Bytecode

After Python successfully processes the source code, CPython compiles it into **bytecode**.

This does not mean Python turns your program directly into native CPU machine code in the same way that a traditional ahead-of-time compiled language might.

Instead, CPython creates an intermediate representation called bytecode.

Conceptually:

```
Python Source
      ↓
Compiler
      ↓
Bytecode
```

This bytecode is designed for execution by Python's runtime machinery.

---

15. What Is Bytecode?

Bytecode is an intermediate form of instructions.

It is not the same thing as:

- Python source code
- CPU machine code

Think of it as a middle layer.

```
Python Source Code
        ↓
     Bytecode
```

```
        ↓
 Python Runtime
```

This helps explain why Python execution is more nuanced than simply saying:

> "Python is interpreted."

Python implementations can involve both compilation and interpretation/runtime execution.

---

16. Why Do People Still Call Python an Interpreted Language?

You may hear:

> "Python is an interpreted language."

This is a useful beginner-level description, but it is not the complete technical picture.

CPython first compiles Python source into bytecode and then executes that bytecode through its runtime.

So a more precise simplified model is:

```
Source Code
    ↓
Compile to Bytecode
    ↓
Execute Bytecode
```

Different Python implementations can work differently.

Therefore, avoid thinking:

> "Interpreted means there is absolutely no compilation."

That is an oversimplification.

---

17. The Python Virtual Machine

The **Python Virtual Machine**, often abbreviated PVM, is the conceptual runtime machinery that executes Python bytecode.

Think of the flow as:

```
.py File
   ↓
Python Source
   ↓
```

```
Bytecode
   ↓
Python Virtual Machine
   ↓
Program Behavior
```

You normally do not interact with the PVM directly.

It works behind the scenes.

But understanding that it exists gives you a better mental model of Python execution.

---

18. A More Complete Execution Model

Now combine the pieces:

```
              Your Code
                 │
                 ▼
           Python Source
                 │
                 ▼
              Parsing
                 │
                 ▼
        Compilation to Bytecode
                 │
                 ▼
              Bytecode
                 │
                 ▼
       Python Runtime / PVM
                 │
                 ▼
            Execution
                 │
                 ▼
         Operating System
                 │
                 ▼
              Hardware
```

This is simplified, but it is a useful conceptual model.

---

19. What Is CPython?

CPython is the most widely used implementation of Python and the reference implementation associated with the official Python project.

It is implemented primarily in C.

When you install Python from the standard Python distribution, you are typically installing CPython.

Therefore:

```
Python
= language

CPython
= a major implementation of that language
```

This distinction becomes useful when you encounter terms such as:

- CPython
- PyPy
- Jython
- IronPython

These are different implementations or environments associated with Python.

---

20. Why Are There Different Python Implementations?

Python is a language specification and ecosystem.

Different implementations can make different engineering choices.

For example, an implementation may focus on:

- compatibility
- performance
- integration with another runtime
- specialized environments

You do not need to learn every implementation as a beginner.

For everyday Python development, CPython is the natural starting point.

---

21. The Interactive Interpreter Works Differently From a Script

Earlier you used:

```
python
```

and saw:

```
>>>
```

This is the interactive interpreter.

You can type:

```
>>> 5 + 5
10
```

and Python immediately evaluates the expression.

Compare that with a file:

```
# main.py

x = 5
y = 5

print(x + y)
```

Then:

```
python main.py
```

The two experiences are:

```
Interactive Mode
→ Enter code
→ Execute
→ See result
```

and:

```
Script Mode
→ Write file
→ Save file
→ Run file
→ Execute program
```

---

22. When Should You Use Interactive Mode?

The interactive interpreter is useful for small experiments.

For example:

```
>>> 100 * 0.2
20.0
```

You can quickly test:

- calculations
- expressions
- small pieces of logic
- library behavior
- object properties

It is especially useful when learning.

But larger programs belong in files.

---

23. When Should You Use `.py` Files?

Use files when you need:

- reusable code
- larger programs
- version control
- testing
- collaboration
- project structure
- debugging
- deployment

For example:

```
calculator.py
```

can be saved, edited, tested, committed to Git, and shared with another developer.

Interactive commands disappear when you exit unless you explicitly save them elsewhere.

---

24. Why Python Does Not Need You to Compile Manually

In languages such as C or C++, developers often have explicit compilation steps.

For example:

```
Source Code
   ↓
```

```
Compiler
   ↓
Executable
   ↓
Run
```

Python hides much of the compilation machinery from you.

You can simply run:

```
python main.py
```

Python handles the necessary processing.

This is one reason Python feels convenient during development.

You do not normally run a separate command such as:

```
compile-python
```

before every execution.

---

25. What Happens If You Change the Code?

Suppose:

```
print("Hello")
```

You run it:

```
Hello
```

Then change it to:

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

Run again:

```
Hello, Python!
```

Python processes the updated source again.

Conceptually:

```
Edit
 ↓
Save
 ↓
Run
 ↓
```

```
Process Updated Source
 ↓
Execute
```

This is why developers can rapidly experiment with Python.

---

26. What Are `.pyc` Files?

You may eventually encounter files such as:

```
__pycache__/
    something.cpython-3xx.pyc
```

These contain compiled Python bytecode associated with modules.

You should not confuse them with your original source code.

You write:

```
module.py
```

Python may create cached bytecode under:

```
__pycache__/
```

The exact behavior depends on how Python is executing the module.

---

27. Why Does Python Create `__pycache__`?

When Python imports modules, it can cache compiled bytecode to improve subsequent loading in appropriate circumstances.

Conceptually:

```
module.py
   ↓
Compile
   ↓
Bytecode
   ↓
Cache
```

On a later import, Python may be able to reuse valid cached bytecode instead of doing all compilation work from scratch.

The cache is an implementation detail.

You generally should not manually edit these files.

---

28. Should You Commit `__pycache__` to Git?

Normally, generated Python cache directories such as:

```
__pycache__/
```

are excluded from version control.

A typical `.gitignore` entry may include:

```
__pycache__/
*.pyc
```

The reason is simple:

These files are generated artifacts, not the source code you normally need to share.

Your source files are the important project assets.

---

29. Source Code vs Bytecode

Keep this distinction clear.

Source Code

Human-written Python:

```
print("Hello")
```

Usually stored in:

```
hello.py
```

Bytecode

An intermediate representation produced by the Python implementation.

Often associated with:

```
__pycache__
```

Machine-Level Execution

The actual lower-level operations carried out by the computer.

Conceptually:

```
.py
 ↓
Bytecode
 ↓
Runtime
```

```
 ↓
Machine execution
```

---

30. Does Python Compile Everything Before Running?

For a normal Python script under CPython, Python processes the code before executing it.

That does not mean the entire development experience works exactly like a traditional compiled language.

Python can execute code interactively.

It can dynamically load modules.

It can generate objects and behavior at runtime.

It can evaluate certain constructs dynamically.

Therefore, a useful beginner model is:

```
Python source
→ processed/compiled to bytecode
→ executed by Python runtime
```

Do not overgeneralize that model to every Python implementation.

---

31. Imports Add Another Layer

Consider:

```
import math

print(math.sqrt(25))
```

Python needs to find the `math` module.

The execution process therefore includes module loading.

Conceptually:

```
main.py
  ↓
import math
  ↓
Find module
  ↓
Load module
  ↓
Make it available
```

```
  ↓
Execute remaining code
```

Imports become very important when building real applications.

---

32. What Happens When a Module Is Imported?

A Python module can contain executable top-level code.

For example:

```
# greeting.py

print("Greeting module loaded")
```

Then:

```
import greeting
```

can cause:

```
Greeting module loaded
```

to appear.

This is because importing a module involves executing its top-level code as part of module initialization.

This is one reason professional Python developers think carefully about what code belongs at module level.

---

33. A Better Module Pattern

Instead of putting everything directly at the top level:

```
print("Application started")
```

larger applications often organize execution through functions.

For example:

```
def main():
    print("Application started")


if __name__ == "__main__":
    main()
```

This pattern is extremely common in Python.

The meaning of:

```
if __name__ == "__main__":
```

will be covered more deeply when you study modules.

For now, understand the basic idea:

A Python file can behave differently when run directly versus imported by another file.

---

34. Running a File vs Importing a File

Consider:

```
main.py
utils.py
```

You can run:

```
python main.py
```

Or `main.py` can contain:

```
import utils
```

These are different operations.

Run

```
python main.py
```

means:

> Start this file as the main program.

Import

```
import utils
```

means:

> Make the `utils` module available to this program.

Understanding this distinction becomes essential as projects grow.

---

35. What Happens When Python Finds an Error?

Suppose you write:

```
x = 10
print(x)
print(y)
```

The first `print()` can execute.

Then Python reaches:

```
print(y)
```

If `y` does not exist, Python raises a `NameError`.

Conceptually:

```
Start
 ↓
x = 10
 ↓
print(x)
 ↓
Output 10
 ↓
print(y)
 ↓
NameError
 ↓
Execution stops
```

This illustrates an important principle:

Python executes a program and can stop when an unhandled error occurs.

Later you will learn how exceptions can be handled intentionally.

---

36. Error Messages Are Information

Suppose Python reports:

```
NameError: name 'y' is not defined
```

Do not treat the message as random noise.

It tells you:

- the type of error
- the name Python could not resolve
- often the line where the problem occurred

Professional debugging starts by reading the error.

A good debugging process is:

```
Read Error
   ↓
Find Line
   ↓
Understand Error Type
   ↓
Inspect Relevant Code
   ↓
Identify Cause
   ↓
Fix Cause
   ↓
Run Again
```

---

37. The Execution Order Matters

Consider:

```
print("A")
print("B")
print("C")
```

Python executes the statements in order:

```
A
↓
B
↓
C
```

Output:

```
A
B
C
```

This seems obvious.

But the same principle becomes extremely important when programs contain:

- conditions
- loops
- functions

- imports
- exceptions
- asynchronous operations

Understanding execution order is one of the foundations of programming.

---

38. Code Is a Sequence of Instructions

Consider:

```
price = 100
quantity = 3
total = price * quantity

print(total)
```

A simplified execution sequence is:

```
1. Create/store 100
2. Create/store 3
3. Calculate 100 × 3
4. Store result
5. Call print()
6. Display result
```

The computer is not "understanding the story."

It is executing instructions according to the rules of the language and runtime.

---

39. Python Objects Exist During Execution

Python programs work heavily with objects.

For example:

```
name = "Hafsa"
```

creates a reference to a string object.

Similarly:

```
age = 25
```

involves an integer object.

A simplified mental model is:

```
Variable Name
      ↓
Reference
      ↓
Object
```

You will study variables, objects, memory, mutability, and references later.

For now, understand that Python execution involves creating and manipulating objects.

---

40. Python Manages Memory

You usually do not manually allocate and release memory for ordinary Python objects.

Python's runtime manages memory for you.

For example:

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

Python creates the necessary objects and manages their memory.

When objects are no longer needed, Python's memory management system can eventually reclaim their memory.

This is one reason Python allows developers to focus on higher-level programming rather than manually managing every memory allocation.

---

41. Garbage Collection

Python has automatic memory management and garbage collection mechanisms.

You will not normally write:

```
free(object)
```

for ordinary Python objects.

Instead, Python manages object lifetimes through mechanisms such as reference counting in CPython, along with cyclic garbage collection.

At your current stage, the key idea is:

```
Create Object
   ↓
Use Object
   ↓
Object No Longer Needed
```

```
   ↓
Python Manages Its Memory
```

Memory management becomes more important when you study performance and advanced Python.

---

42. Why Understanding Execution Helps You Debug

Suppose this program behaves unexpectedly:

```
x = 10

if x > 5:
    print("A")

print("B")
```

If you understand execution order, you can predict:

```
x = 10
 ↓
10 > 5
 ↓
True
 ↓
Print A
 ↓
Print B
```

Output:

```
A
B
```

Without an execution model, beginners often guess.

With an execution model, you reason.

That is the difference between:

> "Why did Python do this?"

and:

> "I can trace why Python did this."

---

43. Trace Execution Manually

One of the best beginner skills is **manual tracing**.

Consider:

```
x = 5
y = 2
z = x + y

print(z)
```

Trace it:

```
x = 5
y = 2
z = 5 + 2
z = 7
print(7)
```

Output:

```
7
```

This technique becomes extremely useful for:

- conditions
- loops
- functions
- nested structures
- debugging

---

44. Execution Trace Example

Consider:

```
age = 20

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

Trace:

```
age = 20
     ↓
```

```
age >= 18
     ↓
20 >= 18
     ↓
True
     ↓
Execute if block
     ↓
Print "Adult"
```

Output:

```
Adult
```

Notice that Python does not execute both branches.

The condition determines which path is taken.

---

45. Why This Matters for Future Topics

Soon you will learn:

- `if`
- `elif`
- `else`
- `for`
- `while`
- functions
- exceptions

All of these modify the basic execution flow.

For example:

```
Normal execution
       ↓
Condition
   ↙       ↘
True      False
 ↓          ↓
Path A     Path B
```

Loops create:

```
Repeat
 ↓
Check
 ↓
Execute
 ↓
Repeat
```

Functions create:

```
Call
 ↓
Enter function
 ↓
Execute
 ↓
Return
 ↓
Continue
```

The concepts you learn here will support all of them.

---

46. Why Python Feels Interactive

Python's interactive nature is one of its useful characteristics.

You can start:

```
python
```

Then immediately test:

```
>>> 2 ** 10
1024
```

There is no need to create an entire application just to test one expression.

This makes Python useful for:

- learning
- exploration
- data analysis
- debugging
- experimentation

It also encourages a productive workflow:

```
Question
 ↓
Small Experiment
 ↓
Observe Result
 ↓
Understand Behavior
 ↓
Implement in Project
```

---

47. The Difference Between Code and Program

These terms are often used interchangeably, but there is a useful distinction.

Code

Instructions written by a developer.

Example:

```
print("Hello")
```

Program

A set of code and related components that performs a task.

Example:

```
Expense Tracker
```

may contain:

```
main.py
database.py
models.py
reports.py
tests/
```

A single line of code is not necessarily a complete application.

---

48. From Python File to Application

Your first project may be:

```
hello.py
```

Later:

```
calculator/
├── main.py
├── operations.py
└── tests/
```

Later still:

```
web_app/
├── app/
├── tests/
├── configuration/
├── database/
├── requirements/
└── deployment/
```

The execution model remains rooted in Python code.

The architecture becomes more sophisticated as the project grows.

---

49. What Happens When You Run a Large Application?

A simplified model:

```
Start Application
      ↓
Load Configuration
      ↓
Load Python Modules
      ↓
Initialize Components
      ↓
Connect to Services
      ↓
Start Application Logic
      ↓
Wait for Requests / Tasks
      ↓
Process Work
```

For a web server:

```
Client
  ↓
HTTP Request
```

```
  ↓
Python Application
  ↓
Route
  ↓
Business Logic
  ↓
Database
  ↓
Response
```

The same Python runtime concepts remain underneath.

---

50. A haas.dev Example

Imagine haas.dev has a Python backend that serves resource information.

A user requests:

```
GET /resources
```

A simplified execution flow could be:

```
Browser
   ↓
HTTP Request
   ↓
Python Web Application
   ↓
Route Handler
   ↓
Resource Service
   ↓
Database
   ↓
Python Objects
   ↓
JSON Response
   ↓
Browser
```

The Python runtime is executing the code that coordinates this process.

This is how a simple programming language becomes part of a real product architecture.

---

51. A Web Application Does Not Run "One Line at a Time" Forever

For a simple script, you can imagine:

```
Line 1
 ↓
Line 2
 ↓
Line 3
 ↓
Finish
```

A long-running web application behaves differently.

It may:

```
Start
 ↓
Initialize
 ↓
Wait
 ↓
Receive request
 ↓
Process request
 ↓
Return response
 ↓
Wait
 ↓
Receive another request
 ↓
Process again
```

This is why understanding execution flow becomes more important as applications become interactive.

---

52. What You Should Not Worry About Yet

At this point, you do **not** need to memorize:

- Python bytecode instructions

- CPython's C source code
- parser implementation details
- garbage collector internals
- interpreter optimization techniques
- virtual machine internals

Those topics are useful later for advanced Python development.

Right now, you need the correct mental model.

```
Source
 ↓
Process
 ↓
Bytecode
 ↓
Runtime
 ↓
Execution
```

That is enough.

---

53. Common Misconceptions

"Python directly executes the `.py` file."

Not quite.

The Python runtime processes the source and, in CPython, compiles it to bytecode before executing it.

"Python has no compiler."

Oversimplified.

CPython compiles source code into bytecode.

"Bytecode is machine code."

No.

Python bytecode is an intermediate representation executed by Python's runtime.

"The interpreter is my code editor."

No.

VS Code is a code editor.

The Python interpreter/runtime executes Python code.

"A `.pyc` file is my original Python program."

No.

It is cached compiled bytecode associated with Python modules.

The source remains your `.py` file.

"If my code has no syntax errors, it will always work."

No.

A syntactically valid program can still have:

- runtime errors
- incorrect logic
- unexpected input
- dependency problems
- network failures
- database failures

Correct syntax is only one requirement for a working program.

---

54. Debugging Through the Execution Model

When your program fails, ask these questions in order.

Question 1

Did Python find the file?

```
Path problem?
```

Question 2

Can Python parse the source?

```
Syntax problem?
```

Question 3

Did the program start executing?

```
Runtime problem?
```

Question 4

What line failed?

```
Traceback
```

Question 5

What state existed immediately before the failure?

```
Variables
Inputs
Conditions
Objects
```

Question 6

What was the program supposed to do?

```
Expected behavior
```

Question 7

What did it actually do?

```
Actual behavior
```

This is a professional debugging mindset.

---

55. Five Minute Execution Challenge

Create:

```
execution_test.py
```

Write:

```
print("Start")

x = 10
y = 5

print("Before calculation")

result = x + y

print("Result:", result)

print("End")
```

Before running it, predict the exact output.

Then run it.

Afterward, manually trace:

```
Start
 ↓
x = 10
 ↓
y = 5
 ↓
Before calculation
 ↓
result = 15
 ↓
Result: 15
 ↓
End
```

Now change:

```
result = x + y
```

to:

```
result = x / 0
```

Run it again.

Observe where execution stops.

This is your first execution-flow debugging exercise.

---

56. Practice Exercises

Exercise 1 — Predict the Output

What will this program print?

```
x = 5
y = 10

print(x)
print(y)
print(x + y)
```

Write your answer before running it.

Exercise 2 — Trace the Program

Trace:

```
a = 10
b = 2
c = a * b

print(c)
```

Write the value of:

```
a =
b =
c =
```

before the `print()` executes.

Exercise 3 — Find the Syntax Problem

Identify the problem:

```
name = "Hafsa"
print(name
```

What kind of error should you expect?

Exercise 4 — Find the Runtime Problem

Identify the problem:

```
number = 10
result = number / 0
print(result)
```

Why can Python parse the program but fail during execution?

Exercise 5 — Execution Order

Predict the output:

```
print("One")

x = 5

print("Two")
```

```
x = x + 10

print(x)
```

Then run it and compare your prediction.

---

57. Mini Quiz

Question 1

What is Python source code?

A. CPU machine code

B. Human-written code using Python syntax

C. A database file

D. A VS Code extension

Answer: B

Question 2

What is bytecode?

A. A type of database

B. Python source code

C. An intermediate representation used by Python implementations such as CPython

D. HTML code

Answer: C

Question 3

What does the Python runtime do?

A. Only edit source files

B. Execute Python programs and provide runtime functionality

C. Replace the operating system

D. Design user interfaces

Answer: B

Question 4

Which sequence best represents the simplified CPython execution model?

A. Source → Bytecode → Runtime execution

B. Source → HTML → Browser

C. Source → Database → CSS

D. Source → Image → Monitor

Answer: A

Question 5

What is the difference between a syntax error and a runtime error?

A. They are identical

B. Syntax errors involve invalid program structure, while runtime errors occur while executing validly parsed code

C. Runtime errors only happen in JavaScript

D. Syntax errors happen only after deployment

Answer: B

---

58. Interview Questions

What happens when you run a Python file?

In a simplified CPython model, Python loads the source code, parses it, compiles it into bytecode, and executes that bytecode through the Python runtime.

Is Python interpreted or compiled?

Python is commonly described as interpreted, but the precise answer depends on the implementation. CPython compiles source code into bytecode before executing it through its runtime.

What is Python bytecode?

Bytecode is an intermediate representation generated from Python source code and executed by the Python runtime.

What is CPython?

CPython is the most widely used implementation of the Python programming language and the reference implementation associated with the Python project.

What is the Python Virtual Machine?

It is a conceptual name for the runtime machinery that executes Python bytecode.

What is the difference between `.py` and `.pyc`?

`.py` files contain Python source code. `.pyc` files contain cached compiled bytecode associated with Python modules.

Why does Python create `__pycache__`?

It can store cached bytecode for imported modules, allowing Python to avoid recompiling unchanged source in situations where the cached bytecode can be reused.

What happens when Python encounters a syntax error?

Python cannot successfully parse the source, so normal execution does not begin.

What happens during a runtime error?

The program has begun executing, but an operation fails and raises an exception. If the exception is not handled, execution stops.

---

59. Execution Cheat Sheet

```
.py
↓
Python source code
```

```
Parser
↓
Checks source structure
```

```
Compiler
↓
Produces bytecode in CPython
```

```
Bytecode
↓
Intermediate representation
```

```
Python Runtime / PVM
↓
Executes bytecode
```

```
Output / Program Behavior
```

Run a script

```
python main.py
```

Start interactive Python

```
python
```

Check version

```
python --version
```

Inspect current executable

```
import sys
print(sys.executable)
```

Inspect Python version

```
import sys
print(sys.version)
```

---

60. The Execution Mental Model

Whenever you see:

```
python app.py
```

think:

```
Terminal
   ↓
Python executable starts
   ↓
app.py is loaded
   ↓
Source is parsed
   ↓
Source is compiled to bytecode
   ↓
Bytecode is executed
   ↓
Program interacts with OS/resources
   ↓
Output or behavior
```

This model will help you understand later topics such as:

- imports
- modules
- exceptions
- debugging
- memory

- packages
- performance
- virtual environments
- application architecture

---

61. What Comes Next

You now understand:

```
What Python is
        ↓
How Python is installed
        ↓
How a Python file is created
        ↓
How Python code is executed
```

The next step is to actually write your first proper Python program and understand its structure.

You will move from:

```
"How does Python execute code?"
```

to:

```
"How do I structure a Python program?"
```

That means learning:

- `print()`
- statements
- expressions
- literals
- basic program structure
- execution flow
- writing readable Python
- turning simple instructions into a working program

The goal is no longer just to run Python.

The goal is to start **programming with Python**.

Visit haas.dev for more resources and guides.

haas.dev

https://dev-roast-app.vercel.app