Reading Python Errors With Confidence

AiCE Software Engineering Foundations — Python Programming

Writing code is only half the job. Understanding what Python does with that code — and what to do when it refuses to run — is the other half. This lesson builds the debugging instinct every developer needs, starting from working with multiple terminal tabs through to reading and fixing Python's most common errors.

1. Working With Multiple Terminal Tabs

As programs grow, you'll frequently want to edit code and see its output side by side without constantly switching contexts. A productive habit is to open two terminal tabs simultaneously — open a new tab with Ctrl + Shift + T.

Terminal Tab	Purpose	Typical Command
Tab 1	Run Python files and see output	python3 filename.py
Tab 2	Edit files in Vim	vim filename.py

This setup means you never have to close Vim to run your program — edit on one side, switch tabs, run on the other, see the result, switch back, and continue editing. Professional developers work this way constantly.

2. Understanding Python Errors

Every programmer encounters errors — this is not a sign of weakness. The difference between a beginner and an experienced developer isn't that one makes errors and the other doesn't; it's that the experienced developer reads errors quickly and fixes them efficiently.

Python's error messages tell you three things: what type of error occurred, which file it's in, and which line it's on.

The Anatomy of a Python Error Message

```
File "hello.py", line 1
  print("Hello, Africa"
        ^
SyntaxError: '(' was never closed
```

Reading from top to bottom:

1. The file and line number where the problem was detected.

2. The line of code itself — exactly what Python was reading when it hit the problem.

3. A caret (^) pointing to the specific part of the line where Python got confused.

4. The error type and a plain-English description of what went wrong.

With this information, you can go directly to the problem and fix it — no need to search the entire file blindly.

3. Common Error Types in Python

SyntaxError — Breaking Python's Grammar Rules

A SyntaxError means your code violates a rule of the Python language, so Python cannot even begin to execute the file. It's the equivalent of a broken grammar rule in a spoken sentence — the listener knows immediately that something is off.

Removing a closing bracket, for example:

```
print("Hello, Africa!"
```

produces:

```
File "hello.py", line 1
  print("Hello, Africa!"
        ^
SyntaxError: '(' was never closed
```

The Reported Line Isn't Always the Broken One

The line number in a SyntaxError is where Python detected the problem — but sometimes the actual mistake is on the line before. If fixing the reported line doesn't work, check the line above it.

NameError — Using Something Python Does Not Recognize

A NameError occurs when you use a name — a variable, function, or command — that Python doesn't recognize, usually from a typo or capitalization mistake. Changing print to Print, for example:

```
Print("Hello, Africa!")
```

produces:

```
File "hello.py", line 1, in <module>
  Print("Hello, Africa!")
  ^^^^^
NameError: name 'Print' is not defined. Did you mean: 'print'?
```

Python Is Case-Sensitive

print, Print, and PRINT are three completely different names as far as Python is concerned. This applies to everything — variables, functions, built-in commands, and keywords. A variable named city is different from City and different from CITY.

Written As	Python Sees It As	Result
print	Built-in Python function	Works correctly
Print	Unknown name	NameError
PRINT	Unknown name	NameError

TypeError — Mixing Incompatible Data

A TypeError occurs when you try to perform an operation on data that doesn't support it — for example, adding a number to a piece of text, or calling a function with the wrong kind of argument. You'll encounter these naturally in the chapters ahead, especially around user input and data types. When you see one, ask: does this operation make sense for the type of data involved?

4. Creating and Running Multiple Python Scripts

A real Python project is almost never a single file. Practice creating a few separate scripts in the same session. A greetings.py file printing several lines runs exactly in the order they were written — Python always executes top to bottom.

A numbers.py file reveals an important distinction:

```
print(100)
print(3.14)
print(2026)
```

> **Two Types of Output, One print() Function**
> *Text must always be wrapped in quotation marks: print("Hello"). Numbers are written without them: print(100). Writing a number inside quotes — print("100") — gives you the text string "100", which looks identical when printed but behaves completely differently in calculations, as the next document explores.*

5. Reading Errors With Confidence — A Full Practice Round

Suppose a file has two deliberate mistakes: a missing closing quotation mark on line 1, and print written as PRINT on line 3. Running it produces:

```
File "facts.py", line 1
  print("Python was created by Guido van Rossum in 1991.)
        ^
SyntaxError: unterminated string literal (detected at line 1)
```

Python stops at line 1 and never even reaches line 3 — it won't report the second error until the first is fixed. This is why you always fix errors from the top down, one at a time. After fixing line 1 and running again, the NameError on line 3 appears next:

```
File "facts.py", line 3, in <module>
  PRINT("Python powers AI systems...")
  ^^^^^
NameError: name 'PRINT' is not defined. Did you mean: 'print'?
```

Fix line 3, save, and run a final time — the program executes perfectly. This is the complete debugging cycle: introduce error → run → read error → identify problem → fix → run again. Professional developers repeat this cycle all day, every day.

Key Takeaways

Errors are not a sign you're doing something wrong — they're Python giving you precise, structured information about exactly what to fix and where. Read from the top down, fix one error at a time, and remember the three big ones: SyntaxError (broken grammar), NameError (an unrecognized name, often from case sensitivity), and TypeError (incompatible data types mixed together). With the environment set up and the confidence to read errors calmly, the final lesson in this unit turns to the habit that ties every session together: pushing your work to GitHub.