PYTHON ERRORS MADE EASY

A Plain-English Guide to Understanding and Fixing Common Python Mistakes

RUTH BARNHART

Python Errors Explained in Plain English

First edition, 2026.

Disclaimer

This book is provided for general educational purposes. The code examples are written to illustrate common Python errors and are offered as is, without warranty of any kind. While great care has been taken to ensure accuracy, the author accepts no responsibility for any loss or damage arising from the use of the information or code in this book.

Python is an open source language maintained by the Python Software Foundation. Any other product, company, or tool names mentioned are the property of their respective owners and are used for identification and educational purposes only.

Error messages can vary slightly depending on your version of Python and your operating system. The wording shown in this book reflects common, recent versions, so do not worry if your screen phrases something a little differently. The meaning, and the way you fix it, stays the same.

Contents

Foreword

If you have ever stared at a screen full of red text and felt your confidence drain away, this little book is for you. I wrote it because the moment a program breaks is the exact moment most beginners quietly decide that coding is not for them. That always breaks my heart, because an error is not a wall. It is a signpost, and once you can read it, it points you straight to the fix.

When I was learning to code, nobody told me that errors were normal. I thought every red message was proof that I had failed. It took me far too long to realize that the people I admired, the ones who seemed to write flawless programs, were quietly fixing dozens of errors a day. They had not avoided errors. They had simply stopped being afraid of them, and that one change made all the difference.

That is the whole purpose of this guide. It is a companion to your main Python journey, focused on a single, powerful skill: reading an error message calmly and knowing exactly what to do next. Each chapter takes one common error, explains in plain English what Python is trying to tell you, walks through the usual causes, and finishes with a short checklist and a little practice so the lesson sticks.

You do not need to read it cover to cover in one sitting, though you certainly can. Keep it beside you while you code. When something breaks, find the matching chapter, follow the steps, and fix it. Over time you will reach for the book less and less, because the calm, methodical habits in these pages will quietly become your own. The final chapter even gives you a single system that ties every error together, so you always know where to start.

A small promise before you begin. By the last page, the red text that once made your stomach drop will feel ordinary, almost friendly. You will read it, nod, and reach for the fix. That quiet confidence is worth more than any single trick, and it is well within your reach. Let us turn the page and begin.

Ruth Barnhart

Why Python Errors Are Not the Enemy

The first time you see a Python error, your stomach probably drops. The screen fills with red text, words you do not recognize show up, and a quiet voice in your head says, “See? You are not cut out for this.” Let me stop that voice right now. An error is not proof that you are bad at coding. It is one of the most normal, expected, and genuinely helpful things that happens to every programmer alive, including the ones who write code for a living every single day.

Errors are simply Python talking to you. When something is unclear, impossible, or written in a way Python cannot follow, it pauses and tells you about it. That is not punishment. That is feedback. By the end of this chapter, you will start to read those red messages as small notes instead of scary verdicts, and that shift in how you see them changes everything that comes next.

What an Error Message Really Means

Python is not angry with you. It cannot be. It is a tool, and like any tool, it only stops when it cannot safely keep going. Think of a map app that says, “Cannot find route.” It is not insulting your sense of direction. It just needs better information before it can help you.

An error message works the same way. It is a clue, not a judgment. It points at the spot where Python got confused, and it gives you a short description of what went wrong. Once you learn to read that clue calmly, you stop guessing wildly and start fixing on purpose. That is the entire difference between a frustrated beginner and a calm one.

The Three Parts of Most Python Errors

Almost every Python error hands you three useful pieces of information. Learn to spot them and most of the fear quietly disappears.

First, it tells you where the problem happened, usually with a line number. Second, it tells you what type of error it is, such as SyntaxError or NameError. Third, it gives you a short hint about the likely cause.

Here is a tiny broken example:

```
print("Hello"
```

Run that and Python will complain. It notices that you opened a parenthesis but never closed it, and it points you toward that line. The wording may look unfriendly, but it is really just saying, “I was waiting for a closing parenthesis and never got one.”

Why Beginners Misread Error Messages

Most beginners do not actually have an error problem. They have a reading problem. The red text appears, panic kicks in, and the useful details get skipped right past.

Here are the most common reactions, and you may recognize one or two of your own:

- Deleting everything in a rush, instead of fixing one small thing.
- Assuming the entire program is broken, when usually only one line is.
- Reading only the last line of your code and ignoring the actual message.
- Not taking Python literally, even though it almost always means exactly what it says.

The fix is wonderfully simple. Slow down for ten seconds and actually read what Python wrote. The answer is often sitting right there in plain sight, waiting for you to notice it.

The Beginner Debugging Mindset

When an error shows up, you do not need to be clever. You need to be calm and systematic. Use this short checklist every single time, and let it carry you when your nerves want to take over.

1. Read the last line first. It usually names the error and the cause.
2. Find the error type, such as SyntaxError or NameError.
3. Find the line number that Python is pointing to.
4. Look one or two lines above the error too, because the real cause sometimes sits just before it.
5. Change one thing at a time.
6. Run the program again and see what changed.

Notice those last two steps. Changing one thing at a time is the single most powerful habit you can build as a new coder. If you change five things at once and the error happens to disappear, you will never know which fix worked, and you will learn nothing for next time.

Mini Practice

Time to practice reading instead of panicking. Below are three short error situations. For each one, do not fix anything yet. Just answer three questions: What is the error type? Which line number is involved? What is the probable cause?

Situation one:

```
File "main.py", line 2
    print("Welcome to Python"
          ^
SyntaxError: '(' was never closed
```

Situation two:

```
File "main.py", line 1, in <module>
    print(total)
NameError: name 'total' is not defined
```

Situation three:

```
File "main.py", line 4, in <module>
    print("You are " + age)
TypeError: can only concatenate str (not "int") to str
```

Work through them slowly, one at a time, and write your answers down before you peek at anything. You are not solving these yet, and you do not need to. You are training your eyes to find the type, the

line, and the hint, in that order, every time. That one habit is worth more than any single fix you will ever make. It is the difference between feeling stuck and feeling in control. So treat this as a calm warm up rather than a test. That one skill will carry you through every chapter that follows. Errors are not the enemy. They are your earliest and most patient teacher.

SyntaxError, When Python Cannot Understand the Sentence

Of all the errors you will ever meet, SyntaxError is the most common, the most harmless, and the easiest to fix once you know what to look for. It does not mean your idea is wrong. It means the way you wrote that idea is not something Python can read. Think of it as a grammar mistake rather than a thinking mistake.

Python is a language with rules, just like English. You can have a brilliant sentence in your head, but if you forget the period, scramble the word order, or leave a quotation mark hanging open, a careful reader gets confused. Python is that careful reader, and it is a strict one. When it cannot turn your line into a valid instruction, it stops and raises a SyntaxError. The good news is that these errors follow patterns. Once you have seen the same five or six slips a few times, you will start spotting them before you even run your code. In this chapter you will learn the small handful of slips that cause almost every SyntaxError a beginner runs into.

What a SyntaxError Really Is

A SyntaxError happens before your program even starts running. Python reads through your code first to check that every line is written correctly. If it finds a line it cannot make sense of, it refuses to run anything at all and points you toward the spot where it got stuck. That is actually good news. Nothing half ran and left a mess. Python simply asked you to fix the sentence before it begins.

Here is the classic example:

```
print("Hello"
```

The idea is perfectly clear. You want to print the word Hello. But you opened a parenthesis and never closed it, so Python keeps waiting for the rest of the instruction. It reaches the end of the line still expecting

a closing parenthesis, gives up, and tells you something is wrong. The fix is tiny. Add the missing closing parenthesis and the line works.

Missing Parentheses

Parentheses come in pairs. Every time you open one with a round bracket, Python expects you to close it later with a matching one. Forgetting the closing bracket is probably the single most common SyntaxError for new coders, and it hides easily inside longer lines.

```
print("Welcome to Python"
```

Python sees the opening bracket after print and waits patiently for a partner that never arrives. The correct version simply closes the pair:

```
print("Welcome to Python")
```

A helpful habit is to type both brackets first, then move your cursor between them and fill in the contents. That way the pair is always complete before you start typing inside it, and you never have to hunt for a missing one later.

Missing or Unclosed Quotes

Text in Python lives inside quotation marks. You can use single quotes or double quotes, but whichever one you open, you must close with the same kind. Leave a quote hanging open and Python cannot tell where your text is supposed to end.

```
name = "Alex
```

Here the opening double quote has no closing partner, so Python keeps reading, looking for the end of the text, and eventually raises a SyntaxError about an unterminated string. The fix is to close the quote:

```
name = "Alex"
```

There is a close cousin of this mistake worth a quick mention. If you forget the quotes entirely and write name equals Alex with no marks at all, Python does not see a SyntaxError. Instead it assumes Alex is the name of another variable and complains that it does not exist.

That is a different error called a NameError, and it gets its own chapter later. For now, just remember that text needs matching quotes on both ends.

Missing Colons

Certain lines in Python act like a heading that introduces a block of code below them. Lines that start with if, else, elif, for, while, and def all need to end with a colon. The colon is Python's way of saying that the indented lines which follow belong to it.

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

That looks almost right, but the first line is missing its colon, so Python raises a SyntaxError. Add the colon and the meaning becomes clear:

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

Whenever you write one of those block opening words, get into the rhythm of finishing the line with a colon before you press Enter. It quickly becomes automatic, and it removes one of the most frequent beginner stumbles in a single habit.

Using the Wrong Symbol

Some SyntaxErrors come from using a character that looks correct but is not. The sneakiest culprit is the curly quote. Word processors and note apps love to turn straight quotes into fancy curved ones, and Python does not accept those.

```
name = ‘Alex’
```

Those slanted quotes will trigger a SyntaxError. Retype them as plain straight quotes and the problem disappears:

```
name = 'Alex'
```

Other symbol slips include mixing a single and a double quote, adding an extra bracket that has no partner, or forgetting a comma between items in a list. Watch this last one closely:

```
fruits = ["apple", "banana" "orange"]
```

Interestingly, that line does not always raise an error. Python quietly glues banana and orange into one item because they sit side by side with no comma between them. You end up with a list of two items instead of three, and no warning at all. That is why a missing comma can be more dangerous than a loud SyntaxError. It hides in plain sight and changes your results without saying a word.

Your SyntaxError Checklist

When a SyntaxError appears, run down this short list and you will catch the cause almost every time:

- Are all your parentheses closed, with one closing bracket for every opening one?
- Are all your quotes closed, using the same style on both ends?
- Does every if, elif, else, for, while, and def line end with a colon?
- Are you using plain straight quotes rather than curly ones?
- Is there a stray extra symbol, or a missing comma between list items?

Mini Practice

Each snippet below contains exactly one SyntaxError. Read it, name what is missing or wrong, and write the corrected version. Work through them one at a time.

Snippet one:

```
print("Good morning"
```

Snippet two:

```
if age > 18
    print("You can vote")
```

Snippet three:

```
city = ‘London’
```

Snippet four:

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

Snippet five:

```
greeting = "Hello there
```

Once you have your fixes, try reading each line out loud as if it were a sentence. A SyntaxError almost always turns out to be something left open, something left out, or something typed in the wrong shape. None of these are signs that you are bad at coding. They are simply the small mechanical habits that every programmer builds in their first week. Fix that one small thing and Python will understand you perfectly.

IndentationError, When Python Cannot See the Code Blocks

Most programming languages use curly braces or special keywords to mark where a block of code begins and ends. Python does something gentler and, for beginners, much friendlier. It uses the blank space at the start of a line. That space is called indentation, and in Python it is not decoration. It carries meaning. When the indentation does not line up the way Python expects, it raises an IndentationError.

This error frightens beginners more than it should, because the cause is invisible. You are staring at spaces, and spaces are hard to see. The trick is to stop looking at the words for a moment and look only at how far each line sits from the left margin. Once you learn to read indentation as structure, this error becomes one of the simplest to fix.

What Indentation Actually Means

Indentation means the spaces at the very beginning of a line. In everyday writing we indent to make text look tidy. In Python, indentation tells the language which lines belong together as a group. A line that is pushed in below another line is saying, I am part of what came just above me.

```
if True:
print("This will fail")
```

Python reads the colon at the end of the first line and expects the next line to be pushed in, showing that it belongs to the if statement. Instead the print line sits flat against the margin, so Python is confused and stops. The correct version indents the line that belongs inside the block:

```
if True:
    print("This works")
```

Why Python Cares About Spaces

In some languages you could write your whole program on one line and the computer would still understand it, because braces mark the groups. Python made a deliberate choice to use indentation instead. The result is code that looks like a clean outline, where structure is visible at a glance.

The payoff is real. Well indented Python is genuinely easier to read, both for you and for anyone who looks at your code later, including the version of you who returns to it next month. The small cost is that the spacing has to be consistent, because Python is using it to understand your intentions. When you treat indentation as part of the meaning rather than as styling, the rules suddenly feel logical instead of fussy, and you stop fighting them.

Expected an Indented Block

This is the message Python gives when a line ends with a colon but nothing underneath it is pushed in. The colon promised a block, and the block never arrived.

```
if age > 18:
print("You can vote")
```

Python sees the colon, looks at the next line, and expects it to step inward. Because the print line is flat, Python reports that it expected an indented block. The fix is to push that line in, usually by four spaces:

```
if age > 18:
    print("You can vote")
```

Unexpected Indent

The opposite problem also exists. Sometimes a line is pushed in when there is no reason for it to be. Python then asks, why does this line suddenly step inward when nothing above it opened a block?

```
name = "Alex"
    print(name)
```

The first line is a plain instruction. There is no colon and no block, so the second line has no reason to be indented. Python raises an IndentationError about an unexpected indent. The fix is to line both statements up against the same margin:

```
name = "Alex"
print(name)
```

Mixing Tabs and Spaces

There is one more cause that drives beginners quietly mad, because the code can look perfectly aligned on screen while Python still complains. It happens when some lines are indented with the Tab key and others with the space bar. To your eyes they match. To Python they are different characters, and that inconsistency triggers an error.

The cure is simple and worth adopting from day one. Pick spaces and never look back. Four spaces for each level of indentation is the standard that almost every Python coder follows. Better still, let your editor do the work. In VS Code you can set it to insert spaces automatically when you press Tab, so you get the convenience of the Tab key with the safety of spaces.

Indentation Inside Loops and Functions

Indentation also decides which lines run repeatedly inside a loop and which run only once afterward. Read this carefully:

```
for item in shopping_list:
    print(item)
print("Done")
```

The print(item) line is indented, so it belongs to the loop and runs once for every item on the list. The print Done line is flat, so it sits outside the loop and runs a single time after the loop finishes. That one difference in spacing completely changes how the program behaves, which is exactly why Python treats indentation as meaning rather than style.

Your Indentation Checklist

When an IndentationError appears, walk through these questions in order:

- Did the line just above end with a colon, promising a block below it?
- Should this line sit inside a block, or out on its own at the margin?
- Do all the lines in the same block step in by the same amount?
- Are you using spaces everywhere, rather than mixing in tabs?
- Did one line drift further right than it should have?

Mini Practice

Each snippet below has an indentation problem. Rewrite it with correct spacing, and as you do, say out loud which lines belong inside a block and which stand on their own.

Snippet one:

```
if score > 50:
print("You passed")
```

Snippet two:

```
total = 10
    print(total)
```

Snippet three:

```
for number in numbers:
print(number)
```

Snippet four:

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

Remember the simple idea behind all four. A colon opens a block, the lines inside that block step in together by the same amount, and the lines outside the block return to the margin. When you are stuck, place your cursor at the start of each line and check how far it sits

from the left edge, ignoring the words for a moment. Hold that picture in your head and IndentationErrors lose almost all of their mystery.

NameError, When Python Does Not Recognize a Name

A NameError means that you used a name, such as a variable or a function, that Python has never been told about. It is a little like calling out to someone across a room and using a name that nobody there answers to. Python looks around, finds nothing that matches, and stops to ask you who you mean.

This error feels mysterious at first because the word you typed looks perfectly fine. The cause is almost always one of a few small things: a name used before it was created, a spelling slip, a difference in capital letters, or a word that should have been wrapped in quotes. None of these are signs of a deep misunderstanding. They are tiny mismatches between what you meant and what you typed. Learn to check those four suspects and most NameErrors solve themselves in seconds.

What a NameError Really Is

Python keeps track of every name you create, whether it is a variable holding a value or a function you can call. When you ask it to use a name, it searches that memory. If the name is not there, it raises a NameError and tells you which name it could not find.

```
print(username)
```

If you never created a variable called username, Python has nothing to print. It reports that the name username is not defined. The word defined simply means created. Python is saying, I have never been given anything by that name, so I do not know what you want me to show.

Using a Variable Before You Create It

Python reads your code from top to bottom, one line at a time, exactly in the order you wrote it. That means a variable has to be created before the line that uses it. If you try to use it first, it does not exist yet.

```
print(name)
name = "Mia"
```

At the moment Python reaches the print line, the name variable has not been created, so it raises a NameError. Swapping the two lines fixes everything, because now the variable exists before it is used:

```
name = "Mia"
print(name)
```

Whenever you see a NameError, one of the first things to check is whether the line that creates the name sits above the line that uses it.

Spelling and Capital Letters

Python is fussy about names in two ways that catch beginners constantly. First, the spelling must match exactly. Second, capital letters count. To Python, the word userName and the word username are two completely different names.

```
userName = "Sam"
print(username)
```

You created a variable called userName with a capital N, then asked to print username all in lowercase. Python sees no variable with that exact spelling, so it raises a NameError. The lesson is to copy your names carefully and keep them consistent. Many coders avoid this trap by always writing variable names in lowercase with underscores, such as user_name, so there is never a guessing game about capital letters.

Forgetting Quotes Around Text

This is a sneaky one, because it looks like a value but behaves like a name. When you want to store a piece of text, it must sit inside quotation marks. Leave the quotes off and Python assumes you are referring to another variable.

```
city = London
```

Without quotes, Python thinks London is the name of a variable somewhere, goes looking for it, and cannot find it. The result is a

NameError about London not being defined. Add the quotes and Python understands that you mean the word London as text:

```
city = "London"
```

A good way to remember this is that quotes are the difference between a label and the thing on the label. With quotes, you are handing Python the actual words. Without quotes, you are pointing at a box and asking what is inside it.

Function Names That Do Not Exist

Functions have names too, and the same exact spelling rule applies to them. A single wrong letter is enough to make Python raise a NameError, because the misspelled word does not match any function it knows.

```
pritn("Hello")
```

The letters in print got jumbled into pritn. Python has no function by that name, so it stops. This is one of the most common typos in all of beginner coding, and the fix is simply to correct the spelling to print. When a function call raises a NameError, suspect a typo before anything else.

Names That Live Inside Functions

There is one more cause that feels surprising until you understand it. A variable created inside a function lives only inside that function. The world outside cannot see it. This idea is called scope, and you only need the gentle version of it for now.

```
def greet():
    message = "Hello"
print(message)
```

The message variable is born inside the greet function and exists only while that function runs. The final print line sits outside the function, where message was never created, so Python raises a NameError. Think of the function as a small private room. What you make in that room stays in that room unless you deliberately hand it out through a return value, which is something you will meet in the main book. For

now the takeaway is simple. If a name was made inside a function, you cannot reach it from outside that function.

Your NameError Checklist

When a NameError appears, ask yourself these questions one by one:

- Did I actually create this variable somewhere before using it?
- Does the line that creates it sit above the line that uses it?
- Is the spelling exactly the same in both places?
- Do the capital letters match exactly?
- Should this really be text wrapped in quotes?
- Was the name created inside a function I am now outside of?

Mini Practice

Each snippet below raises a NameError. Find the cause, then write the corrected version. Try to name which of the suspects above is to blame each time.

Snippet one:

```
print(total)
```

Snippet two:

```
print(age)
age = 25
```

Snippet three:

```
firstName = "Sam"
print(firstname)
```

Snippet four:

```
country = Spain
```

Snippet five:

```
prnt("Welcome")
```

As you fix each one, notice how rarely the problem is anything deep or complicated. A NameError is almost always a small mismatch

between what you typed and what you created. Tighten up your spelling, mind your capital letters, create things before you use them, and this error will quietly fade into the background.

TypeError, When Python Gets the Wrong Kind of Thing

Every value in Python has a type. Text is a string, whole numbers are integers, decimal numbers are floats, and true or false values are booleans. A TypeError appears when you ask Python to do something that makes sense in general, but you hand it the wrong kind of value to do it with. The action is fine. The ingredient does not fit.

Picture trying to pour a glass of orange juice into a calculator. Pouring is a normal action and a glass is a normal object, but the two were never meant to meet. A TypeError is Python noticing that kind of mismatch and stopping before it makes a mess. The reassuring part is that Python tells you the types involved right inside the message, so you are never left guessing. Most TypeErrors for beginners come from mixing text and numbers, so that is where we will spend most of our time, but the same way of thinking applies to every type you will ever use.

What a TypeError Really Is

A TypeError tells you that the operation you tried does not work on the type of value you gave it. The clearest example is trying to glue text and a number together with a plus sign.

```
age = 30
print("You are " + age)
```

You wanted to join the words You are with the number stored in age. The trouble is that the plus sign means two different things depending on the type. For two strings it joins them. For two numbers it adds them. Python cannot quietly decide to mix a string and a number, so it raises a TypeError rather than guessing what you meant.

Strings and Numbers Do Not Mix Automatically

To join text and a number, you have to make them the same type first. The most common fix is to turn the number into text with str, so that both sides of the plus sign are strings.

```
print("You are " + str(age))
```

There is an even simpler approach. If you give print several items separated by commas, it prints them all with a space in between and handles the types for you.

```
print("You are", age)
```

This is also where the conversion tools earn their keep. Use str to turn a value into text, int to turn text into a whole number, and float to turn text into a decimal number. Choosing the right one for the moment solves a large share of all TypeErrors. It helps to think of these three as small translators standing between the world of text and the world of numbers, ready to carry a value across whenever you need it on the other side.

Input Always Arrives as Text

Here is a concept that trips up nearly every beginner. Whatever a user types in response to input is handed to your program as text, even when they type digits. Watch what happens when you forget that.

```
num1 = input("First number: ")
num2 = input("Second number: ")
print(num1 + num2)
```

If the user types 5 and then 7, you might expect 12. Instead you get 57. No error appears at all, because both values are text and the plus sign simply joins them. This silent wrong answer is sneakier than a crash, and the root cause is the same type confusion behind many TypeErrors. The fix is to convert each input into a number before doing math:

```
num1 = int(input("First number: "))
num2 = int(input("Second number: "))
print(num1 + num2)
```

Now both values are whole numbers, the plus sign adds them properly, and you get 12. Whenever a calculation gives a strange result or a TypeError, check whether a value that should be a number is secretly still text from an input.

Calling Something That Is Not a Function

Round brackets after a name mean run this like a function. If you put brackets after something that is not a function, Python does not know how to run it and raises a TypeError.

```
name = "Alex"
name()
```

Here name holds a piece of text, not a function. The brackets ask Python to call it, but text cannot be called, so it reports that a string object is not callable. The fix is to drop the brackets when you only want to use the value, and to keep them only when you are actually calling a real function. A quick mental check helps here. Brackets mean do something. No brackets mean use the thing as it is. If you are not asking Python to perform an action, the brackets do not belong.

Using a Method on the Wrong Type

Different types come with their own built in abilities, called methods. Text can be made uppercase, for example, but numbers cannot, because the idea does not apply to them. When you try a method on a type that does not have it, Python stops.

```
age = 25
age.upper()
```

The method called upper turns text into capital letters, so it works on strings but not on numbers. Asking a number to make itself uppercase makes no sense, so Python raises an AttributeError, which is a close relative of the TypeError. It belongs to the same family of mismatches, where the value you have simply does not support the thing you asked it to do. The practical rule is easy to remember. Text methods like upper and lower work on strings, so make sure the value really is a string before you reach for them.

Too Few or Too Many Arguments

When you write a function that expects some information, you have to give it exactly what it asks for. Leaving out a required value, or supplying extra ones, leads to a TypeError.

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

The greet function expects one piece of information, the name to greet. Calling greet with empty brackets gives it nothing, so Python raises a TypeError saying a required argument is missing. The same kind of error appears in reverse if you hand a function more values than it asked for. Provide exactly the value the function is waiting for and it runs happily:

```
greet("Sam")
```

Your TypeError Checklist

When a TypeError appears, work through these questions:

- What type is each value involved, text or a number?
- Am I trying to join text and a number with a plus sign?
- Did this value come from input, which means it is still text?
- Do I need str, int, or float to convert a value first?
- Did I put brackets after something that is not a function?
- Did I give a function the right number of values?

Mini Practice

Each snippet below raises a TypeError. Identify the type mismatch and write a corrected version that does what the code clearly intended.

Snippet one:

```
print("Total: " + 5)
```

Snippet two:

```
age = input("Your age: ")
print(age + 10)
```

Snippet three:

```
result = 7
result()
```

Snippet four:

```
def add(a, b):
    return a + b
add(3)
```

Snippet five:

```
print(len(5))
```

As you fix these, notice the pattern that ties them together. A TypeError is rarely about a broken idea. It is about a value showing up in the wrong outfit. Once you get into the habit of asking what type is this really, the cure usually presents itself, whether that is a quick conversion, a removed pair of brackets, or a missing value handed to a function.

ValueError, When the Type Is Right but the Value Is Wrong

A ValueError is the close cousin of the TypeError from the last chapter, and beginners often confuse the two. The difference is worth holding onto. A TypeError means you used the wrong kind of value. A ValueError means you used the right kind of value, but its actual contents cannot be handled by the action you asked for. The shape is correct. The specific value is not.

A simple comparison makes it stick. Imagine a form that asks for your age and you write the word banana in the box. The box was the right place to write something, and writing is the right action. The trouble is that banana is not a sensible age. Python runs into the same situation when you ask it to turn a value into a number and that value cannot become one. The container and the operation are both fine, and only the contents are the problem. Keeping that picture in mind will help you tell a ValueError apart from its noisier cousin, the TypeError.

What a ValueError Really Is

The clearest example is asking Python to convert a piece of text into a number when the text is not a number at all.

```
age = int("hello")
```

The int tool exists, and turning text into a number is a perfectly normal thing to do, so this is not a TypeError. The problem is the value itself. The word hello has no numeric meaning, so Python cannot complete the conversion and raises a ValueError. Notice the distinction. The action was allowed. The value simply did not fit the action.

The Most Common Beginner ValueError

By far the most frequent way to meet this error is when you ask a user for a number and they type something that is not one. Remember

from the last chapter that input always arrives as text, so converting it is normal. The danger is what happens when the text is not a clean number.

```
number = int(input("Enter a number: "))
```

If the user politely types 42, everything works. But if they type the word ten, or leave the box empty, or add a stray letter, the int conversion fails and the program crashes with a ValueError. This is one of the first places where you learn that real users do not always do what you expect, and your code has to be ready for that. It is nothing personal. People mistype, get distracted, or simply read the prompt differently than you intended. A program that survives those moments feels far more professional than one that falls over at the first surprise.

Why 10 Works but Ten Does Not

The rule Python follows here is stricter than it first appears. The text has to look exactly like a number for the conversion to succeed.

```
int("10")
```

That works, because the characters one and zero spell a valid whole number, even though they are written as text. This, however, does not work:

```
int("ten")
```

The word ten means a number to a human reader, but to Python it is just three letters with no numeric value. Python does not translate spelled out words. It only accepts text that is written in digits.

The Same Problem with Decimals

Floats follow exactly the same logic. The float tool can turn text into a decimal number, but only when the text actually looks like one.

```
price = float("12.99")
```

That succeeds without complaint. The next line, however, fails:

```
price = float("twelve")
```

Once again the word is meaningful to you but meaningless to Python as a number. Whether you are using int or float, the lesson is the same. The text must be written in digits, not in words, for the conversion to work.

Handling Bad Input Gently

You do not have to let a single bad entry crash your whole program. Python gives you a way to try something and then catch the problem if it goes wrong, using try and except. You will explore this idea more fully elsewhere, so here is just enough to keep your programs calm.

```
try:
    age = int(input("Enter your age: "))
    print("You are", age)
except ValueError:
    print("Please enter a valid number.")
```

Python first attempts the lines under try. If the conversion succeeds, it carries on as normal. If it raises a ValueError, Python skips down to the except block and runs your friendly message instead of crashing. Notice that the except line names ValueError on purpose. Catching the specific error you expect is a good habit, because it leaves real surprises visible instead of hiding every possible problem at once. A common beginner temptation is to catch everything with a bare except, but that can quietly swallow bugs you would actually want to know about. Naming the error keeps you honest and keeps your program readable.

ValueError in Lists

This error is not only about converting numbers. It also shows up when you ask a list to remove an item that is not there.

```
numbers = [1, 2, 3]
numbers.remove(5)
```

Here the list exists, and remove is a real and valid action for a list, so this is not a TypeError. The problem is the value five, which is simply not in the list. Python cannot remove something that was never there,

so it raises a ValueError. The pattern is the same one you have seen all chapter. The action was fine. The value did not fit the situation.

Your ValueError Checklist

When a ValueError appears, ask yourself these questions:

- Is the value actually suitable for the conversion I asked for?
- Am I trying to turn text into a number when the text is not written in digits?
- Is the value coming from a user, who might type anything at all?
- Should I wrap this in a try and except block to handle bad input?
- For a list, does the item really exist before I try to remove it?

Mini Practice

The first four snippets are about converting values. The last two are about lists. Identify why each one raises a ValueError, then describe how you would handle or correct it.

Snippet one:

```
int("seven")
```

Snippet two:

```
price = float("ten dollars")
```

Snippet three:

```
year = int("2023x")
```

Snippet four:

```
quantity = int(input("How many? "))
```

For snippet four, imagine the user typed the word two rather than the digit 2.

Snippet five:

```
colors = ["red", "green", "blue"]
colors.remove("purple")
```

Snippet six:

```
scores = [10, 20, 30]
scores.remove(40)
```

By now the theme should feel familiar. A ValueError is never about doing the wrong kind of operation. It is about feeding a sensible operation a value it cannot use. Check your values, expect surprises from anything a user types, and lean on try and except when you want your programs to stay polite under pressure.

FileNotFoundError, When Python Cannot Find Your File

As soon as your programs start reading and writing files, a new error joins the party. The FileNotFoundError appears when you ask Python to open a file that it cannot locate. The good news is that this error is almost always about where a file is, or what it is called, rather than anything wrong with your logic. Once you understand how Python looks for files, it becomes one of the most predictable errors of all.

Beginners often assume Python can see the file because they can see it sitting on their desktop. Python is not looking at your screen, though. It follows a precise set of rules about names and locations, and when reality does not match those rules, it stops and tells you the file was not found.

What a FileNotFoundError Really Is

When you open a file for reading, you are asking Python to go find an existing file and hand you its contents.

```
file = open("notes.txt", "r")
```

The letter r here means read. If a file called notes.txt exists in the place Python expects, this works perfectly. If it does not, Python cannot read something that is not there, so it raises a FileNotFoundError. The message usually includes the exact name it tried to open, which is your single most useful clue for fixing the problem.

The File May Not Exist Yet

Reading and writing are not the same. Opening a file in read mode only works if the file already exists. Opening a file in write mode is different, because write mode will create a brand new file if one is not there.

```
file = open("notes.txt", "w")
file.write("My first note")
file.close()
```

The letter w means write. This code happily creates notes.txt if it does not exist, puts a line of text inside it, and then closes it. So if you keep getting a FileNotFoundError while reading, one honest question to ask is whether the file has ever actually been created in the first place.

Python Looks in the Current Folder

This is the single most common cause of the error, and the one that surprises people most. When you give Python a plain file name, it looks for that file in the folder it is currently working from, which is usually the folder your script is running in. It does not search your whole computer.

```
open("data.txt", "r")
```

For this to work, data.txt needs to sit in the same folder as the script that is trying to open it. If the file is in your Downloads folder while your script lives in Documents, Python will not find it, even though both are clearly visible to you. The quick question to ask yourself is always the same. Is this file really in the same folder as my script?

Wrong Name or Wrong Extension

Python matches file names exactly, just as it matches variable names exactly. A tiny difference is enough to cause a miss.

The classic slips include a small spelling difference, such as asking for note.txt when the file is really called notes.txt, or getting the extension wrong, such as asking for data.txt when the file is actually data.csv. On Windows there is an extra trap, because the system often hides file extensions by default. A file that looks like notes on screen might really be notes.txt, and getting that ending right matters. Turning on the option to show file extensions removes a lot of confusion.

Wrong Folder Path

Sometimes you deliberately keep files in a subfolder, and you point Python at it with a path.

```
open("files/notes.txt", "r")
```

This tells Python to look inside a folder called files for notes.txt. It only works if that folder truly exists and contains that file. If you spell the folder name wrong, or the folder is not where you think, you get the same FileNotFoundError. When a path is involved, check every part of it, the folder name first and then the file name, rather than only looking at the file.

Opening Files Safely

Just as with bad user input, you do not have to let a missing file crash everything. You can try to open it and catch the problem if it is not there.

```
try:
    file = open("notes.txt", "r")
    print(file.read())
    file.close()
except FileNotFoundError:
    print("The file was not found. Check the filename and
folder.")
```

Python attempts the lines under try. If the file opens, it reads and prints the contents. If the file is missing, it jumps to the except block and shows a calm, helpful message instead of a frightening crash. As before, naming the specific FileNotFoundError is the tidy habit, because it handles the problem you expected without hiding anything else.

The Cleaner Way with the With Statement

There is a tidier style for working with files that experienced coders reach for almost automatically. It uses the word with, and its great advantage is that it closes the file for you when you are done, even if something goes wrong.

```
with open("notes.txt", "r") as file:
    content = file.read()
    print(content)
```

Everything indented under the with line has access to the file. The moment that block finishes, Python closes the file automatically, so you never have to remember to call close yourself. Building this habit early saves you from a whole category of small mistakes later on.

Your FileNotFoundError Checklist

When this error appears, walk through these questions:

- Does the file actually exist yet, or do I still need to create it?
- Is it in the same folder as the script that is opening it?
- Is the file name spelled exactly right, including capital letters?
- Is the extension correct, such as txt rather than csv?
- If I used a path, does every folder in that path really exist?
- Am I reading with r when I actually need to create the file first with w?

Mini Practice

This time you will make the error happen on purpose, which is a great way to lose your fear of it. Try these three small steps in order.

Step one, create a file by writing to it:

```
with open("practice.txt", "w") as file:
    file.write("Learning about files")
```

Step two, read the file back and print it:

```
with open("practice.txt", "r") as file:
    print(file.read())
```

Step three, on purpose ask for a name that does not exist and watch the error:

```
with open("pract1ce.txt", "r") as file:
    print(file.read())
```

That last line uses a misspelled name, so Python will raise a FileNotFoundError. Read the message, find the file name it reports, and compare it letter by letter with the file you actually created. Doing this once, deliberately, turns a scary error into an old friend. From now on, when you see a FileNotFoundError, you will know to check the name, the folder, and the spelling, and the fix will rarely be more than a small correction.

The Beginner's Error-Solving System

You now know the most common errors a beginner meets, one type at a time. This final chapter ties them together into a single calm routine you can use for any problem, no matter which error shows up. The goal is simple. When something breaks, you should never feel lost. You should reach for the same steps every time, the way a pilot reaches for a checklist.

Errors will keep happening for as long as you write code, and that is completely normal. The difference between a frustrated beginner and a confident one is not the number of errors they make. It is how calmly and quickly they work through them. This chapter is about building that calm into a habit.

The Six Step Error Fixing Method

Whenever an error appears, walk through these six steps in order. They work for every error in this book and almost every error you will meet later.

7. Stop and breathe. The red text is information, not a disaster.
8. Read the last line of the error first. It usually names the error and the cause.
9. Identify the error type, such as SyntaxError or ValueError.
10. Go to the line number that Python points to.
11. Check the surrounding lines, since the real cause sometimes sits just above.
12. Make one small fix and run the program again.

That last step matters more than it looks. Resist the urge to change several things at once. One change, then run, then look again. This rhythm keeps you in control and teaches you something with every loop. With practice, these six steps stop feeling like a list you have to

remember and start feeling like the natural thing you do the moment anything goes wrong.

How to Know Which Error You Have

Each error type is really a short message about what went wrong. Once you can translate the name into plain English, half the work is done. Keep this little table close until the meanings feel automatic.

Error	What It Usually Means
SyntaxError	Python cannot read the code sentence.
IndentationError	Python cannot understand the code block structure.
NameError	Python does not recognize the name.
TypeError	Python got the wrong kind of value.
ValueError	Python got the right kind of value, but a bad one.
FileNotFoundError	Python cannot find the file.

Notice how each meaning points you straight toward the fix. If Python cannot read the sentence, you check brackets, quotes, and colons. If it does not recognize a name, you check spelling and whether the name was created. The type tells you where to look.

Fix One Thing at a Time

It is tempting, especially when you are frustrated, to change five things at once and hope the error vanishes. The problem is that even if it does vanish, you will not know which change actually fixed it, and you may have created a brand new problem in the process. Change one thing, run the code, and read the result. If the error is different now, you have made progress, even when it does not feel like it. Slow and steady genuinely wins here.

Use Print to See What Python Sees

One of the most powerful debugging tools is also one of the simplest. When you are not sure what a value contains, print it out, along with its type.

```
age = input("Enter your age: ")
print("DEBUG:", age, type(age))
```

This shows you both the value and its type. Very often you will discover that something you assumed was a number is actually text, which instantly explains a TypeError or a strange result. The word DEBUG is just a label you add so you can spot these temporary lines easily and find them again when it is time to remove them. Sprinkling a few of these print lines through your code, then taking them out once it works, is a habit that will serve you for your entire coding life. It costs nothing, it requires no special tools, and it answers the most important debugging question of all, which is what is really happening at this exact point in my program.

Read Your Code from Top to Bottom

Python runs your code in order, one line after another, from the top of the file to the bottom. Many errors happen simply because something is used before it is ready, such as a variable printed before it is created. When you are stuck, trace through your code slowly in the same order Python does, pretending to be the computer and keeping track of what each variable holds at each step. This single technique uncovers a surprising number of bugs all on its own, and it costs you nothing but a few minutes of patience. If you find that tedious, that is a good sign, because it usually means the bug is hiding in a step you have been skipping over in your head.

Keep an Error Journal

Here is a habit that turns every error into a lasting lesson. Keep a simple note, on paper or in a file, and add to it whenever you solve something tricky. For each entry, jot down a few details.

- The date.

- The error type.
- What you were trying to do.
- What caused the error.
- How you fixed it.
- What you learned.

After a few weeks you will have your own personal troubleshooting guide, written in your own words, tuned to the exact mistakes you tend to make. Beginners who keep a journal like this improve noticeably faster, because they stop solving the same problem twice.

Final Practice, Diagnose the Bug

Here are six small broken snippets, one for each error type in this book. For each one, name the error type, point to the cause, and write a corrected version. Use the six step method and the table as you go.

Snippet one:

```
print("Hi"
```

Snippet two:

```
if True:
print("Yes")
```

Snippet three:

```
print(message)
```

Snippet four:

```
print("Age: " + 30)
```

Snippet five:

```
number = int("abc")
```

Snippet six:

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

A Closing Word of Encouragement

Errors do not disappear as you get better. The professionals you admire still see red text every single day. What changes is your relationship with it. You stop reading errors as proof that you cannot do this, and start reading them as ordinary feedback that points the way forward. Every error you solve makes you a little more independent, a little quicker, and a lot calmer. Keep your checklist handy, keep your journal growing, and trust that the panic you felt at the very first red message will soon be replaced by a quiet, confident, let us see what Python is telling me.