**FREE COMPANION DOWNLOAD** TYPE · PREDICT · RUN · MODIFY

# Common Python Errors

*What the messages mean and what to check first. Python often adds a suggestion of its own after the text shown here*

From **Python the TPRM Way** by John I. Jerkovic PRACTICAL TECH MADE SIMPLE

```
01 SyntaxError: invalid syntax
```

**WHAT IT MEANS** Python could not make sense of the structure of the line, so nothing ran at all. This is the most general syntax complaint, and the reported line is often one line after the real mistake.

**FIX** Read the line named in the message, then read the line above it. Look for a missing comma, a missing operator, an unclosed bracket, or a keyword that has been misspelled.

```
total = 5 6
```

```
02 SyntaxError: unterminated string literal
```

**WHAT IT MEANS** A string was opened with a quotation mark that was never closed, so Python read to the end of the line still looking for the partner.

**FIX** Add the closing quotation mark, and use the same character that opened the string.

```
print("Python is precise)
```

```
03 SyntaxError: '(' was never closed
```

**WHAT IT MEANS** An opening bracket has no matching closing bracket. Python kept reading past the end of the statement, waiting for it.

**FIX** Count the brackets on the reported line. Most editors highlight the partner when the cursor sits next to one.

```
print("hi"
print("there")
```

```
04 SyntaxError: expected ':'
```

**WHAT IT MEANS** A statement that opens a block was written without its colon. Every if, `elif`, else, for, while, `def`, class, try, and `except` line ends in one.

**FIX** Add the colon at the end of the line, then indent the block beneath it.

```
if 1 > 0
    print("positive")
```

```
05 SyntaxError: invalid syntax. Maybe you meant '==' or
    ↳ ':=' instead of '='?
```

**WHAT IT MEANS** A condition used a single equals sign. One equals sign assigns a value, and a condition has to compare instead.

**FIX** Use == in the condition. Keep the single = for assignment statements.

```
x = 5
if x = 5:
    print("five")
```

```
06 IndentationError: expected an indented block
```

**WHAT IT MEANS** A line ended with a colon, so Python expected an indented block underneath it, and the next line was not indented.

**FIX** Indent the body of the block, four spaces deeper than the line with the colon.

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

```
07 IndentationError: unexpected indent
```

**WHAT IT MEANS** A line is indented, and nothing above it opened a block. Indentation is how Python groups statements, so a stray space or two is a real error.

**FIX** Move the line back to the level of the statements around it.

```
x = 1
    y = 2
```

```
08 IndentationError: unindent does not match any outer
    ↳ indentation level
```

**WHAT IT MEANS** A line came back out from a block, but not to a level that any enclosing block uses.

**FIX** Line the statement up exactly with the block it belongs to. Mixed indent widths in one file are the usual cause.

```
if True:
    x = 1
  y = 2
```

```
09 TabError: inconsistent use of tabs and spaces in
    ↳ indentation
```

**WHAT IT MEANS** Some lines are indented with tab characters and others with spaces. They can look identical on screen and mean different things to Python.

**FIX** Pick spaces and use them everywhere. Most editors have a setting that inserts spaces when the Tab key is pressed.

```
if True:
    x = 1
        y = 2
```

```
10 NameError: name 'Print' is not defined
```

**WHAT IT MEANS** A name was used that Python has never been told about. Python is case-sensitive, so Print and `print` are two different names.

**FIX** Check the spelling and the capitalization against the name you meant.

```
Print("Hi")
```

```
11 NameError: name 'total' is not defined
```

**WHAT IT MEANS** The name exists somewhere, but not where it is being used. A variable created inside a function is local to that function and disappears when the call ends.

**FIX** Return the value from the function and store it in a variable at the call site.

```
def make_total():
    total = 10
    return total

print(make_total())
print(total)
```

```
12 TypeError: can only concatenate str (not "int") to
str
```

**WHAT IT MEANS** A string and a number were joined with +. Python will not quietly decide which of the two meanings of + you intended.

**FIX** Convert deliberately. Use `int(text)` to add numbers, or `str(number)` to build text, or put the value in an `f-string`.

```
age = "20"
print(age + 5)
```

```
13 TypeError: unsupported operand type(s) for -: 'str'
    ↳ and 'int'
```

**WHAT IT MEANS** Arithmetic was attempted on text. This one turns up constantly right after `input()`, because `input()` always hands back a string.

**FIX** Convert the `input` at the moment you read it: value = `int(input("Number: "))`.

```
price = "10"
print(price - 5)
```

```
14 ValueError: invalid literal for int() with base 10:
    ↳ 'twenty'
```

**WHAT IT MEANS** The type was acceptable in principle, since `int()` does take a string, but this particular value does not spell a whole number.

**FIX** Validate before converting, or wrap the conversion in try and `except ValueError` and ask again.

```
int("twenty")
```

```
15 ValueError: invalid literal for int() with base 10:
    ↳ '3.9'
```

**WHAT IT MEANS** The text spells a number, but not a whole one. The `int()` function refuses text with a decimal point rather than guessing what to do with it.

**FIX** Convert with `float()` first, then apply `int()` to that result if you truly want the whole part.

```
int("3.9")
```

```
16 ValueError: could not convert string to float: 'abc'
```

**WHAT IT MEANS** The same problem as the previous two, on the `float()` side. The value does not spell a number of any kind.

**FIX** Handle it with try and `except ValueError`, then `print` a clear message and ask the user again.

```
float("abc")
```

```
17 TypeError: '<' not supported between instances of
    ↳ 'int' and 'str'
```

**WHAT IT MEANS** A number was compared with text. Python has no rule for ordering the two, so it refuses.

**FIX** Convert the `input` to a number before comparing it against a numeric threshold.

```
guess = "12"
print(20 < guess)
```

```
18 ZeroDivisionError: division by zero
```

**WHAT IT MEANS** A division or a remainder operation used zero as the divisor. In beginner programs the divisor is usually `len()` of a collection that turned out to be empty.

**FIX** Check for the empty case before you divide, and report it rather than calculating.

```
values = []
print(sum(values) / len(values))
```

```
19 KeyboardInterrupt
```

**WHAT IT MEANS** Not a defect in the usual sense. It is what you see after pressing Ctrl+C to stop a running program, which is how you escape an accidental infinite loop.

**FIX** Find the reason the loop never ends. A while loop has to change something its condition depends on, on every pass.

```
20 TypeError: unsupported operand type(s) for +:
    ↳ 'NoneType' and 'int'
```

**WHAT IT MEANS** A function was used as though it produced a value, and it has no return statement. A function without one hands back `None`.

**FIX** Add the return statement. Keep in mind that printing a value inside a function is not the same as returning it.

```
def square(number):
    result = number * number

print(square(5) + 1)
```

```
21 TypeError: greet() missing 1 required positional
    ↳ argument: 'name'
```

**WHAT IT MEANS** The function was called with fewer arguments than its definition requires.

**FIX** Supply the missing argument, or give the parameter a default value in the `def` line.

```
def greet(name):
    return "Hello, " + name

print(greet())
```

```
22 TypeError: 'str' object does not support item
    ↳ assignment
```

**WHAT IT MEANS** An attempt was made to change one character of a string in place. Strings are immutable.

**FIX** Build a new string instead, with slicing or `replace()`, and assign it back to the name.

```
word = "cat"
word[0] = "C"
```

```
23 IndexError: string index out of range
```

**WHAT IT MEANS** A character was requested at a position the string does not have. The last valid position is `len(text)` - 1.

**FIX** Check the length before indexing, or use a slice, which never raises this error.

```
print("abc"[5])
```

```
24 AttributeError: 'str' object has no attribute
'append'
```

**WHAT IT MEANS** A list method was called on a string. The message names the type it actually got, which is the fastest clue you have.

**FIX** Print the value and its `type()` at that point. Often a variable holds a different kind of value than you assumed.

```
text = "abc"
text.append("d")
```

```
25 IndexError: list index out of range
```

**WHAT IT MEANS** A position was requested that the list does not have. With four items the valid positions are 0, 1, 2, and 3.

**FIX** Compare the index against `len(items)`. An off-by-one error in a loop is the usual source.

```
items = [1, 2, 3]
print(items[3])
```

```
26 ValueError: list.remove(x): x not in list
```

**WHAT IT MEANS** The `remove()` method needs the value to be present, and it was not. Running the same removal twice is a common way to meet this.

**FIX** Test with an in check first, or use a different structure if repeated removals are expected.

```
scores = [70, 85, 92, 60]
scores.remove(60)
scores.remove(60)
```

```
27 AttributeError: 'list' object has no attribute
'push'
```

**WHAT IT MEANS** The method does not exist under that name. Other languages call it push; Python calls it `append`.

**FIX** Check the name against the quick reference. Spelling and language habits from elsewhere both cause this.

```
items = [1, 2]
items.push(3)
```

```
28 TypeError: 'tuple' object does not support item
    ↳ assignment
```

**WHAT IT MEANS** A `tuple` is an ordered sequence that cannot be modified in place. That is the whole point of choosing one.

**FIX** Build a new `tuple` from the pieces you want, or use a list if the values genuinely need to change.

```
point = (3, 4)
point[0] = 9
```

```
29 TypeError: 'set' object is not subscriptable
```

**WHAT IT MEANS** A set has no positions, so there is no first item to ask for. Sets are about uniqueness and membership.

**FIX** Loop over the set, or test membership with in. Convert to a list with `sorted()` when you need a definite order.

```
colors = {"red", "blue"}
print(colors[0])
```

```
30 ValueError: not enough values to unpack (expected 3,
    ↳ got 2)
```

**WHAT IT MEANS** Unpacking needs exactly as many names on the left as there are items on the right.

**FIX** Count both sides. When the count varies with the data, `print` the sequence first and check its actual length.

```
point = (3, 4)
x, y, z = point
```

```
KeyError: 'age'
```

**WHAT IT MEANS** Bracket access asked a dictionary for a key it does not hold. The message shows the key that was missing.

**FIX** Use `record.get("age")` for a default of `None`, `record.get("age", 0)` for your own default, or test with `"age" in record`.

```
student = {"name": "Alex", "grade": 87}
print(student["age"])
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'journal.txt'
```

**WHAT IT MEANS** Read mode requires the file to exist already. Python looks in the folder the program was started from, not the folder the `.py` file lives in.

**FIX** On a first run this is an expected failure, so wrap the read in try and `except FileNotFoundError` and `print` a message such as No entries yet. Where the file should already exist, check the folder you started the program from.

```
with open("journal.txt", "r", encoding="utf-8") as file:
    print(file.read())
```

```
TypeError: __init__() takes 2 positional arguments but 3 were given
```

**WHAT IT MEANS** A method was defined without `self` as its first parameter. Python passes the object itself as the first argument, so every count is off by one.

**FIX** Add `self` as the first parameter of every method in the class.

```
class Account:
    def __init__(name, balance):
        self.name = name
        self.balance = balance

acct = Account("Ava", 100)
```

```
AttributeError: 'Player' object has no attribute 'nmae'
```

**WHAT IT MEANS** The attribute was never stored under that name. Assigning a misspelled attribute creates a new one rather than raising an error, so the mistake surfaces only when you read it back.

**FIX** Compare the spelling against the assignments in `__init__`.

```
class Player:
    def __init__(self, name):
        self.name = name

p = Player("Ava")
print(p.nmae)
```

```
ModuleNotFoundError: No module named 'randon'
```

**WHAT IT MEANS** The import name does not match any module Python can find. A misspelling is the common cause.

**FIX** Check the spelling. Also make sure no file of your own carries the module's name, since a file named `random.py` in your folder is found before the standard module.

```
import randon
```

```
No error message at all, and the answer is wrong
```

**WHAT IT MEANS** This is a logic error. The code was valid, it ran to the end, and your instructions did not say what you meant.

**FIX** Find the first point where reality differs from your expectation. Print the values and types involved, change one thing, and rerun the same test.