**Section 9 — Typical Errors in Python Programs**

So you’ve written and run your first Python program — and it worked!   
But let’s be honest: programming isn’t always smooth sailing.  
Every programmer — even professionals — makes mistakes.  
In fact, that’s half of what programming *is*: writing code, breaking it, and then figuring out why.

Don’t worry — errors are *not failures*, they’re *feedback*.  
They’re Python’s way of telling you, “Hey, something here doesn’t quite make sense. Let’s fix it together.”

So in this section, we’ll look at the **most common Python errors**, what causes them, and how to fix them quickly.

**9.1 What Are Errors (a.k.a. “Bugs”)?**

In programming, when something goes wrong, Python stops your program and shows an **error message**.  
These messages might look scary at first — a wall of red text — but they’re actually clues.  
Each line tells you *where* the problem happened and *what kind* of problem it is.

There are two main types of errors you’ll encounter in Python:

1. **Syntax Errors**
2. **Runtime Errors (Exceptions)**

Let’s look at both in detail

**9.2. Syntax Errors – The Grammar Mistakes**

A **syntax error** happens when Python doesn’t understand your code — like a grammar mistake in English.

Think of Python as reading your code like a sentence.  
If something’s out of order, missing a punctuation mark, or spelled wrong, it can’t read it and throws a **SyntaxError**.

**Example:**

print("Hello, world!"

⚠️ Missing the closing parenthesis.

When you run this, Python says:

SyntaxError: unexpected EOF while parsing

“EOF” means *End of File* — Python reached the end of your code but expected something else (like a missing )).

**🧠 How to Fix It:**

Always check for missing parentheses (), colons :, or quotation marks ".  
VS Code helps by highlighting syntax errors before you even run the program — just look for the red underlines.

**9.3. Runtime Errors (Exceptions) – Mistakes While Running**

A **runtime error**, also called an **exception**, happens *after* your code has started running — when something goes wrong during execution.

Python understands the code, but it can’t actually *do* what you asked for.

**Example:**

print(10 / 0)

⚠️ You’re dividing by zero, which is mathematically impossible.

Python says:

ZeroDivisionError: division by zero

The good thing? The error message tells you *exactly* what went wrong.

**Other Common Errors You’ll See**

Let’s look at a few more that every beginner encounters:

**9.3.1. NameError**

Happens when you try to use a variable that hasn’t been defined yet.

print(name)

Output:

NameError: name 'name' is not defined

**Fix:** Make sure you define variables *before* using them.

**2. IndentationError**

Python uses **indentation (spaces or tabs)** to structure code.  
If something is misaligned, Python will complain.

if True:

print("Hello")

Output:

IndentationError: expected an indented block

🧩 **Fix:** Always indent code that belongs inside loops, functions, or conditions (usually 4 spaces).

**3. TypeError**

Happens when you mix incompatible data types.

age = 25

print("I am " + age + " years old.")

Output:

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

**Fix:** Convert the number to a string:

print("I am " + str(age) + " years old.")

**4. ValueError**

Python understood the type of data, but the value doesn’t make sense.

number = int("hello")

Output:

ValueError: invalid literal for int() with base 10: 'hello'

**Fix:** Only convert strings that actually represent numbers, e.g. int("123").

**5. IndexError**

Happens when you try to access a position in a list that doesn’t exist.

numbers = [10, 20, 30]

print(numbers[5])

Output:

IndexError: list index out of range

**Fix:** Check that your index is within range — lists start at 0!

**9.4 Bonus: Logical Errors**

Sometimes your code runs perfectly fine — no syntax or runtime errors — but the **output is wrong**.  
That’s called a **logical error**, and Python can’t detect it.

**Example:**

# supposed to multiply

print(2 + 2) # oops!

The code runs without error, but it doesn’t do what you intended.  
Fixing logical errors requires testing, debugging, and understanding your logic clearly.

**9.5 How to Read Error Messages Like a Pro**

Error messages might look intimidating at first, but here’s how to read them:

1. **Start at the bottom line.**  
   That’s usually where Python tells you *what* the error is.  
   Example: TypeError: unsupported operand type(s) for +: 'int' and 'str'
2. **Look at the file and line number above.**  
   It’ll say something like File "main.py", line 5. That’s where the problem is.
3. **Check the red underlines in VS Code.**  
   They often show you exactly where the mistake happened — before you even run the code.

**9.6 Tips to Avoid Common Mistakes**

* Always close parentheses (), brackets [], and quotes "".
* Pay attention to indentation — Python relies on it.
* Give variables clear names and spell them consistently.
* Don’t mix text and numbers without converting.
* Run your code often — small steps make debugging easier.

**In Short**

* Errors are normal — every programmer makes them.
* Syntax errors stop your code from running.
* Runtime errors happen *while* your code runs.
* Logical errors happen when the logic is wrong — even if the code is correct.
* The best way to learn is to *break your code, read the error, and fix it.*

Every red error message you see is just Python trying to teach you something new.

**9.7 Closing for This Section**

So now you know how to recognize and fix typical Python errors — you’re officially debugging like a real programmer!  
In the next section, we’ll go deeper and learn how to **use variables and data types** — the core building blocks of any Python program.

Because once you can store and manipulate data, you can start making your programs actually *think*!

Exercises:

1. Print letters from A to Z;
2. Print numbers from 1 to 10

10. **Variables in Python**

Now that we know how to read and fix common errors, it’s time to move on to one of the most important concepts in programming — **variables**.  
If code is the language we use to *talk* to the computer, then **variables** are the way we help it *remember things*.

Every program you’ll ever write — games, apps, websites, or AI — uses variables.  
They’re the foundation of everything in Python.

Let’s break it down step by step. 🧠

**10.1 What Is a Variable?**

A **variable** is like a box where you can store a piece of information — a number, a word, anything — and give it a name.  
Later, you can open that box and use or change what’s inside.

Think of it as a *label* you attach to some data so you can refer to it easily.

Example:

name = "Alice"

age = 25

Here, we created two boxes:

* name holds the text "Alice"
* age holds the number 25

From now on, whenever Python sees the word name, it knows it’s talking about "Alice".

**10.2 How Variables Work in Memory**

When you create a variable, Python does three things:

1. **Creates a space in memory** to store the value.
2. **Stores the data** in that space.
3. **Links your chosen name** (like age) to that data.

It’s like writing a person’s name on a sticky note and attaching it to their backpack — the name refers to the contents.

**10.3 Creating a Variable**

In Python, creating a variable is super simple — no special keywords or symbols needed.  
You just write:

variable\_name = value

The = sign doesn’t mean “equals” like in math — it means **assignment**.  
You’re assigning the value on the right to the name on the left.

**Example:**

city = "New York"

temperature = 21

is\_raining = False

Here, we stored three different types of data in variables:

* A **string** ("New York")
* A **number** (21)
* A **boolean** (False)

**10.4 Using Variables**

Once you’ve stored data in a variable, you can use it anywhere in your code.

name = "Alice"

print("Hello,", name)

Output:

Hello, Alice

You can even combine variables together or use them in calculations:

x = 10

y = 5

result = x + y

print(result)

Output:

15

Here, Python took the values of x and y, added them, and stored the result in a new variable.

**10.5 Updating Variables**

You can change the value of a variable anytime — Python simply replaces the old value with the new one.

score = 0

print("Score:", score)

score = 10

print("Score:", score)

Output:

Score: 0

Score: 10

The variable score was updated from 0 to 10.  
This is how games, apps, and programs keep track of changing information — like your score, progress, or settings.

**10.6 Variable Naming Rules**

Python is flexible, but it still has a few rules for naming variables properly:

✅ You **can use**:

* Letters (a–z, A–Z)
* Numbers (0–9)
* Underscores (\_)

🚫 You **cannot start** a variable name with a number.  
🚫 You **can’t use spaces or special characters** (!, @, #, etc.).

**Examples:**

✅ user\_name, total\_score, age2  
❌ 2player, user name, total-score

Also, variable names are **case-sensitive**:

name = "Alice"

Name = "Bob"

Here, name and Name are *two different variables!*

**10.7 Naming Tips (Best Practices)**

To make your code easy to read, use clear, descriptive names:

* ✅ user\_age is better than ua
* ✅ total\_price is better than tp

Python programmers usually use **snake\_case** (lowercase with underscores).  
Example: my\_first\_variable

10.8 C**ommon Mistakes with Variables**

1. Using a variable before defining it:

print(age)

age = 20

👉 Error: NameError: name 'age' is not defined

1. ❌ Forgetting quotes around text:

name = Alice

👉 Error: NameError: name 'Alice' is not defined  
✅ Fix: name = "Alice"

1. ❌ Mixing variable types incorrectly:

age = 20

print("Age: " + age)

👉 Error: TypeError  
✅ Fix: print("Age: " + str(age))

**10.9 Multiple Variables in One Line**

You can assign several variables at once if you want:

a, b, c = 10, 20, 30

print(a, b, c)

Output:

10 20 30

Or assign the same value to multiple variables:

x = y = z = 100

**10.10 Variables in Action**

Let’s combine what we’ve learned in a small example:

# A simple greeting program

first\_name = "Alex"

last\_name = "Johnson"

age = 21

print("Hello,", first\_name, last\_name)

print("You are", age, "years old.")

Output:

Hello, Alex Johnson!

You are 21 years old.

Notice how easy it is to reuse and combine variable values.  
This is the foundation of making interactive, dynamic programs.

**10.11 Summary**

* A **variable** stores information that your program can use and modify.
* You create one using name = value.
* Variables can change as your program runs.
* Clear, descriptive names make your code readable.

Think of variables as the *memory* of your program — they let your code keep track of information and make decisions based on it.

**10.12 Closing for This Section**

You’ve just mastered one of the most important ideas in all of programming — **variables**.  
Everything from simple scripts to massive applications depends on them.

In the next section, we’ll go one level deeper and explore **data types** — the kinds of information your variables can hold, like numbers, text, and more.

Understanding data types will unlock your ability to perform calculations, comparisons, and logic in Python. 🧠💻

11. **Primitive Data Types in Python**

Now that we understand **variables**, it’s time to look inside them — to see what kind of information they actually hold.

You see, not all data is the same. Some are numbers, some are text, and others are true-or-false values.

These different forms of information are called **data types**, and they’re what make programming possible.  
Every time you create a variable, Python quietly decides *what type* of data you’re storing — and how it should treat it.

11.1 **What Are Data Types?**

A **data type** tells Python what kind of value a variable holds.  
Just like a chef needs to know whether an ingredient is sugar or salt before cooking, Python needs to know whether it’s working with text, numbers, or logic before running your code.

Without data types, Python wouldn’t know how to add, compare, or print anything correctly.

**11.2 Primitive Data Types in Python**

Python has a few basic, or **primitive**, data types — the building blocks for all other types.  
Let’s look at the main ones:

**1️⃣ Integers (int)**

Whole numbers — no decimals, just positive or negative values.

age = 25

year = 2025

temperature = -5

✅ You can do arithmetic operations with integers:

print(10 + 5) # 15

print(8 \* 3) # 24

print(9 - 2) # 7

Python automatically treats these values as type **int**.

**2️⃣ Floating Point Numbers (float)**

Numbers with decimal points — used for more precise values like measurements or prices.

price = 19.99

height = 1.75

score = -3.5

✅ You can mix integers and floats in calculations:

result = 10 + 2.5

print(result) # 12.5

When you combine them, Python automatically converts the result into a float.

**3️⃣ Strings (str)**

Text data — anything written inside quotes.  
Strings can hold letters, numbers, symbols, even spaces.

name = "Alice"

message = 'Welcome to Python!'

✅ You can join (concatenate) strings:

greeting = "Hello, " + name

print(greeting)

Output:

Hello, Alice

✅ Or use *f-strings* for cleaner formatting:

print(f"Hello {name}, welcome to Python!")

Output:

Hello Alice, welcome to Python!

Strings are everywhere — in user input, printed text, filenames, and more.

**4️⃣ Booleans (bool)**

Logical values that represent **True** or **False** — the foundation of all decisions in programming.

is\_raining = True

is\_sunny = False

✅ You’ll often use Booleans in conditions:

if is\_raining:

print("Take an umbrella!")

else:

print("Enjoy the sunshine!")

Output (if is\_raining = True):

Take an umbrella!

Booleans help your program make choices and react differently depending on the situation.

**13.3 Checking a Variable’s Type**

You can always check what type of data a variable holds using the built-in type() function.

x = 10

y = 3.14

z = "Hello"

w = True

print(type(x))

print(type(y))

print(type(z))

print(type(w))

Output:

<class 'int'>

<class 'float'>

<class 'str'>

<class 'bool'>

This is super useful when debugging or learning how Python treats your data.

**13.4 Converting Between Data Types (Type Casting)**

Sometimes, you’ll want to change one data type into another — for example, turning a string "10" into a number 10.

x = "10"

y = int(x) # convert string to integer

z = float(x) # convert string to float

print(y + 5) # 15

print(z + 2.5) # 12.5

You can also go the other way around:

age = 25

print("Age: " + str(age))

Or convert numbers to Booleans:

print(bool(0)) # False

print(bool(5)) # True

**13.5 Common Mistakes with Primitive Types**

1️⃣ Forgetting to convert data types:

print("Result: " + 10) # ❌ Error

✅ Fix:

print("Result: " + str(10))

2️⃣ Using quotes incorrectly:

age = "25"

print(age + 5) # ❌ Error (string + number)

✅ Fix:

age = 25

print(age + 5)

3️⃣ Misunderstanding Booleans:

print(bool("")) # False

print(bool(" ")) # True

Even a single space makes a string “truthy” in Python!

**Summary**

* Python’s **primitive data types** are the simplest building blocks for all programs:  
  🔹 **int** — whole numbers  
  🔹 **float** — decimals  
  🔹 **str** — text  
  🔹 **bool** — True or False values
* Use type() to check a variable’s data type.
* Use int(), float(), str(), and bool() to convert between them.

Understanding these four types is crucial — they’re the DNA of every Python program you’ll ever write.

**Closing for This Section**

Now that you know Python’s primitive data types, you can store, modify, and mix different kinds of information confidently.

**Section 12 — Arithmetic Operators in Python**

Now that you know about Python’s **primitive data types** — numbers, text, and booleans — it’s time to make them *do something!*

Python isn’t just about storing data; it’s about **performing operations** — calculations, logic, and transformations.  
And that starts with **arithmetic operators** — the mathematical tools that let you calculate, count, and compute results.

Let’s dive in and see how Python handles math like a pro.

**12.1 What Are Arithmetic Operators?**

Arithmetic operators are symbols that tell Python to perform basic mathematical operations — things like addition, subtraction, multiplication, and more.

You’ve used these symbols in math before — Python uses them in almost the same way!

**12.2 The Main Arithmetic Operators in Python**

| **Operator** | **Name** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 10 - 4 | 6 |
| \* | Multiplication | 6 \* 2 | 12 |
| / | Division | 9 / 3 | 3.0 |
| // | Floor Division | 10 // 3 | 3 |
| % | Modulus (Remainder) | 10 % 3 | 1 |
| \*\* | Exponentiation (Power) | 2 \*\* 3 | 8 |

These seven operators form the core of Python’s arithmetic toolkit.  
Let’s explore how each one works in action.

**1. Addition (+)**

Adds two numbers together.

a = 10

b = 5

result = a + b

print(result)

Output:

15

You can also use + to join strings:

name = "Python"

version = "3"

print(name + " " + version)

Output:

Python 3

**2. Subtraction (-)**

Subtracts one number from another.

a = 10

b = 3

print(a - b)

Output:

7

Negative results are perfectly fine in Python.

**✖️ 3. Multiplication (\*)**

Multiplies two numbers together.

x = 4

y = 7

print(x \* y)

Output:

28

You can even multiply strings to repeat them:

print("Hi! " \* 3)

Output:

Hi! Hi! Hi!

**➗ 4. Division (/)**

Divides one number by another — always returns a **float**, even if it divides evenly.

x = 9

y = 3

print(x / y)

Output:

3.0

Notice that the result is 3.0, not 3 — Python always keeps division results precise.

**🧮 5. Floor Division (//)**

Performs division but **drops the decimal part** — it gives the *whole number result only*.

x = 10

y = 3

print(x // y)

Output:

3

This is useful when you only need whole-number results, like counting items or dividing into groups.

**🔢 6. Modulus (%)**

Returns the **remainder** after division.

x = 10

y = 3

print(x % y)

Output:

1

The % operator is super handy for checking if a number is even or odd:

number = 7

print(number % 2 == 0)

Output:

False

Because 7 divided by 2 leaves a remainder of 1 — it’s not even.

**⚡ 7. Exponentiation (\*\*)**

Raises one number to the power of another.

x = 2

y = 3

print(x \*\* y)

Output:

8

Meaning 2 \*\* 3 → 2 × 2 × 2 = 8

You can use this for powers, squares, and roots:

print(5 \*\* 2) # 25 (5 squared)

print(9 \*\* 0.5) # 3.0 (square root of 9)