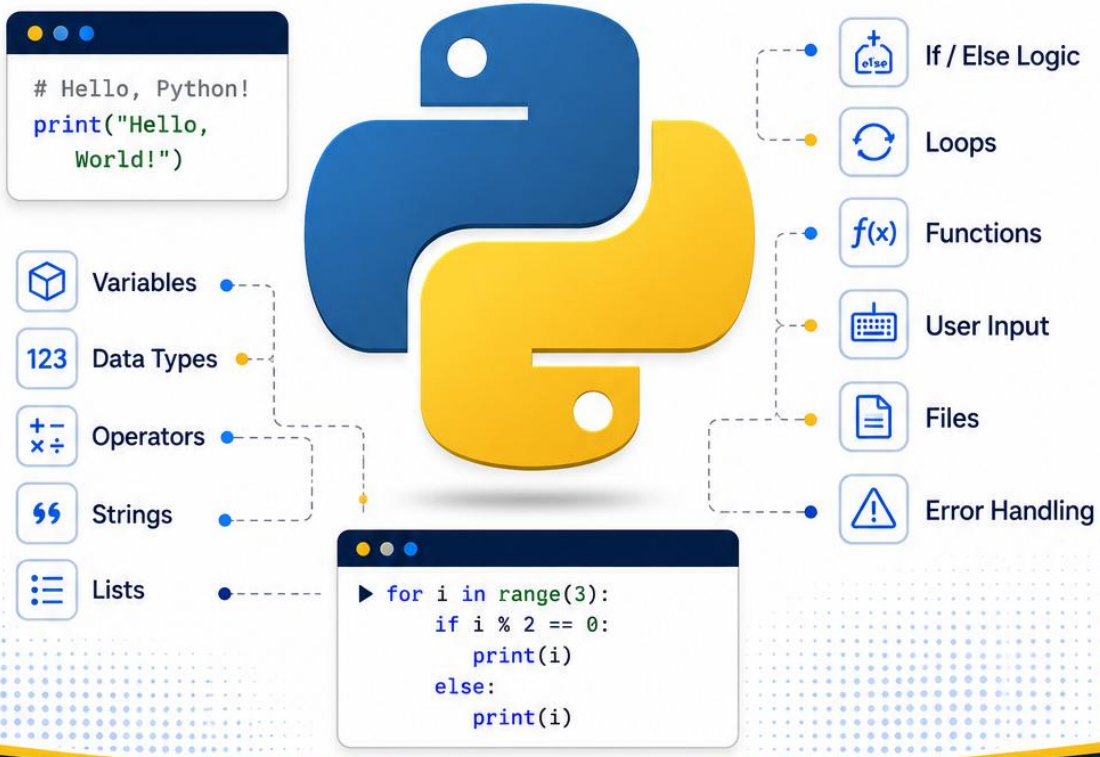
Python Cheat Sheet for Absolute Beginners

Your Fast Reference to Variables, Loops, Lists, Functions, and Files



RUTH BARNHART

Python Cheat Sheet for Absolute Beginners

First edition, 2025.

Table of Contents

Disclaimer

The code and examples in this book are provided for educational purposes only and are offered as is, without warranty of any kind, either expressed or implied. While every effort has been made to keep the examples accurate and up to date, the author and publisher accept no responsibility for any errors, omissions, or for any loss or damage arising from the use of the information it contains.

Python and any other product or company names mentioned may be trademarks of their respective owners and are used for reference only.

Foreword

If you have ever sat in front of a blank screen, certain that everyone else simply gets this, you are exactly who this book is for. Learning to code is not about having a special kind of brain. It is about knowing a handful of patterns and having somewhere to look them up when they slip your mind.

That is what this cheat sheet is. It is the companion you keep beside the main crash course while you practice. The course walks you through the ideas day by day. This book sits next to your keyboard, so that when you think how did I write a loop again, the answer is only a page away.

You do not need to read it from cover to cover. Jump to the part you need, copy the example, run it, then change one small thing to see what happens. Every chapter is short on purpose, because the goal is to get you back to writing real code as quickly as possible.

Be patient with yourself. Every programmer you admire was once a beginner staring at an error message that made no sense. The only real difference is that they kept going. Keep this book open, keep experimenting, and you will be surprised how quickly the strange starts to feel familiar.

Happy coding.

Chapter 1: How to Use This Cheat Sheet

Think of this cheat sheet as the notebook you keep open next to your keyboard. It is not a course to read front to back. It is a fast lookup for when you know what you want to do and just need the Python for it.

1.1 What This Cheat Sheet Is For

Reach for it when a quick question pops up, such as:

- How do I create a variable?
- How do I write an if statement?
- How does a for loop work?
- How do I open a file?
- How do I keep my program from crashing?

Every answer is short. Find it, use it, move on.

1.2 How to Read the Examples

Each example is small and written for someone brand new to coding. Do not just read them. Open VS Code, type each snippet yourself, and run it. Typing code by hand is what makes the patterns stick.

1.3 The Copy, Run, Change Method

The fastest way to learn from an example is to play with it:

- Copy the code
- Run it
- Change one thing
- Run it again
- Notice what changed

Try it now with these two lines:

```
print("Hello!")
print("I am learning Python.")
```

Run them, then change the words inside the quotes and run them again. That small loop is how programmers learn.

1.4 What to Do When Something Breaks

Errors are normal. Every programmer sees them daily, and they do not mean you are failing. When your code breaks, take a breath and read the message. For common errors and simple fixes, see the chapter on files, user input, and error handling. You do not need to memorize this book. You just need to know where to look.

Chapter 2: Python Syntax Basics

These are the ground rules that every Python file follows. None of them are hard, but they trip up almost every beginner at least once. Get them right and most of your early errors disappear before they even start.

2.1 Running a Python File

Writing code is only half the job. To see it work, you have to run it. Save the file first, then open the terminal, move into the folder where the file lives, and run it by name. The terminal is just a text window where you type commands to your computer.

```
python filename.py
```

On Mac or Linux, the command is usually python3 instead of python:

```
python3 filename.py
```

- The file must end in .py
- Save before you run, or you will run the old version
- The terminal must sit in the same folder as your file

2.2 Printing Output

The print function is how your program talks back to you, and it is the tool you will reach for most often. You can print text, numbers, or both at once.

```
print("Hello, world!")
print(5 + 3)
print("Age:", 30)
```

Use a comma to keep items separate and let Python add the space for you. Use the plus sign only when you are joining pieces of text yourself, and only between text and text.

2.3 Comments

A comment is a note for humans, not for the computer. Python ignores everything after a number sign on that line, so a comment never changes how your code runs.

```
# This is a comment
print("This line runs")
```

Use comments to explain a tricky part, leave yourself a reminder for later, or temporarily switch off a line while you test something.

2.4 Indentation Rules

In many languages the spacing is just for looks. In Python it carries meaning. Python reads the indentation to decide which lines belong together, so the indented line below runs only when the condition is true.

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

- Indent with four spaces
- Keep the same indentation for every line inside a block
- Never mix tabs and spaces in the same file, or you will get an error

2.5 Parentheses, Quotes, and Colons

A few small symbols show up again and again. Once you learn to spot this pattern, most Python lines start to look familiar:

```
print("text")
input("question")
if condition:
for item in list:
def function_name():
```

Text sits inside quotes, function calls use parentheses, and any line that opens a block ends with a colon.

2.6 Beginner Syntax Mistakes

Most early errors come from a small handful of slips. When a file refuses to run, scan this list before anything else:

- Missing colon at the end of an if, for, or def line
- Mismatched or missing quotes around text
- Forgetting the parentheses after print or input
- Wrong or uneven indentation inside a block
- Smart quotes pasted from a document instead of straight quotes

The fix is almost always small. Read the line number in the error message, check that one line, and correct a single thing at a time.

Chapter 3: Variables and Data Types

Every program stores information so it can use it later. In Python you store information in variables, and every value has a type that tells Python how to treat it. This chapter is your quick reference for naming things, storing data, and converting between types.

3.1 Variables

A variable is a label for a piece of information. You pick a name, use a single equals sign, and store a value behind it. From then on, the name stands in for that value wherever you use it, which saves you from typing the same thing over and over.

```
name = "Alex"
age = 30
price = 9.99
is_active = True
```

You can give a variable a new value at any time, and the new value simply replaces the old one. Names are also case sensitive, so age and Age are two different variables.

3.2 Naming Rules

Clear names make code easy to read weeks later, when you have forgotten what you were thinking. Python has a few firm rules, plus a couple of habits worth copying from day one.

- Use clear, descriptive names that say what the value is
- Use lowercase letters with underscores between words
- No spaces inside a name
- Do not start a name with a number
- Do not use words Python reserves, such as if, for, or print

```
user_name = "Mia"
total_price = 49.95
```

3.3 Strings

A string is text. You wrap it in quotes, single or double, as long as the opening and closing quotes match. You can join strings together with the plus sign to build a longer message.

```
message = "Hello"
name = "Sarah"
print(message + " " + name)
```

3.4 Integers and Floats

Numbers come in two everyday types. An integer is a whole number. A float has a decimal point. Python decides which one you mean from the way you write the value, so do not add commas to large numbers.

```
count = 10
price = 4.99
```

3.5 Booleans

A boolean is a simple yes or no value, written as True or False with a capital letter. Booleans quietly power almost every decision your program makes, from checking a login to ending a loop.

```
is_logged_in = True
has_paid = False
```

3.6 Type Conversion

Sometimes you need to change one type into another, for example turning text typed by a user into a number you can do math with. The functions int, float, and str do the converting for you.

```
age = int("25")
price = float("9.99")
text = str(100)
```

3.7 Checking a Type

When you are not sure what type a value is, ask Python directly with the type function. This is one of the fastest ways to track down a confusing error, because many errors come from a value being the wrong type.

```
print(type(age))
```

3.8 Beginner Mistakes with Data Types

The most common slip is trying to join text and a number with the plus sign. Python will not guess what you mean, so it stops and raises an error instead.

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

The fix is to convert the number to text first with str, so both sides of the plus sign are the same type:

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

Now both sides are text, the plus sign just joins them, and the message prints cleanly.

Chapter 4: Operators and Comparisons

Operators are the symbols that let your code do math and make comparisons. This chapter gathers the ones you will use most often, all in one place, so you can copy the pattern you need and move on.

4.1 Math Operators

These operators handle everyday arithmetic.

Operator	Meaning	Example
+	add	5 + 2
-	subtract	5 - 2
*	multiply	5 * 2
/	divide	5 / 2
//	floor divide	5 // 2
%	remainder	5 % 2
**	power	5 ** 2

One thing to remember: the single slash always gives a decimal result, so 5 / 2 is 2.5, while the double slash keeps only the whole part, giving 2.

4.2 Assignment Operators

Assignment operators update a variable based on its current value, which saves you from writing the whole calculation out again. The same shortcut works for the other math operators too, such as times-equals and divide-equals.

```
score = 10
score += 5
score -= 2
```

After these three lines, score holds 13.

4.3 Comparison Operators

Comparison operators ask a true or false question about two values, so the answer is always a boolean.

Operator	Meaning
==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

4.4 Logical Operators

Logical operators let you combine conditions into one decision. Use and when both parts must be true, or when at least one must be true, and not to flip a condition to its opposite. They are how a single if statement can weigh more than one thing at once.

```
if age >= 18 and has_id:
    print("Allowed")

if is_admin or is_owner:
    print("Access granted")

if not is_logged_in:
    print("Please log in")
```

4.5 Common Operator Confusion

The single most common mix-up is using one equals sign where you need two. One equals sign stores a value. Two equals signs compare two values.

```
age = 18
age == 18
```

The first line sets age to 18. The second asks whether age is equal to 18 and answers True or False. Mixing these up is a classic beginner bug, because the code often still runs but does the wrong thing.

Chapter 5: Strings, Lists, and Built-In Helpers

Text and lists are the two kinds of data beginners work with most. This chapter is your quick reference for shaping text and for storing, reading, and changing lists of values. Keep it open whenever you are wrangling words or collections.

5.1 String Basics

A string holds text. You store it in a variable and print it just like any other value, and you can reuse that variable as often as you like throughout your program.

```
name = "Python"
print(name)
```

5.2 String Methods

Strings come with built-in helpers called methods. You attach a method with a dot right after the string. These are the ones beginners reach for most:

```
text.lower()
text.upper()
text.strip()
text.replace("old", "new")
text.split()
```

- lower makes every letter lowercase
- upper makes every letter uppercase
- strip removes spaces from the start and end
- replace swaps one piece of text for another
- split breaks a sentence into a list of words

One thing to know: a method does not change the original string, it returns a new one. So you usually print the result or save it into a variable. Try this:

```
greeting = "  Hello  "
print(greeting.strip())
```

It prints Hello with the extra spaces removed, while greeting itself still holds the spaced version.

5.3 F-Strings

An f-string lets you drop variables straight into text. Put the letter f before the opening quote, then wrap each variable in curly braces wherever you want it to appear.

```
name = "Alex"
age = 30
print(f"{name} is {age} years old.")
```

Forget the f and Python prints the braces as plain text, so that missing letter is a common thing to check. F-strings are usually cleaner than joining text with plus signs.

5.4 Lists

A list stores several values in order, inside square brackets. You reach any single item by its position in the list, and you can store strings, numbers, or a mix.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])
```

5.5 List Indexing

Counting positions in Python starts at zero, which surprises almost everyone at first. Once it clicks, it stops being confusing.

- The first item is at position 0
- The second item is at position 1
- The last item is at position -1, counting from the end

So with the fruits list above, this line prints cherry:

```
print(fruits[-1])
```

5.6 Adding and Removing Items

Lists can grow and shrink while your program runs. Use append to add an item to the end, and remove to take one out by its value. Trying to remove something that is not there will cause an error, so be sure it exists first.

```
fruits.append("orange")
fruits.remove("banana")
```

5.7 Useful List Functions

Two helpers come up constantly. The len function counts how many items a list has, and sorted gives you the items back in order without changing the original list.

```
len(fruits)
sorted(fruits)
```

5.8 Checking If Something Is in a List

Use the word in to check whether a value is already in a list before you act on it. The check answers True or False, which makes it perfect inside an if statement.

```
if "apple" in fruits:
    print("Found it")
```

5.9 Mini Cheat Sheet: Text and List Tasks

When you just need the right tool for a quick job, scan this short list:

- Make text lowercase: text.lower()
- Remove surrounding spaces: text.strip()
- Split a sentence into words: text.split()
- Count the items in a list: len(my_list)
- Add an item to a list: my_list.append(item)
- Loop through a list: for item in my_list

Each of these is a small building block, and you will combine them again and again as your scripts grow.

Chapter 6: If, Else, Elif, and Logic Flow

Decisions are what turn a flat list of instructions into a real program that reacts. This chapter is your quick reference for letting Python choose what to do based on a condition that is either true or false.

6.1 Basic If Statement

An if statement runs a block of code only when a condition is true. The indented line below runs because 20 is at least 18. If the condition were false, Python would simply skip the indented block and carry on.

```
age = 20
if age >= 18:
    print("You are an adult.")
```

6.2 If and Else

Add else to handle the case when the condition is false. Exactly one of the two blocks runs, never both and never neither.

```
if password == "secret":
    print("Access granted")
else:
    print("Access denied")
```

6.3 Elif

Use elif, short for else if, when you have more than two paths. Python checks each condition in order from top to bottom and runs the first one that is true, then skips the rest.

```
score = 85
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
else:
    print("Keep practicing")
```

With a score of 85, this prints B. The first condition is false, the second is true, and the else block never gets a turn.

6.4 Nested If Statements

You can place an if inside another if for a second question that only matters once the first one is true.

```
if is_logged_in:
    if is_admin:
        print("Admin panel")
```

Keep nesting shallow. More than two levels deep gets hard to read very quickly, so it is often clearer to combine conditions with and instead.

6.5 Truthy and Falsy Basics

Python treats some values as true and others as false even without a comparison. An empty string counts as false, so this checks whether the user actually typed anything:

```
name = ""
if name:
    print("Name entered")
else:
    print("No name entered")
```

Zero, an empty string, and an empty list all count as false, which makes this a handy shortcut for checking whether something has a value.

6.6 Common Logic Mistakes

When an if statement behaves oddly, check this list before anything else:

- Using one equals sign instead of two in the condition
- Forgetting the colon at the end of the if, elif, or else line
- Wrong indentation under the if, so the wrong line is inside the block
- Putting conditions in the wrong order, so a broad one runs before a specific one
- Comparing a number to text, such as the number 18 to the word eighteen

Most of these show up as a wrong result rather than a crash, so when the output looks off, read the conditions one line at a time.

Chapter 7: Loops: For, While, Range, Break, Continue

Loops let you repeat work without copying the same lines over and over. This chapter is your quick reference for the handful of tools that handle almost all repetition in Python, from counting numbers to walking through a list.

7.1 For Loops

A for loop repeats a block once for each item in a sequence. This one counts from 0 to 4 and prints each number. The for loop is the one you will use most, because so much of programming is doing the same thing to every item.

```
for number in range(5):
    print(number)
```

7.2 Looping Through a List

More often, you loop over a list and handle each item in turn. The loop variable takes the value of each item, one after another, until the list runs out. This is how you process a whole collection in just two lines.

```
names = ["Ana", "Ben", "Cara"]
for name in names:
    print(name)
```

7.3 Range

The range function builds a sequence of numbers for a loop. It takes up to three values: where to start, where to stop, and how big each step is. The stop value is never included.

```
range(5)
range(1, 6)
range(0, 10, 2)
```

- range(5) gives 0, 1, 2, 3, 4
- range(1, 6) gives 1, 2, 3, 4, 5
- range(0, 10, 2) gives 0, 2, 4, 6, 8

7.4 While Loops

A while loop keeps going as long as its condition stays true. Reach for it when you do not know in advance how many times you need to repeat. You must change something inside the loop so the condition eventually becomes false, or it will never stop.

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

7.5 Break

Use break to leave a loop early, the moment you are done, even if the sequence has more items left.

```
for number in range(10):
    if number == 5:
        break
    print(number)
```

This prints 0 through 4, then stops the instant number reaches 5.

7.6 Continue

Use continue to skip the rest of the current round and jump straight to the next one.

```
for number in range(5):
    if number == 2:
        continue
    print(number)
```

This prints 0, 1, 3, and 4, skipping only the 2.

7.7 Avoiding Infinite Loops

If a while loop never changes its condition, it runs forever. This version is missing the update, so it would print 1 again and again without ever stopping:

```
count = 1
while count <= 5:
    print(count)
```

The fix is to move the counter forward inside the loop, so the condition can finally become false. If you ever start a runaway loop by accident, press Control and C in the terminal to stop it:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

7.8 Loop Cheat Sheet

Not sure which tool to reach for? Use this quick guide:

- Use for when you know what you are looping through
- Use while when something should continue until a condition changes
- Use break to stop a loop early
- Use continue to skip a single round and keep going

Pick the loop that matches the question you are asking, and let break and continue fine-tune how it behaves.

Chapter 8: Functions and Reusable Code

Functions let you write a piece of code once and reuse it as often as you like, instead of copying the same lines around your file. This chapter is your quick reference for defining functions, passing them information, and getting answers back.

8.1 What a Function Does

A function is a named block of code you can run whenever you need it. You write it once, give it a name, and then call that name to run the whole block again, which keeps your code short and tidy.

8.2 Basic Function

You define a function with def, a name, a pair of parentheses, and a colon. The indented lines are its body. Nothing happens until you call it by name.

```
def say_hello():
    print("Hello!")

say_hello()
```

8.3 Functions with Parameters

A parameter lets you hand information to a function, so the same function can work with different values each time you call it.

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

greet("Mia")
```

This call prints Hello, Mia. Call greet again with a different name and you get a different greeting, all from the same function.

8.4 Return Values

Use return to send a value back to wherever the function was called. You can store that value in a variable or print it straight away.

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

total = add(3, 4)
print(total)
```

Here total holds 7, because the function handed that result back. A function without a return still runs, it just does not give you a value to keep.

8.5 Print vs Return

These two are easy to mix up. The print function shows something on the screen and then it is gone. A return hands a value back so the rest of your program can keep using it.

- Use print when you only want to see the result
- Use return when you need the result for more work later

As a rule, calculations usually return, while messages to the reader usually print.

8.6 Default Parameters

Give a parameter a default value and the caller can leave it out. When they do, Python quietly uses the default instead. This is handy when one option is by far the most common.

```
def greet(name="friend"):
    print("Hello, " + name)
```

Calling greet with no name prints Hello, friend.

8.7 Function Naming Tips

A good function name tells you what it does at a single glance, long after you wrote it.

- Start the name with a verb
- Use lowercase letters with underscores between words
- Keep it clear rather than clever

```
calculate_total()
clean_text()
save_file()
```

8.8 Common Function Mistakes

When a function does not behave, check this list first:

- Forgetting the parentheses when you call it
- Forgetting the colon at the end of the def line
- Wrong indentation in the body, so a line is left out
- Using a variable that only exists inside another function
- Printing a value when you actually needed to return it

Most of these are quick to spot once you know to look for them, so treat this as your first checklist when a function will not cooperate.

Chapter 9: Files, User Input, and Error Handling

Reading input, working with files, and handling errors are where beginners hit their first real walls. This chapter brings all three together, because in real scripts they almost always turn up at the same time. Keep it close when you build your first tool that talks to a user or saves something.

9.1 User Input

The input function pauses your program, shows a prompt, and waits for the user to type something and press enter. Whatever they type comes back as text.

```
name = input("What is your name? ")
print("Hello, " + name)
```

The space at the end of the prompt is on purpose, so the typed answer does not sit right against the question mark.

9.2 Converting User Input

This is important: input always gives you a string, even when the user types a number. To do math with it, convert it first with int or float.

```
age = int(input("How old are you? "))
```

Skip the conversion and any math on age will either fail or join text in ways you did not expect.

9.3 Reading a File

To read a file, open it in read mode, pull in the contents, and close it when you are done.

```
file = open("notes.txt", "r")
content = file.read()
print(content)
file.close()
```

The read method brings back the whole file as one long string, which you can then print or search.

9.4 Writing to a File

Open a file in write mode to put text into it. Write mode replaces whatever was there before, so reach for it only when you mean to start fresh.

```
file = open("output.txt", "w")
file.write("Hello file!")
file.close()
```

9.5 Appending to a File

Append mode adds to the end of a file instead of replacing it, which makes it perfect for logs and running notes.

```
file = open("log.txt", "a")
file.write("New entry\n")
file.close()
```

The backslash n at the end is a newline, so each entry lands on its own line instead of running together.

9.6 Using With Open

The cleanest beginner pattern is with open. Python closes the file for you automatically when the block ends, even if something goes wrong along the way. Prefer this over calling close yourself.

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

Once the indented block ends, the file is already closed for you, with nothing left to remember.

9.7 Common File Modes

The mode is the short letter you pass when opening a file:

Mode	Meaning
"r"	read an existing file
"w"	write, replaces the file
"a"	append, adds to the end
"x"	create a brand new file

9.8 Basic Try and Except

Wrap risky code in a try block. If it fails, the matching except block runs instead of the whole program crashing.

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("That was not a valid number.")
```

If the user types letters, int cannot convert them, so the except block runs and your program keeps going.

9.9 Handling FileNotFoundError

Opening a file that does not exist raises a FileNotFoundError. Catch it to show a friendly message instead of a wall of red text.

```
try:
    with open("notes.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("File not found.")
```

9.10 Catching the Error Message

You can catch any error and print its message, which is handy while you are still working out what goes wrong.

```
try:
    result = 10 / 0
except Exception as e:
    print("Error:", e)
```

This prints Error followed by the message division by zero, so you see exactly what Python objected to.

9.11 Common Beginner Errors

These are the error names you will meet most often. Each one points at a different kind of slip:

- SyntaxError: the code is written incorrectly
- NameError: you used a name Python does not recognize
- TypeError: you combined the wrong types
- ValueError: the type is right but the content is wrong
- IndentationError: the spacing does not line up
- FileNotFoundError: the file is not where you said it was

- ZeroDivisionError: you tried to divide by zero

9.12 Safer Beginner Pattern

Here is a small template you can reuse whenever you ask for a number. It asks, converts, and catches a bad entry, all in one tidy block:

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

Lean on patterns like this one. You do not need to memorize every error, only to wrap the risky part and handle the most likely problem.

Chapter 10: Final Beginner Reference Checklist

10.1 Before You Run Your Code

Run through this list before you hit run:

- Did you save the file?
- Does the file end in .py?
- Are your quotes straight quotes, not smart ones?
- Is there a colon at the end of every if, for, and def line?
- Is the indentation even and consistent?
- Are your variable names spelled the same way each time?
- Did you convert input to a number where you need one?

10.2 When Your Code Breaks

When something breaks, work through these steps in order:

- Read the whole error message, not just the last line
- Look at the line number it shows
- Check spelling and capitalization
- Check the indentation around that line
- Print a variable to see what it holds
- Test with simple input
- Fix one thing at a time, then run again

10.3 Final Encouragement

You do not need to memorize this book, and you never will. That is not the goal. The goal is knowing where to look, testing ideas in small steps, and solving little problems one at a time. Do that often enough and you will be coding before you realize it.