

Python Programming

A Topic-by-Topic Learning Perspective

A beginner textbook aligned with the Python topics in KIPS Computer Science with Python 11.

26 focused chapters • 130 solved problems • 130 unsolved problems

Concepts • small examples • execution traces • original practice

ABSTRACT

This book teaches Python from the first script through collections and standard-library modules. Every chapter addresses one topic, beginning with its purpose and mental model, then developing syntax, behaviour, examples and practice. Its learning goal is independent work on the Python question types in KIPS: explain concepts, predict output, repair errors and construct unfamiliar programs. All demonstration and worked-solution programs were executed in Python 3 during preparation.

HOW TO USE THIS BOOK

Read in chapter order on your first pass. First predict a small example, then type it, run it and explain the result. In a worked problem, cover the solution and make a plan before looking. Finally attempt the five unsolved exercises. A good solution includes a clear algorithm, readable code and useful tests. Tasks may combine subparts, but each chapter has exactly five numbered solved and five numbered unsolved problems.

PREREQUISITES AND NOTATION

No previous Python knowledge is assumed. You need basic arithmetic and access to a Python 3 environment. A function call such as `print(value)` asks Python to perform an operation with an argument. A dot, as in `text.lower()`, selects an attribute or method from an object. These ideas are explained again in context. Code blocks are complete unless described as a fragment. Keep their indentation exactly; four spaces form one level.

SOURCE AND STRUCTURE

The KIPS cloud reader was used to review the Python chapter topics, worked-program patterns and exercise formats from Programming with Python through Introduction to Python Modules (printed pages 94-299). This is an original companion, not a transcription or a one-to-one exercise answer key. The supplied Processor Microarchitecture: An Implementation Perspective informed the numbered hierarchy, continuous explanations, restrained typography and captioned figures. Its technical content is not reused.

ACCURACY NOTES

Explanations follow Python 3 semantics where a textbook shortcut could mislead: dictionaries preserve insertion order, floor division rounds down, identity differs from equality, and a triple-quoted string is not a separate comment syntax. Fixed input values make sample outputs reproducible. For random examples, outputs usually show invariant checks rather than a particular random draw.

Contents

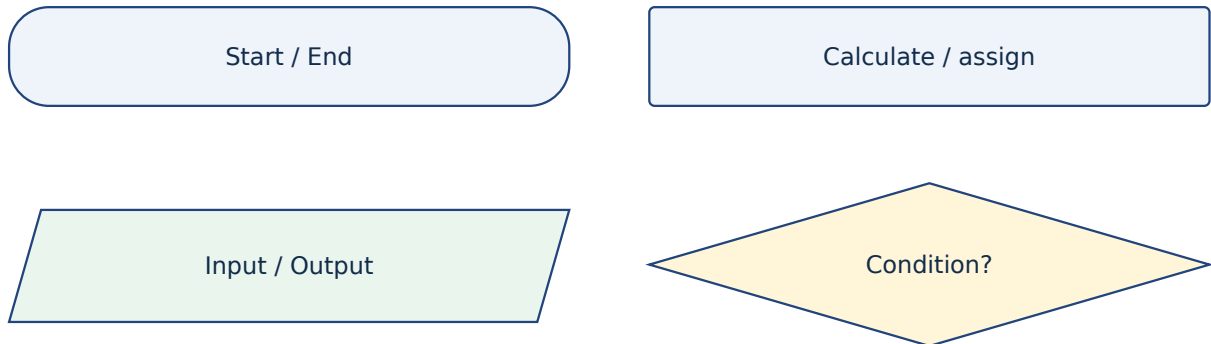
1. Running Python	5
2. Tokens	14
3. Variables	22
4. Data types	30
5. Type conversion	38
6. Input and output	45
7. Operators	53
8. Expressions	61
9. Errors	69
10. Conditional statements	77
11. For loops	86
12. While loops	96
13. Nested loops	106
14. Loop control	114
15. Strings	122
16. Lists	131
17. Tuples	141
18. Dictionaries	149
19. Sets	157
20. Modules	165
21. The math module	172
22. The random module	180
23. The statistics module	188
24. Printing patterns	196
25. Functions	204
26. Classes	214

PART II. KIPS PYTHON THEORY AND PROGRAM PRACTICE 226

Nine units with answered theory, traced programs and new practice

Reading the diagrams and code

Use the arrows to trace execution



Arrows show direction; loop-back arrows mean repeat.

A rounded box marks a start or end. A rectangle represents a calculation or assignment. A slanted box represents input or output. A diamond asks a condition, with labelled exits. A backward arrow repeats a sequence. Concept diagrams about tokens or object references are explicitly labelled and should not be read as executable algorithms.

Teach with a trace

Choose a small input. Place a finger on the starting node, record the variables, follow one arrow at a time and choose a branch only after evaluating its condition. On a loop-back arrow, use the updated values. Compare the chart with the adjacent code and output. Counter-based flowcharts for for loops show the equivalent iteration mechanism rather than extra assignments required in the Python code.

Code formatting

Python keywords are bold purple; strings are green, numbers brown, built-in names blue and comments grey. Shaded inline monospace identifies code fragments inside explanations. Larger shaded blocks contain Python source. Plain output blocks contain the program result, preserving spaces and newlines.

```
for row in range(1, 4):
    # Print one complete row.
    print("*" * row)
```

Exercise format

Each exercise has a separate numbered heading. Read the task, inspect any code block, then provide the requested prediction, repair, explanation or program. Fragments with deliberate errors are labelled. There are five solved problems and five unsolved exercises per chapter.

CHAPTER 1

1. Running Python

A program is a set of instructions. Your first goal is to enter a short program, run it and recognise which text is code and which text is output.

1.1 What Python does

Python is a general-purpose, high-level programming language. High-level means that you can express a task without specifying individual processor instructions. Python is used for automation, data analysis, web applications and many other tasks. Its readable syntax helps beginners, but a program still needs precise punctuation and ordering. Python was created by Guido van Rossum. This book uses Python 3; you do not need to study Python 2.

1.2 Prepare a working environment

Use a school-provided Python 3 environment or obtain Python from python.org. Installation screens vary by operating system, so use the installer instructions for your system. IDLE is a simple editor and interactive shell often included with Python. An editor is where you write source code; the interpreter is what executes it. Create a folder for your practice files. Avoid filenames such as `random.py` and `math.py`, which conflict with standard-library modules used later.

1.3 Interactive mode

In a Python shell, type `print("Hello")` and press Enter. The interpreter executes the statement and displays Hello. The `>>>` symbol is the shell prompt, not part of your code. Typing `4 + 3` in the shell displays `7` because the shell shows expression results. A script does not automatically display a bare expression; use `print` when you want visible output. Interactive mode is useful for testing one small idea.

1.4 Script mode

Open a new editor file, enter `print("Hello")`, save it as `hello.py` and run it with the editor's Run command. A script runs its statements in order. Save after an edit so that you run the intended version. Do not name a file with spaces if you plan to run it from a terminal. If you already use a terminal, `python3 hello.py` is a common command; some Windows setups use `py hello.py` instead. These are terminal commands, not Python statements.

1.5 Your first debugging habits

Use ordinary straight quotes in code, not curved quotation marks copied from a word processor. Match every opening parenthesis with a closing parenthesis. Python is case-sensitive: `print` works, but `Print` is a different name. When the program fails, read the final error message and the indicated line. Make one correction at a time. Begin with tiny working programs and add one idea, then rerun.

1.6 A repeatable learning cycle

Predict what a program should display, run it, compare the output with your prediction, and explain any difference. This predict-run-explain cycle is more useful than copying code without thinking. Comments beginning with `#` let you explain intent to a reader. They are not executed. A program that runs is not automatically correct: its output must match the task. Later chapters introduce input, decisions and repetition one at a time.

1.7 What a function call means

A function is a named operation that can receive inputs, called arguments, and produce a result. In `print("Hello")`, `print` is the function and `"Hello"` is its argument. Some functions mainly perform an action; `print` displays text and returns `None`. Others return useful data: `len("cat")` returns `3`. You can use built-in functions before learning how to define your own. A method is a function-like operation accessed through an object, such as `"cat".upper()`.

1.8 Features and uses in context

Python is free and open source, with a large standard library. It is dynamically typed: the objects used at runtime determine the operations and type checks. The common CPython implementation compiles source into bytecode and executes it in an interpreter; "interpreted" does not mean that no compilation occurs. Python programs can be portable across operating systems, although paths, external libraries and operating-system-specific features may need changes. Python supports procedural, object-oriented and functional styles. It can be embedded in other applications and extended using components written in other languages. Common uses include scripts, web services, education, scientific work and automation. Readability and development speed are strengths; execution speed and memory cost depend on the workload and implementation.

Visual walkthrough: Running Python

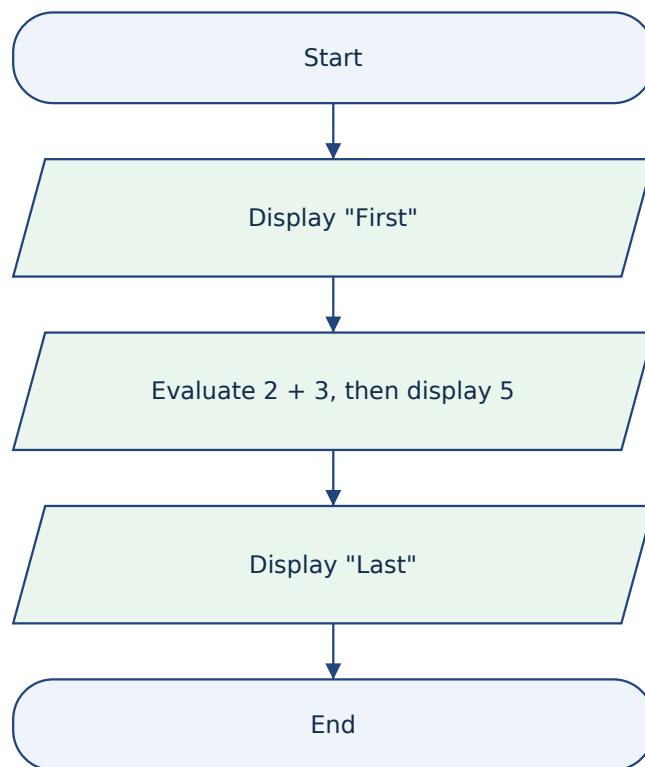


Figure 1.1. Execution follows the statements from top to bottom. Trace this chart with the first example.

1.9 A first example

```
print("First")
print(2 + 3)
print("Last")
```

Output

```
First
5
Last
```

Python executes the three statements in order. The middle statement calculates a value before printing it.

CHAPTER 1 / WORKED WALKTHROUGH

From a question to a running script

Begin by stating the expected output, then identify the values required to produce it. Type a small script, run it, and compare the exact output, including spaces and newlines. Python executes top-level statements from top to bottom; a function definition only stores a function until a later call.

Trace the program

```
print("Start")
score = 7
print("Score:", score)
```

State trace

Before line 1	Nothing printed
After line 1	Start
After line 3	Score: 7

A common wrong turn

A `SyntaxError` prevents execution of that file; a later runtime error may occur after earlier prints. Trace both the line and the state when debugging.

Check your understanding

Predict the output before running. Change score to 0 and check whether the output format changes.

1. Concept workshop: Predict, run, explain

A successful run is evidence about one input and one path, not proof that every version of a program is correct. Begin with a prediction written independently of the interpreter. Run the file, compare the complete transcript, and explain each difference by pointing to the statement that produced it. This habit matters even for a two-line program: it separates what the source says from what you expected it to say. A comment is for the reader and does not execute; a string passed to print is data and does. If an error appears, identify whether parsing failed before execution or a reached instruction failed during execution.

Execution model



Use the model before the code

- 1.1 The task asks for two separate lines, so use two print calls in the required order.
- 1.2 An unquoted arithmetic expression is evaluated before print receives it.
- 1.3 Read the error message from the indicated line.
- 1.4 List the three statements and update an output transcript after each print.
- 1.5 A comment documents intent for a human reader.

Boundary habit

Predict the output before running. Change score to 0 and check whether the output format changes.

Challenge the model

- 1.1 Check the exact spaces, capital letters and newline after each call. An empty quoted string would still print a blank line.
- 1.3 Repair one defect at a time. The resulting two calls should both execute without a `SyntaxError` or `NameError`.
- 1.5 Remove the comment and check that the computed result is identical.

1. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 1.1

Write and run a two-line greeting.

Reasoning and algorithm

The task asks for two separate lines, so use two print calls in the required order. Write the expected transcript before typing the program; each call supplies one line.

Program

```
print("Hello, learner!")  
print("Python is ready.")
```

Output

```
Hello, learner!  
Python is ready.
```

Why this result follows

Each print call produces one line. The quoted words are data supplied to print.

Boundary and error check

Check the exact spaces, capital letters and newline after each call. An empty quoted string would still print a blank line.

Solved problem 1.2

Predict the difference between quoted and unquoted arithmetic.

Reasoning and algorithm

An unquoted arithmetic expression is evaluated before print receives it. Quoted characters are a string, so Python displays them literally.

Program

```
print(7 + 4)  
print("7 + 4")
```

Output

```
11  
7 + 4
```

Why this result follows

Without quotes Python calculates. Inside quotes the characters are text and are printed unchanged.

Boundary and error check

Change one operand and predict the numeric line while the quoted line stays the same.

Solved problem 1.3

Repair the incorrect name Print and a missing closing parenthesis.

Reasoning and algorithm

Read the error message from the indicated line. Python names are case-sensitive, and every opening parenthesis needs a closing one.

Program

```
print("Ready")
```

Output

```
Ready
```

Why this result follows

The corrected statement uses the exact built-in name print and matches both its quotes and parentheses.

Boundary and error check

Repair one defect at a time. The corrected single call executes and prints Ready.

Solved problem 1.4

Check the order of three actions.

Reasoning and algorithm

List the three statements and update an output transcript after each print. Assignments can affect later output but cannot change text already printed.

Program

```
print("Start")  
print("Working")  
print("Finish")
```

Output

```
Start  
Working  
Finish
```

Why this result follows

In this straight-line program, execution follows the order of the statements from top to bottom.

Boundary and error check

Swap the first and last statements and predict the new transcript before running.

Solved problem 1.5

Add a comment explaining a calculation.

Reasoning and algorithm

A comment documents intent for a human reader. Put it before the calculation or after a complete statement, outside quoted text.

Program

```
# A week contains seven days.
print(3 * 7)
```

Output

```
21
```

Why this result follows

The comment explains the meaning of 7. Only the print statement generates output.

Boundary and error check

Remove the comment and check that the computed result is identical.

Trace the source, not the comment

Step	Python action	Visible result
1	Read comment line	No operation
2	Evaluate <code>3 * 7</code>	21
3	Call <code>print(21)</code>	21 and newline

Try replacing the comment with `# print(999)`: the output remains 21. Put `#` inside quotes and it becomes printed data.

Change one token and predict

Variant	What Python sees	Output
<code># print(9)</code>	Only a comment	No output
<code>print("#")</code>	A string containing <code>#</code>	<code>#</code>
<code>print(3 * 7) # note</code>	Call, then comment	21
<code>print("3 * 7")</code>	Literal characters	<code>3 * 7</code>

Decision rule: determine whether characters are inside quotes, then ignore any unquoted `#` and its following text.

1. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 1.1

Explain the difference between the shell prompt, source code and output. Identify which of these belongs in a script: `>>>`, `print("Hi")`, `Hi`.

Exercise 1.2

Predict the output, then run the two statements.

```
print(6 * 2)
print("6 * 2")
```

Exercise 1.3

Repair this intentionally incorrect statement and explain both mistakes.

```
Print("My first script".
```

Exercise 1.4

Create a script that displays your name, class and favourite subject on separate lines. Save it and run it twice.

Exercise 1.5

Use a comment to explain a program that prints the number of hours in two days. Change it to three days and rerun.

CHAPTER 2

2. Tokens

Learn the small pieces from which Python statements are built. Recognising them makes syntax and error messages less mysterious.

2.1 The vocabulary of source code

A token is a meaningful unit recognised while reading source code. In `total = price + 5`, `total` and `price` are identifiers, `5` is a numeric literal, `+` is an operator and `=` introduces assignment. Punctuation such as parentheses and commas organises a statement. A line can contain several token categories at once. Spaces often separate tokens; spaces inside a quoted string are part of its value.

2.2 Keywords and identifiers

Keywords have a meaning defined by Python, including `if`, `else`, `for` and `while`. You cannot use an ordinary reserved keyword as a variable name. An identifier is a name you choose. For beginner programs, use letters, digits and underscores, starting with a letter or underscore. `total_2` is valid; `2_total` is not. Names are case-sensitive. Avoid built-in names such as `print` and `list` even though they are not reserved keywords, because rebinding them hides useful functionality.

2.3 Literals

A literal writes a value directly in source code: `18` is an integer literal, `2.75` a floating-point literal and `"desk"` a string literal. `True`, `False` and `None` are special keyword values. Single and double quotes both create strings; matching them is essential. Use the other quote style when text contains an apostrophe. Triple-quoted strings can span lines. They are strings, not a separate Python comment syntax.

2.4 Numeric spellings and escapes

Integers can be written in decimal or with prefixes: `0b` for binary, `0o` for octal and `0x` for hexadecimal. `0b1010` and `10` represent the same integer. An exponent notation such as `2e3` represents a float. A `j` suffix denotes an imaginary component. In strings, backslash-`n` represents a newline and backslash-`t` a tab. Two backslashes represent one literal backslash. You will practise these in the output chapter.

2.5 Delimiters, comments and indentation

Parentheses group expressions and enclose call arguments. Brackets enclose list elements or indexes, braces enclose dictionaries or sets, commas separate items and colons introduce blocks or appear in slices. You will learn each context before using it. `#` begins a comment outside a string and continues to the end of the physical line. Leading whitespace controls block structure; do not add arbitrary indentation to a top-level statement.

2.6 Literal details that appear in questions

A leading zero is not a general way to write a decimal integer: write `8`, not `08`. Use `0o10` when you mean octal. An imaginary literal such as `4j` is valid; bare `j` is a variable name and must be defined before use. Empty strings are valid. A raw-string prefix `r` changes how most backslash escapes are treated, but a raw string cannot end with an odd number of backslashes. Do not confuse an empty string with a string containing one space.

2.7 Looking up reserved words

You do not need to guess whether a name is reserved. After learning imports in Chapter 20, you can use `import keyword` and `keyword.iskeyword("while")` to test a hard keyword, or inspect `keyword.kwlist`. Some newer Python words have special meaning only in particular grammatical contexts and are called soft keywords. For beginner variable names, avoid words already used by Python syntax. `True`, `False` and `None` begin with capital letters; writing `true` does not produce the Boolean `True`.

Visual walkthrough: Tokens

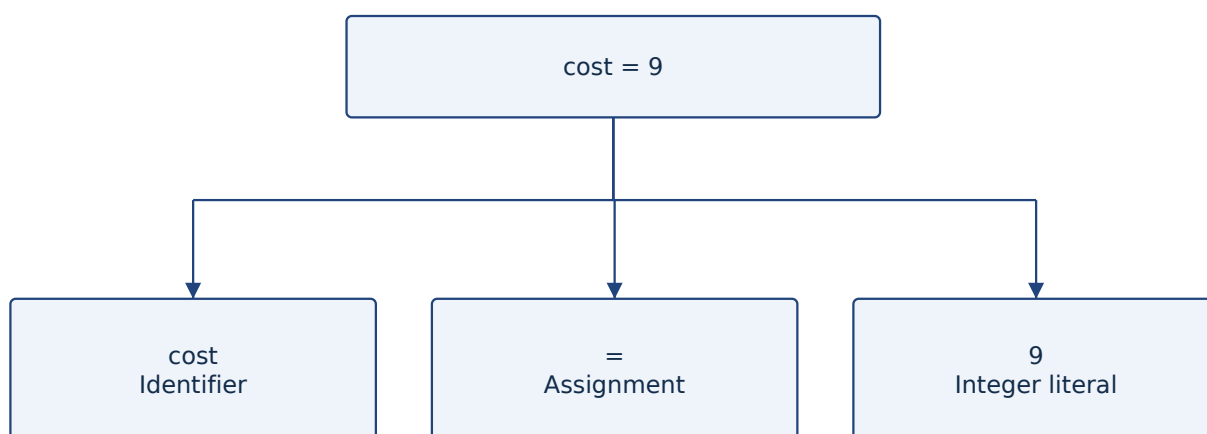


Figure 2.1. A statement is assembled from tokens. The branches classify the three parts; they are not alternative execution paths.

2.8 A first example

```
print("A token is a small building block.")
```

Output

```
A token is a small building block.
```

The call contains an identifier, parentheses and a string literal. The full sentence is one string value, not a series of variable names.

CHAPTER 2 / WORKED WALKTHROUGH

Read a line as a sequence of tokens

A name and a value are different things. In `total = price + 5`, `total` and `price` are identifiers; `5` is a literal; `=` and `+` are operators. The tokenizer does not decide what value `price` has. After identifying tokens, use the surrounding expression to explain what the program does.

Trace the program

```
price = 12
total = price + 5
print(total)
```

State trace

price = 12	Name price refers to 12
price + 5	Expression yields 17
total = ...	Name total refers to 17

A common wrong turn

`class` cannot be a variable name because it is a keyword; `Class` is a different, valid name. Avoid using `print` as a name because it hides the built-in.

Check your understanding

Classify each token in `total = price + 5`; then decide whether `total_2` and `2_total` are valid names.

2. Concept workshop: Source spelling and runtime value

A token is part of the source language, not the object itself. The spellings 10, 0b1010 and 0xA produce equal integer values. By contrast, "10" produces a string even though its characters look like a number. Separate three questions: Is the source legal? Which token category does it contain? What value and type result when it runs? An identifier also has two roles: its spelling must obey the grammar, and later it binds to an object. Changing a name from total_1 to total_2 does not change the arithmetic, but changing the literal from 10 to "10" can.

Execution model



Use the model before the code

- 2.1 Separate the assignment into chosen names, the assignment operator and the literal.
- 2.2 An identifier may contain digits after its first character.
- 2.3 Choose a quote delimiter that is absent from the text, or escape the embedded quote.
- 2.4 Evaluate each spelling as a number: decimal, binary and hexadecimal can denote the same integer.
- 2.5 The # character starts a comment only outside a quoted string.

Boundary habit

Classify each token in total = price + 5; then decide whether total_2 and 2_total are valid names.

Challenge the model

- 2.1 Change the number without changing the token categories. Distinguish = from ==.
- 2.3 Print the string and check that the backslash, if used only as an escape, is absent from output.
- 2.5 Predict the output of print("# note") and of a statement followed by # note.

2. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 2.1

Identify the tokens in a simple assignment.

Reasoning and algorithm

Separate the assignment into chosen names, the assignment operator and the literal.

Token classification describes source spelling; evaluating the statement is a separate step.

Program

```
cost = 9
print(cost)
```

Output

```
9
```

Why this result follows

cost and print are identifiers, 9 is a literal, = performs assignment, and the parentheses delimit a function call.

Boundary and error check

Change the number without changing the token categories. Distinguish = from ==.

Solved problem 2.2

Repair an identifier that begins with a digit.

Reasoning and algorithm

An identifier may contain digits after its first character. Replace the leading digit or move it after a letter; keep the name meaningful.

Program

```
score_2 = 18
print(score_2)
```

Output

```
18
```

Why this result follows

2score is invalid because it starts with a digit. The repaired identifier retains the intended meaning.

Boundary and error check

Confirm that the repaired name can be assigned and read, and that a reserved keyword still cannot be used.

Solved problem 2.3

Write a string containing an apostrophe.

Reasoning and algorithm

Choose a quote delimiter that is absent from the text, or escape the embedded quote. The apostrophe belongs to the string value.

Program

```
message = "It is Riya's notebook."  
print(message)
```

Output

```
It is Riya's notebook.
```

Why this result follows

Double quotes enclose the string, so the apostrophe inside does not terminate it.

Boundary and error check

Print the string and check that the backslash, if used only as an escape, is absent from output.

Solved problem 2.4

Compare three integer-literal notations.

Reasoning and algorithm

Evaluate each spelling as a number: decimal, binary and hexadecimal can denote the same integer. A prefix changes how source digits are interpreted.

Program

```
print(0b1100)  
print(0o14)  
print(0xC)
```

Output

```
12  
12  
12
```

Why this result follows

All three literals represent decimal 12. The prefix controls how the written digits are interpreted.

Boundary and error check

Compare values with `==` and types with `type`; do not mistake the spelling for the stored integer.

Solved problem 2.5

Distinguish a comment marker from a character in a string.

Reasoning and algorithm

The `#` character starts a comment only outside a quoted string. Inside quotes it is ordinary data.

Program

```
print("# is a character here") # This part is a comment.
```

Output

```
# is a character here
```

Why this result follows

The `#` inside the quotes is data. The later `#` starts a comment because it is outside the string.

Boundary and error check

Predict the output of `print("# note")` and of a statement followed by `# note`.

Trace the two hash characters

Position	Context	Python treatment
First #	Inside quoted string	Printed as data
Second #	Outside the string	Rest of line ignored
print call	One string argument	# is a character here

The lexer must find the closing quote before it can decide that the later `#` starts a comment.

2. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 2.1

Classify each token in the statement. Which names are chosen by the programmer?

```
result = length * 4
```

Exercise 2.2

Decide which identifier candidates are valid. Explain each decision.

```
net_total  
3total  
total-price  
_total  
while  
While
```

Exercise 2.3

Write a valid string containing both the word Python and an apostrophe. Display it.

Exercise 2.4

Predict the decimal values printed for 0b101, 0o10 and 0x10. Verify your answer.

Exercise 2.5

Explain why a triple-quoted string and a `#` comment are not the same kind of source-code element. Show a true two-line comment using `#`.

CHAPTER 3

3. Variables

A variable name lets you refer to a value and use it again. Follow names and values separately to understand assignment.

3.1 Binding a name

The assignment `points = 12` binds the name `points` to an integer object whose value is 12. Read `=` as "is assigned", not "is mathematically equal to". The expression on the right is evaluated first. A name must already exist before you read its value; `print(points)` before any assignment to `points` raises `NameError`. Use names that describe meaning rather than unexplained letters when the program grows.

3.2 Reassignment

`points = points + 3` reads the old value, calculates a new value and rebinds the name. If `points` was 12, it becomes 15. A name can later refer to a different type of object; Python does not require a permanent type declaration for the name. This flexibility does not mean arbitrary operations are valid. A string and an integer cannot be added directly with `+`.

3.3 Multiple assignment

`a, b = 4, 9` evaluates the right-hand values and assigns them to the corresponding targets. `a, b = b, a` swaps values because both old values are obtained first. `a = b = 0` binds both names to the same zero object. With immutable numbers this is harmless; with a mutable list, chained assignment can make both names share one list. Avoid repeating a target in the same assignment while learning; it makes tracing unnecessarily difficult.

3.4 Identity and mutability

Every object has a type, value and identity. `id(obj)` gives an identity integer that remains stable during that object's lifetime; do not memorise its numeric result or assume it is portable between runs. Immutable objects cannot change internally. Mutable objects, such as lists, can change. If two names refer to the same mutable object, a change through either name is visible through both. A new assignment to one name does not rebind the other name.

3.5 Tracing assignments

A trace table records the value of each relevant name after each statement. Start with an empty table and update only the target names. For `x = 2; y = x + 1; x = 8`, the final values are `x = 8` and `y = 3`. `y` does not continuously follow `x`; the expression was evaluated at the time of assignment. Informal terms `lvalue` and `rvalue` refer to the target and the value-producing side of assignment.

Visual walkthrough: Variables

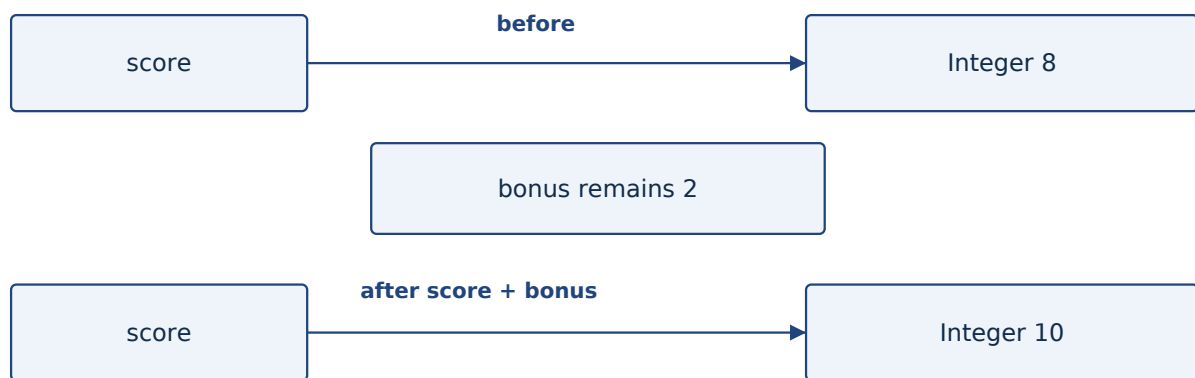


Figure 3.1. Reassignment changes the binding of `score` from 8 to 10. It does not change `bonus` or mutate the integer 8.

3.6 A first example

```
score = 8
bonus = 2
score = score + bonus
print(score)
```

Output

```
10
```

The expression `8 + 2` produces `10`. Only `score` is reassigned; `bonus` remains `2`.

Execution trace

Statement	Values after execution
<code>x = 4</code>	<code>x: 4; y not yet bound</code>
<code>y = x + 2</code>	<code>x: 4; y: 6</code>
<code>x = 10</code>	<code>x: 10; y: 6</code>

CHAPTER 3 / WORKED WALKTHROUGH

Assignment changes a binding

Draw an arrow from each name to its value. Assignment evaluates the right side first and then changes the left name. A second name does not automatically track later reassignments of the first. This model explains why a swap requires a saved value or simultaneous assignment.

Trace the program

```
a = 4
b = a
a = 9
print(a, b)
```

State trace

a = 4	a -> 4
b = a	a -> 4, b -> 4
a = 9	a -> 9, b -> 4

A common wrong turn

Reading an undefined name fails before an assignment can complete. Use a temporary name or `a, b = b, a` for a swap.

Check your understanding

Trace `a = 1; b = a; b = 3`. What will `print(a, b)` display?

3. Concept workshop: Names are arrows; objects hold values

Trace assignment by drawing a name on the left and an arrow to the object on the right. For immutable integers, `x = x + 1` computes a new integer and changes the arrow `x`; it does not edit the old integer. This is why another name previously bound to the old value stays there. A list is different: `append` modifies the object reached by the arrow, so every alias can observe the change. Rebinding a name to another list and mutating the original list are different operations. When tracing a program, write both the binding and the object state after every line.

Execution model



Use the model before the code

- 3.1 Draw one arrow per name.
- 3.2 A sequential swap that assigns the second name first can lose a value.
- 3.3 Trace each assignment at the moment it happens.
- 3.4 Find the first read of the undefined name.
- 3.5 Two names can point at one mutable list.

Boundary habit

Trace `a = 1; b = a; b = 3`. What will `print(a, b)` display?

Challenge the model

- 3.1 Test an initial value of zero and check whether the derived name changes when the source is later rebound.
- 3.3 Write a two-column state table after every line and predict the final print.
- 3.5 Contrast with `a.copy()`, which creates a different outer list. Use is to test identity.

3. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 3.1

Trace an update and a derived value.

Reasoning and algorithm

Draw one arrow per name. Evaluate the right-hand expression with the old binding, then rebound the left name; repeat for the derived value.

Program

```
x = 4
y = x + 2
x = 10
print(x, y)
```

Output

```
10 6
```

Why this result follows

y receives 6 when x is 4. Reassigning x later leaves y unchanged.

Boundary and error check

Test an initial value of zero and check whether the derived name changes when the source is later rebound.

Solved problem 3.2

Swap two ticket numbers.

Reasoning and algorithm

A sequential swap that assigns the second name first can lose a value. Preserve one old value in a temporary name or evaluate both right sides before rebinding.

Program

```
first, second = 21, 34
first, second = second, first
print(first, second)
```

Output

```
34 21
```

Why this result follows

Both right-hand values are evaluated before either left-hand target is rebound.

Boundary and error check

Test two unequal values and two equal values. Both methods should leave the pair reversed.

Solved problem 3.3

Compare a repeated assignment with a later update.

Reasoning and algorithm

Trace each assignment at the moment it happens. A later update to one name does not replay an earlier assignment to another.

Program

```
a = b = 7
a = 11
print(a, b)
```

Output

```
11 7
```

Why this result follows

Both names initially refer to 7. Rebinding a to 11 does not alter b.

Boundary and error check

Write a two-column state table after every line and predict the final print.

Solved problem 3.4

Repair use of an undefined name.

Reasoning and algorithm

Find the first read of the undefined name. Either initialise it before reading or use the name that was actually defined.

Program

```
count = 0
count = count + 1
print(count)
```

Output

```
1
```

Why this result follows

Initialise count before using it on the right of an assignment.

Boundary and error check

Run the repaired code with a simple value and verify there is no `NameError`.

Solved problem 3.5

Show shared-reference behaviour using a small list.

Reasoning and algorithm

Two names can point at one mutable list. Draw a single list box with two arrows; appending through either name changes that box.

Program

```
original = [3, 6]
alias = original
alias[0] = 9
print(original)
```

Output

```
[9, 6]
```

Why this result follows

The brackets create a mutable list. Index 0 selects its first item. Both names identify the same list; the list chapter develops these operations fully.

Boundary and error check

Contrast with `a.copy()`, which creates a different outer list. Use is to test identity.

Trace aliasing through an assignment

Step	original	alias
Create	[3, 6]	Unbound
alias = original	[3, 6]	Same list
alias[0] = 9	[9, 6]	Same list

The assignment copies a reference, not the list. A separately created copy would keep [3, 6].

3. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 3.1

Trace the following program. Record a and b after each statement.

```
a = 5
b = a * 2
a = b - 3
```

Exercise 3.2

Explain why $x = x + 1$ is valid after $x = 0$ but invalid when x has never been assigned.

Exercise 3.3

Write two different ways to swap a and b: one with a temporary name and one with simultaneous assignment.

Exercise 3.4

Predict the final values and explain why changing q does not change p.

```
p = q = 8
q = q + 2
```

Exercise 3.5

In your own words, distinguish assigning a new value to a name from mutating an existing object. Use a number and a list as examples.

CHAPTER 4

4. Data types

The type of a value determines which operations make sense. Begin with the type families, then study individual collections in later chapters.

4.1 Why type matters

The values `25` and `"25"` look similar when printed but have different types. One is an integer and the other is text. `25 + 5` gives `30`; `"25" + "5"` gives `"255"`. `type(value)` reports the type. A variable name is a reference to an object; the object has the type. Knowing the type helps you choose operations and interpret errors.

4.2 Numbers

`int` represents whole numbers of arbitrary precision subject to available memory. `float` represents real-number approximations, usually using binary floating point. Many decimal fractions are not represented exactly, so round only for display when appropriate. `complex` represents a real and an imaginary part; `3 + 2j` has real part `3` and imaginary part `2`. Python `3` has no separate built-in long integer type.

4.3 Booleans and absence

`bool` has two values, `True` and `False`, used for conditions. Booleans are a subclass of integers, so `True == 1` is true, but conditions should normally be written for clarity rather than arithmetic tricks. `None` is the single absence-of-value object, of type `NoneType`. It is different from zero, `False` and an empty string. Use `is None` when testing explicitly for absence.

4.4 Collection families

A string is a sequence of characters; a list is an ordered mutable collection; a tuple is an ordered immutable collection. A dictionary maps keys to values and a set keeps unique elements without sequence indexing. `[]` makes an empty list, `()` an empty tuple, `{}` an empty dictionary and `set()` an empty set. A collection may contain other collections; you will learn nested structures after the basics.

4.5 Mutable and immutable objects

Numbers, booleans, strings, tuples and `None` are immutable. Lists, dictionaries and sets are mutable. Immutable means that an existing object's contents cannot be changed, not that its name can never be reassigned. A tuple can contain a list whose contents change; the tuple's element reference remains the same. Avoid assumptions about memory layout: index `0` is a sequence position, not a literal memory address.

Visual walkthrough: Data types

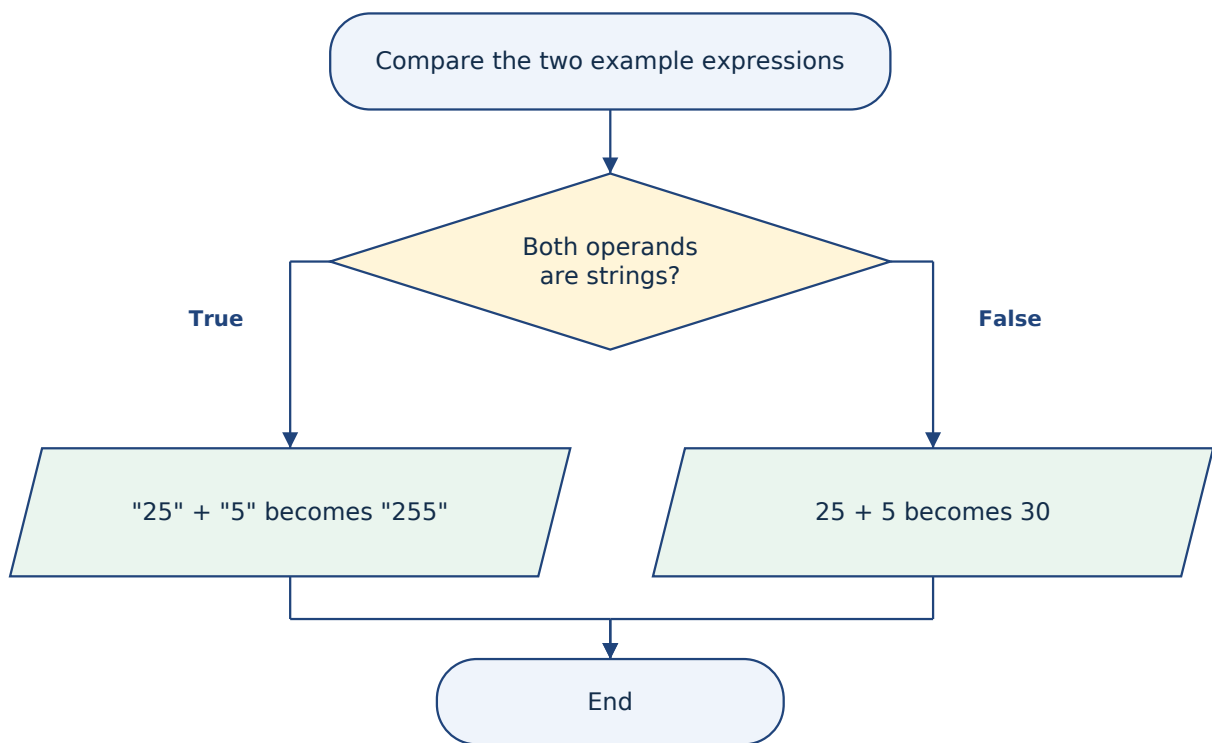


Figure 4.1. The operand types determine how + behaves in these two examples. This is a conceptual comparison, not a general type-checking program.

4.6 A first example

```
print(25 + 5)
print("25" + "5")
```

Output

```
30
255
```

The operator symbol is the same, but the types select different behaviour: arithmetic addition or text concatenation.

CHAPTER 4 / WORKED WALKTHROUGH

Choose a type from the operation

Ask what operations the value must support. A count is an integer, a measured temperature may be a float, text remains a string even if it contains digits, and a yes/no fact is a bool. `type` tells you the current type; it does not convert a value.

Trace the program

```
age = 16
label = "16"
print(age + 1)
print(label + "1")
```

State trace

age + 1	17: arithmetic
label + "1"	161: concatenation

A common wrong turn

`bool` is a subclass of `int` in Python, but use booleans for logical state rather than numeric counts. `None` means no value was supplied.

Check your understanding

Explain why `"8" + 2` fails and which conversion matches an arithmetic goal.

4. Concept workshop: Types describe available operations

Do not choose a type solely from how a value prints. A string of digits can be indexed and concatenated, while an integer supports arithmetic and cannot be indexed. A float is an approximation to a real measurement, so exact equality after calculations may be unsuitable. Bool represents a logical decision; None is often used as a deliberate absence marker. Collections introduce another dimension: whether their contents can change after creation. A tuple cannot replace its slots, yet it may hold a list whose contents can change. Ask which operations the problem requires, then select and test the representation.

Execution model



Use the model before the code

- 4.1 Source spelling determines whether 25 is an int or "25" is a str.
- 4.2 The / operator produces a float even for exactly divisible integers.
- 4.3 A complex literal has real and imaginary components.
- 4.4 None, False and 0 play different semantic roles: absence, a truth value and a quantity.
- 4.5 Each collection literal has its own type, and {} is an empty dictionary rather than a set.

Boundary habit

Explain why "8" + 2 fails and which conversion matches an arithmetic goal.

Challenge the model

- 4.1 Use type and a small operation to distinguish the pair; conversion does not happen just because text looks numeric.
- 4.3 Test a number with a zero imaginary component; it is still a complex object.
- 4.5 Inspect type for [], (), {}, and set(); test whether each can be modified.

4. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 4.1

Compare the types of two similar-looking values.

Reasoning and algorithm

Source spelling determines whether 25 is an int or "25" is a str. Inspect the values before selecting arithmetic or concatenation.

Program

```
print(type(25).__name__)  
print(type("25).__name__)
```

Output

```
int  
str
```

Why this result follows

type reports the type object; its `__name__` attribute displays a short readable name. The double underscores are part of the attribute spelling.

Boundary and error check

Use type and a small operation to distinguish the pair; conversion does not happen just because text looks numeric.

Solved problem 4.2

Predict the type of division and multiplication.

Reasoning and algorithm

The `/` operator produces a float even for exactly divisible integers. Multiplying integers usually produces an integer.

Program

```
print(type(8 / 2).__name__)  
print(type(8 * 2).__name__)
```

Output

```
float  
int
```

Why this result follows

Ordinary true division returns a float, even when the mathematical result is whole. Multiplying two integers gives an integer.

Boundary and error check

Predict values and types separately, including a non-divisible input.

Solved problem 4.3

Inspect a complex number.

Reasoning and algorithm

A complex literal has real and imaginary components. Access `.real` and `.imag` rather than trying to order complex values.

Program

```
z = 3 + 2j
print(z.real, z.imag)
```

Output

```
3.0 2.0
```

Why this result follows

The attributes `real` and `imag` return the two components as floats. `j` is Python's imaginary-unit notation.

Boundary and error check

Test a number with a zero imaginary component; it is still a complex object.

Solved problem 4.4

Distinguish `None` from `false` and `zero`.

Reasoning and algorithm

`None`, `False` and `0` play different semantic roles: absence, a truth value and a quantity. Equality can blur distinctions, so use `is None` for the absence check.

Program

```
print(None == 0)
print(None == False)
print(None is None)
```

Output

```
False
False
True
```

Why this result follows

None represents absence rather than an ordinary zero-like value. The identity test is the standard explicit absence check.

Boundary and error check

Print type for each. Avoid using if not value when zero is valid data but None is not.

Solved problem 4.5

Identify empty collection types.

Reasoning and algorithm

Each collection literal has its own type, and {} is an empty dictionary rather than a set. An empty set requires set().

Program

```
print(type([]).__name__)
print(type(()).__name__)
print(type({}).__name__)
print(type(set()).__name__)
```

Output

```
list
tuple
dict
set
```

Why this result follows

Empty braces denote a dictionary, so an empty set needs the set constructor.

Boundary and error check

Inspect type for [], (), {}, and set(); test whether each can be modified.

Choose the empty collection

Expression	Type	Mutable?
[]	list	Yes
()	tuple	No
{}	dict	Yes
set()	set	Yes

The spelling {} already belongs to dictionaries, so an empty set needs a constructor.

4. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 4.1

Classify 4, 4.0, 4j, "4", True and None by type. Explain two cases that might look similar in output.

Exercise 4.2

Predict the results and types of $9 + 2.0$, $9 // 2$ and $9 / 2$.

Exercise 4.3

Choose suitable types for a name, a count of students, a temperature, a pass/fail flag and an optional missing mark.

Exercise 4.4

Explain whether a list, string and tuple can have their first element replaced in place.

Exercise 4.5

Create one example each of a list, tuple, dictionary and set. Explain how you would identify an element in each collection.

CHAPTER 5

5. Type conversion

Convert data deliberately when it arrives in the wrong representation for the operation you need.

5.1 Conversion creates a result

A conversion call such as `int("18")` returns a new value of the requested type when that conversion is supported. It does not automatically change the original variable. `raw = "18"; number = int(raw)` leaves `raw` as text and gives `number` the integer 18. Conversions are not arbitrary reinterpretations of memory; they follow defined rules and may reject a value.

5.2 Integer and floating conversion

`int` accepts an integer-shaped string such as `"-12"` and can truncate a finite float towards zero. Thus `int(-4.8)` is `-4`. A decimal-point string such as `"4.8"` is not directly accepted by `int`. `float` accepts strings such as `"4.8"` and `"2e3"`. `int(float(text))` is only appropriate if deliberately discarding a fractional part is part of your specification, not a general repair for every failed conversion.

5.3 Implicit numeric conversion

Python can combine compatible built-in numeric types, for example `3 + 0.5` gives `3.5`. The integer participates in floating-point arithmetic. Python does not implicitly convert text digits into numbers for addition: `"3" + 0.5` raises `TypeError`. The distinction protects against guessing whether a string should be treated as text or as a quantity.

5.4 Text and Boolean conversion

`str(value)` gives a readable string representation. This is useful for concatenation, although `print` can accept several values without manual conversion. `bool(value)` tests truthiness, not the spelling of the text: `bool("False")` is `True` because the string is nonempty. To accept a yes/no response, compare a normalised string with a recognised response and reject unexpected text when necessary.

5.5 Conversion is not validation

A successful numeric conversion does not establish that an age is non-negative or a mark lies between `0` and `100`. Validation checks the problem's rules after conversion. `ValueError` commonly reports malformed numeric text; `TypeError` can indicate an unsupported input type. Do not use `eval` to read a number or collection from an unknown user. The input chapter shows explicit conversion instead.

Visual walkthrough: Type conversion

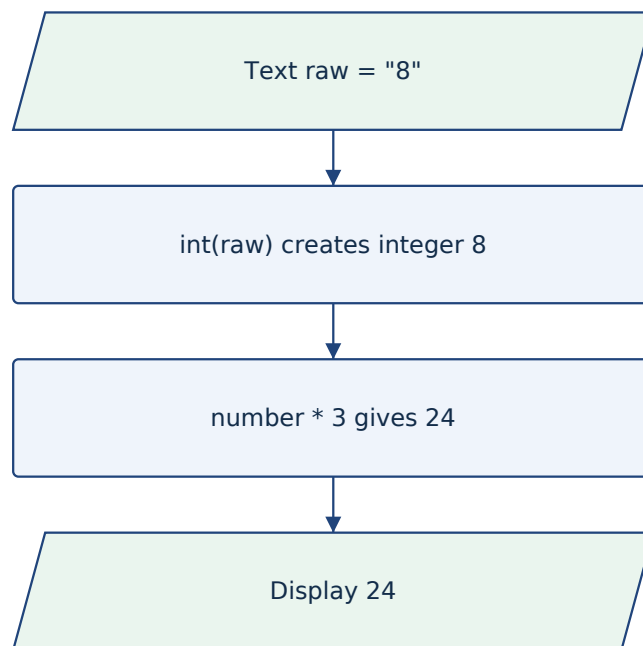


Figure 5.1. Conversion happens before multiplication. Follow the value and its type together.

5.6 A first example

```
raw = "8"  
number = int(raw)  
print(number * 3)
```

Output

```
24
```

The conversion changes the operation from repeating text to multiplying a quantity.

Concept bridge: Conversions are explicit transformations

Converting a value makes a new value of another type.

5. Concept workshop: Conversions are explicit transformations

Converting a value makes a new value of another type. It does not automatically rewrite the source name. `int("12")` interprets digit text as an integer, while `str(12)` prepares an integer for display or concatenation. `int(2.9)` discards the fractional part; `round(2.9)` answers a different question. `bool` examines truthiness, not the English meaning of a string. Keep conversion near the boundary where data enters or leaves a program. That placement makes it easy to test bad input and to see whether arithmetic is being done on numbers or text.

Execution model



Use the model before the code

- 5.1 input-like text values concatenate.
- 5.2 `int` of a finite float truncates towards zero, not down toward negative infinity.
- 5.3 Conversion returns a new value; it does not rewrite the original string binding.
- 5.4 `bool` of a nonempty string is `True`, including the string `"False"`.
- 5.5 Build display text by explicitly converting a number or by using a formatted string.

Boundary habit

Try `"0"`, `"-2"`, and `"eight"`. Which inputs meet the stated numeric contract?

Challenge the model

- 5.1 Test `"0"`, a negative spelling, and a nonnumeric spelling. The last raises `ValueError`.
- 5.3 Print the original and converted types after the operation.
- 5.5 Check a zero count and a multi-digit count; preserve a space between words and number.

5. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 5.1

Convert two text counts and add them.

Reasoning and algorithm

input-like text values concatenate. Convert each to int before adding; conversion belongs at the boundary between text and arithmetic.

Program

```
a, b = "14", "6"  
print(int(a) + int(b))
```

Output

```
20
```

Why this result follows

Explicit integer conversion turns two text values into numbers before addition. Convert at the boundary Input arrives as text. Convert it when you need arithmetic, and keep the original if you still need its exact spelling. `int("8")` succeeds; `int("8.0")` raises

Boundary and error check

Test "0", a negative spelling, and a nonnumeric spelling. The last raises `ValueError`.

Solved problem 5.2

Demonstrate truncation towards zero.

Reasoning and algorithm

`int` of a finite float truncates towards zero, not down toward negative infinity. Compare a positive and negative fractional value.

Program

```
print(int(4.9), int(-4.9))
```

Output

```
4 -4
```

Why this result follows

The fractional part is discarded. This differs from `floor` for negative numbers.

Boundary and error check

Check `int(-4.8)` against `math.floor(-4.8)` to expose the different rule.

Solved problem 5.3

Show that conversion does not mutate the source name.

Reasoning and algorithm

Conversion returns a new value; it does not rewrite the original string binding. Keep two names if both forms are needed.

Program

```
raw = "12.5"
number = float(raw)
print(type(raw).__name__, type(number).__name__)
```

Output

```
str float
```

Why this result follows

raw remains a string. number holds the conversion result.

Boundary and error check

Print the original and converted types after the operation.

Solved problem 5.4

Explain a surprising Boolean conversion.

Reasoning and algorithm

bool of a nonempty string is True, including the string "False". It tests whether the container is empty, not whether the word describes truth.

Program

```
print(bool("False"))
print(bool(""))
print(bool(0))
```

Output

```
True
False
False
```

Why this result follows

Nonempty text is true regardless of its meaning. Empty text and numeric zero are false.

Boundary and error check

Compare bool(""), bool("0"), bool(0), and bool("False").

Solved problem 5.5

Create a message containing a numeric count.

Reasoning and algorithm

Build display text by explicitly converting a number or by using a formatted string. Addition between str and int is not automatic.

Program

```
count = 6
message = "Count: " + str(count)
print(message)
```

Output

```
Count: 6
```

Why this result follows

Both operands of concatenation are strings after conversion.

Boundary and error check

Check a zero count and a multi-digit count; preserve a space between words and number.

Trace value and type

Step	Expression	Value / type
1	count = 6	6 / int
2	str(count)	"6" / str
3	"Count: " + "6"	"Count: 6" / str
4	print(message)	Count: 6

Without str(count), the + operation receives str and int and raises TypeError before print runs.

5. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 5.1

Predict each result and state its type. Evaluate these expressions independently.

```
int(7.99)
int(-7.99)
float("7")
str(7.0)
```

Exercise 5.2

Explain why `int("8.2")` fails. Give one correct way to retain the fraction and one way to deliberately discard it.

Exercise 5.3

A user types "no". Explain why `bool("no")` is unsuitable for interpreting their answer.

Exercise 5.4

Convert "015" to an integer and back to a string. Explain why the leading zeroes disappear.

Exercise 5.5

A height arrives as the string "1.72" in metres. Convert it, calculate centimetres and state one validation rule.

CHAPTER 6

6. Input and output

Read information from a user and present results clearly. Input and output are the boundary between your program and the person using it.

6.1 Reading a line

`input(prompt)` displays an optional prompt, waits for a line of input and returns that line as a string without its trailing newline. It does not automatically interpret a number. Use `age = int(input("Age: "))` only when the input is expected to be an integer. During this book's solved examples, fixed strings represent typed responses so that the code and output are reproducible.

6.2 Collect and convert separately

For debugging, split input and conversion into two statements: `raw = input("Price: "); price = float(raw)`. You can then inspect `raw` when conversion fails. A phone number or postal code is often best kept as text, because leading zeroes and formatting are significant and arithmetic is not needed. A numeric-looking value is not automatically a quantity.

6.3 Printing values

`print` accepts zero or more values. With several arguments, it inserts `sep` between them; the default is one space. After the values, it writes `end`; the default is a newline. `print()` by itself produces a blank line. `print("x")` displays the letter `x`, while `print(x)` looks up the variable `x`. `print` returns `None`; do not use its return value as the text you wanted to display.

6.4 Formatted strings

An f-string begins with `f` before the quote and puts expressions inside braces. `f"{total:.2f}"` displays two digits after the decimal point. It controls appearance, not the stored value. Include meaningful units and labels so the output is understandable. A displayed `3.00` is still text; formatting a value is different from converting the underlying calculation into exact decimal arithmetic.

6.5 Escapes and clean presentation

The escape sequence `\n` inserts a newline, `\t` a tab and `\\` a backslash. A tab's visual width depends on the viewer; fixed tables are often clearer with carefully formatted text. Quotes inside a string must not accidentally end it. In prompts, state the expected unit and type. Test output for spacing as well as numbers, especially when `sep` or `end` has changed.

Visual walkthrough: Input and output

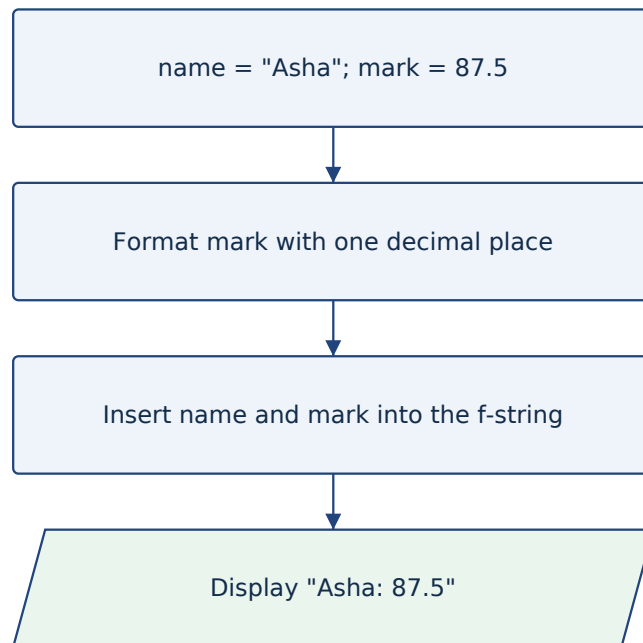


Figure 6.1. Output formatting constructs display text; it does not alter the stored mark.

6.6 A first example

```

name = "Asha"
mark = 87.5
print(f"{name}: {mark:.1f}")
  
```

Output

```
Asha: 87.5
```

The f-string inserts a name and a formatted number into one piece of text. `.1f` requests one digit after the decimal point.

Quick reference

Operation	Purpose / example
<code>input(prompt)</code>	Read one line of text.
<code>int(input(...))</code>	Read and convert an integer response.
<code>print(a, b)</code>	Display several values.
<code>sep=" "</code>	Choose the separator between values.
<code>end=" "</code>	Choose what follows this print call.
<code>f"{value:.2f}"</code>	Format a numeric value to two decimal places.
<code>\n / \t / \\\</code>	Newline / tab / literal backslash.

CHAPTER 6 / WORKED WALKTHROUGH

Treat input and display as a boundary

Separate three steps: read text, convert to the needed type, and format the result. This keeps presentation choices such as precision, separators and prompts apart from the calculation. `input` always returns a string even when the user types digits.

Trace the program

```
age_text = input("Age: ")
age = int(age_text)
print(f"Next year: {age + 1}")
```

State trace

User enters 16	age_text = "16"
Conversion	age = 16
Display	Next year: 17

A common wrong turn

Use `print(a, b, sep=" | ")` to choose a separator and `end=""` to suppress its final newline. f-string precision such as `{area:.2f}` changes the display, not the stored float.

Check your understanding

Try zero and a nonnumeric age. Explain precisely which operation fails in the second case.

6. Concept workshop: A clean input-to-output pipeline

Divide an interactive task into input, validation or conversion, calculation, and formatting. For a radius, read text, convert to float, check the documented non-negative assumption, compute area, then format the display. This order identifies exactly where invalid text fails and prevents display precision from contaminating the calculation. `print` has two orthogonal layout controls: `sep` joins several arguments inside one call, and `end` determines what follows the complete call. An f-string formats one expression within a larger message. Write the desired transcript, including prompts and spaces, before coding the display.

Execution model



Use the model before the code

- 6.1 input returns text.
- 6.2 Multiply numeric quantity and price first.
- 6.3 Use the same two digit strings twice: `+` concatenates before conversion and adds after int conversion.
- 6.4 Separate input, int conversion and output formatting.
- 6.5 The first print sets end to a space rather than a newline.

Boundary habit

Try zero and a nonnumeric age. Explain precisely which operation fails in the second case.

Challenge the model

- 6.1 For input 0 expect 1. For "sixteen" identify the conversion line that raises `ValueError`.
- 6.3 Check that the operands are still strings after the second print unless rebound explicitly.
- 6.5 Test a one-digit day and month. Count spaces and hyphens in the exact output.

6. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 6.1

Read an age and display next year's age. Sample response: 16.

Reasoning and algorithm

input returns text. Read the age, convert once, then add one before displaying; the sample response is typed by the user, not produced by the program.

Program

```
raw_age = input("Age: ")
age = int(raw_age)
print(age + 1)
```

Output

```
Age: 16
17
```

Why this result follows

The displayed transcript includes the sample response 16. In your own run, type your age when the prompt appears. input returns text; int converts it before arithmetic.

Boundary and error check

For input 0 expect 1. For "sixteen" identify the conversion line that raises ValueError.

Solved problem 6.2

Display an invoice total with two decimal places.

Reasoning and algorithm

Multiply numeric quantity and price first. Format the displayed total to two decimals without assuming that formatting changes the stored value.

Program

```
quantity = 3
price = 19.95
total = quantity * price
print(f"Total: {total:.2f}")
```

Output

```
Total: 59.85
```

Why this result follows

The format specifier `.2f` controls the presentation of a floating-point result.

Boundary and error check

Try zero quantity and a price with more than two fractional digits.

Solved problem 6.3

Compare addition before and after conversion.

Reasoning and algorithm

Use the same two digit strings twice: `+` concatenates before conversion and adds after int conversion. Predict both outcomes before running.

Program

```
a, b = "12", "3"
print(a + b)
print(int(a) + int(b))
```

Output

```
123
15
```

Why this result follows

The first operation joins strings. The second adds integers. Identical-looking digits can participate in different operations depending on their type.

Boundary and error check

Check that the operands are still strings after the second print unless rebound explicitly.

Solved problem 6.4

Read an integer mark and print a labelled result using one decimal place.

Reasoning and algorithm

Separate input, int conversion and output formatting. A one-decimal format works on the converted numeric mark.

Program

```
raw = input("Mark: ")
mark = int(raw)
print(f"Mark: {mark:.1f}")
```

Output

```
Mark: 87
Mark: 87.0
```

Boundary and error check

For input 87 expect 87.0 in the label; nonnumeric input fails at int, not print.

Solved problem 6.5

Print a date with a custom separator and a label without an extra newline.

Reasoning and algorithm

The first print sets end to a space rather than a newline. The second print inserts sep only between its three arguments.

Program

```
day, month, year = 29, 9, 2026
print("Date:", end=" ")
print(day, month, year, sep="-")
```

Output

```
Date: 29-9-2026
```

Why this result follows

end controls what follows the first call; sep controls what goes between the arguments in the second call.

Boundary and error check

Test a one-digit day and month. Count spaces and hyphens in the exact output.

Trace the print boundaries

Call	Written characters	What follows
First	Date:	One space from end
Second	29-9-2026	One newline
Combined	Date: 29-9-2026	One output line

sep is placed only between day, month and year; end belongs only to the first call.

6. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 6.1

Read a name and an integer age, then display a complete sentence giving the age in five years. State what happens for input "sixteen".

Exercise 6.2

Read radius as a float and display the circle area using 3.14159, rounded to two decimal places. Assume radius is non-negative.

Exercise 6.3

Read a price and quantity, then print the total to two decimal places. Test quantity zero.

Exercise 6.4

Read a first and last name and display them on one line separated by one space.

Exercise 6.5

Write a program that displays three subject marks separated by " | " and their average to one decimal place.

CHAPTER 7

7. Operators

Evaluate expressions accurately, select the correct operator and distinguish value equality from object identity.

7.1 Arithmetic operators

Use `+`, `-`, `*`, `/`, `//`, `%` and `**` for addition, subtraction, multiplication, true division, floor division, remainder and exponentiation. `/` produces a float for ordinary numeric operands. `//` rounds the quotient down towards negative infinity: `-7 // 3` is `-3`. The identity `a == (a // b) * b + a % b` holds for integers when `b` is nonzero. Python's remainder has the divisor's sign, or is zero.

7.2 Comparison and Boolean logic

Comparison operators `==`, `!=`, `<`, `<=`, `>` and `>=` compare values. A chained comparison such as `0 <= score <= 100` expresses an inclusive range. `and` requires both conditions, or requires at least one, and `not` negates truth. `and` and `or` short-circuit: an unnecessary second operand is not evaluated. They return an operand, which is not always a bool. Zero, empty collections, empty strings, `None` and `False` are falsey; most other values are truthy.

7.3 Membership, identity and assignment

`in` and `not in` test membership; for a dictionary they test keys. `is` and `is not` test whether two references identify the same object. Use `==` for ordinary value comparisons and `is None` for absence. `=` assigns, whereas `==` compares. Augmented assignment such as `total += amount` combines an operation with assignment. With mutable objects, some augmented assignments can modify the existing object.

7.4 Bitwise operations

Bitwise `&`, `|` and `^` operate on integer bits: AND retains common 1 bits, OR retains any 1 bit, and XOR retains differing bits. `<<` shifts left; `>>` shifts right. For non-negative integers, shifting left by `n` multiplies by `2**n` and shifting right by `n` performs floor division by `2**n`. `~x` equals `-x - 1`. Bitwise operators do not replace `and/or` in ordinary condition expressions.

7.5 Choosing the correct operator

Distinguish `=` from `==`, `==` from `is`, and logical `and/or` from bitwise `&/|`. Membership tests a substring in a string, an element in a sequence and a key in a dictionary. Unary operators act on one operand; binary operators use two. Read `5 - -2` as subtracting a negative number. Augmented assignment such as `x += 2` updates a target; the symbol must be written together without a space between `+` and `=`. Expression grouping is developed separately in the next chapter.

7.6 Bitwise tracing and Boolean operands

For a bitwise calculation, align the binary digits and apply the rule column by column. 10 is 1010 and 6 is 0110, so their AND is 0010. XOR is 1 only where the two bits differ. Python integers do not overflow at a fixed eight-bit boundary. For logical operators, 3 and 8 returns 8, while 0 and 8 returns 0; 0 or 8 returns 8. not always produces a Boolean. This operand-return behaviour explains many output-prediction questions.

Visual walkthrough: Operators

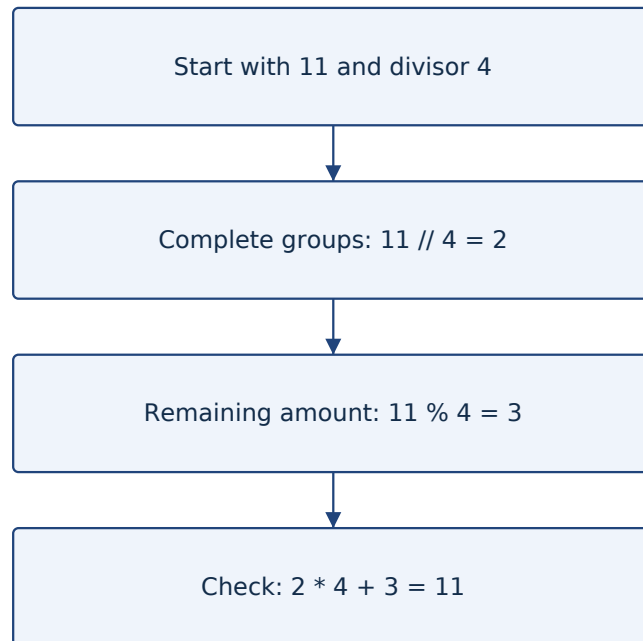


Figure 7.1. Floor division and remainder describe the same grouping in complementary ways.

7.7 A first example

```
print(11 // 4, 11 % 4)
print(3 < 8 and 8 < 10)
```

Output

```
2 3
True
```

The first line splits 11 into complete groups of four and a remainder. The second combines two comparisons.

Quick reference

Family	Operators
Arithmetic	+ - * // % **
Comparison	== != < <= > >=
Logical	and or not
Bitwise	& ^ ~ << >>
Membership	in, not in
Identity	is, is not
Assignment	= and augmented forms such as +=

Concept bridge: Operator results and evaluation order

An expression has a value and an evaluation path. Division and remainder are linked by the quotient identity, including negative operands. Boolean and/or may omit the right operand, which can safely guard a risky division. Bitwise operators instead work on integer bits and do not have the same short-circuit behavior. Equality asks whether values compare equal; identity asks whether they are the same object. Parentheses should record the grouping a reader needs, even when precedence rules would happen to produce the same result.

7. Concept workshop: Operator results and evaluation order

An expression has a value and an evaluation path. Division and remainder are linked by the quotient identity, including negative operands. Boolean and/or may omit the right operand, which can safely guard a risky division. Bitwise operators instead work on integer bits and do not have the same short-circuit behavior. Equality asks whether values compare equal; identity asks whether they are the same object. Parentheses should record the grouping a reader needs, even when precedence rules would happen to produce the same result.

Execution model



Use the model before the code

- 7.1 Apply the discount to the original amount, then compute tax on the discounted subtotal.
- 7.2 Floor division moves toward negative infinity.
- 7.3 A Boolean and short-circuits.
- 7.4 Equality compares list values, while is compares object identity.
- 7.5 Write 10 and 6 in aligned binary columns, apply &, | and ^ bit by bit, then convert back to decimal.

Boundary habit

Compute $7 // 3$ and $-7 // 3$ by hand, then verify both remainder identities.

Challenge the model

- 7.1 Test a zero bill and verify that swapping tax and discount only happens to commute for simple proportional rates.
- 7.3 Test denominator zero and nonzero; ensure the zero case does not raise.
- 7.5 Confirm that a bit present in both operands appears in &, whereas a bit present in exactly one appears in ^.

7. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 7.1

Evaluate a bill after a discount and tax.

Reasoning and algorithm

Apply the discount to the original amount, then compute tax on the discounted subtotal. Named intermediate values make the order auditable.

Program

```
subtotal = 800
discounted = subtotal * (1 - 0.10)
final = discounted * (1 + 0.05)
print(f"{final:.2f}")
```

Output

```
756.00
```

Why this result follows

Percentage changes apply to the current base. Tax is calculated on the discounted amount, as specified.

Boundary and error check

Test a zero bill and verify that swapping tax and discount only happens to commute for simple proportional rates.

Solved problem 7.2

Investigate negative floor division and remainder.

Reasoning and algorithm

Floor division moves toward negative infinity. Compute quotient first, then choose the remainder so divisor times quotient plus remainder reconstructs the dividend.

Program

```
a, b = -7, 3
q, r = a // b, a % b
print(q, r)
print(q * b + r)
```

Output

```
-3 2
-7
```

Why this result follows

Flooring -2.333... gives -3. The remainder 2 restores the original dividend: $-3 \times 3 + 2 = -7$. Evaluate operator families deliberately. Parentheses make the intended grouping visible. First evaluate arithmetic, then comparisons, then Boolean expressions; short-c

Boundary and error check

Test -7 and 3, then 7 and -3. Do not substitute truncation toward zero.

Solved problem 7.3

Check a score and guard against division by zero.

Reasoning and algorithm

A Boolean and short-circuits. Put the denominator guard before division so the risky expression is evaluated only when safe.

Program

```
score = 87
print(0 <= score <= 100)
denominator = 0
print(denominator != 0 and 10 / denominator > 2)
```

Output

```
True
False
```

Why this result follows

The first condition is an inclusive range. Short-circuiting stops before division when the denominator is zero.

Boundary and error check

Test denominator zero and nonzero; ensure the zero case does not raise.

Solved problem 7.4

Compare equal lists and shared references.

Reasoning and algorithm

Equality compares list values, while is compares object identity. Construct equal but distinct lists to separate the two questions.

Program

```
a = [1, 2]
b = [1, 2]
c = a
print(a == b, a is b, a is c)
```

Output

```
True False True
```

Why this result follows

a and b contain equal values but are distinct objects. c refers to the exact object that a refers to.

Boundary and error check

Mutate one list and check whether the other follows; that reveals an alias.

Solved problem 7.5

Calculate bitwise operations on 10 and 6.

Reasoning and algorithm

Write 10 and 6 in aligned binary columns, apply &, | and ^ bit by bit, then convert back to decimal.

Program

```
a, b = 10, 6
print(a & b, a | b, a ^ b)
print(a << 1, a >> 1, ~a)
```

Output

```
2 14 12
20 5 -11
```

Why this result follows

In four bits, 10 is 1010 and 6 is 0110. The shared 1 bit gives 0010 (2); the union gives 1110 (14).

Boundary and error check

Confirm that a bit present in both operands appears in &, whereas a bit present in exactly one appears in ^.

Align bits before combining

Operation	Binary result	Decimal
1010 & 0110	0010	2
1010 0110	1110	14
1010 ^ 0110	1100	12
1010 << 1	10100	20
~10	-(10 + 1)	-11

The complement uses Python integer semantics, not a fixed four-bit width.

7. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 7.1

Predict and then check: $18 / 4$, $18 // 4$, $18 \% 4$, $-18 // 4$ and $-18 \% 4$. Explain the negative case.

Exercise 7.2

Evaluate $2 ** 3 ** 2$, $(2 ** 3) ** 2$, $-3 ** 2$ and $(-3) ** 2$. Explain the grouping.

Exercise 7.3

Write a Boolean expression for a temperature outside the inclusive range 18 to 30, with parentheses for clarity.

Exercise 7.4

Given $a = [4, 5]$, $b = a$ and $c = a.copy()$, predict all equality and identity comparisons, then append 6 to b and reassess.

Exercise 7.5

Use bitwise operators to calculate $12 \& 5$, $12 | 5$, $12 \wedge 5$ and $12 >> 2$. Show four-bit working.

CHAPTER 8

8. Expressions

An expression produces a value. Learn to translate formulas and evaluate a compound expression without guessing.

8.1 Expressions and statements

A literal, a variable reference, a function call or a combination of operators can be an expression. `3 + 4` produces 7. An assignment statement such as `total = 3 + 4` uses that result to bind a name. Expressions also appear inside print calls and conditions; they are not restricted to the right side of assignments. In a script, an unused expression usually produces no visible output.

8.2 Precedence

Precedence determines grouping when several operators appear without parentheses. `**` binds more tightly than multiplication and division, which bind more tightly than addition and subtraction. Thus `4 + 3 * 2` is 10. Parentheses override the normal grouping: `(4 + 3) * 2` is 14. Write extra parentheses when they make the intended calculation easier for a reader to verify.

8.3 Associativity

At most arithmetic precedence levels, operators group left to right. `20 / 5 * 2` is `(20 / 5) * 2`, or 8.0. Exponentiation groups right to left: `2 ** 3 ** 2` means `2 ** (3 ** 2)`, or 512. Unary minus interacts with exponentiation: `-3 ** 2` is -9, while `(-3) ** 2` is 9. Associativity does not mean arbitrary reordering is safe for floating-point calculations.

8.4 Boolean expressions

Comparisons bind more tightly than not, then and, then or. Parenthesise mixed conditions when you are unsure. Python short-circuits and/or, so some subexpressions are not evaluated at all. The expression `denominator != 0` and `total / denominator > 5` avoids dividing by zero. A chain such as `1 <= x <= 10` checks both boundaries, with the middle value evaluated once.

8.5 Translate mathematics explicitly

Python needs `*` for multiplication: `2x` must be written `2 * x`. Use `**` for powers; `^` is bitwise XOR. Group a whole numerator and denominator: `(a + b) / (c + d)`. To compute a percentage, state the total possible marks. To apply a discount followed by tax, apply each rate to the specified base rather than merely adding the percentages.

8.6 Trace subexpressions

Substitute known values, mark parentheses, evaluate tight operations and then combine the results. Record intermediate values for difficult calculations. If a formula is long, use meaningful temporary names rather than one dense expression. Check zero denominators and negative square-root inputs separately. A correct operator order cannot rescue an incorrect formula.

8.7 Rates, dimensions and formula checks

Use $\text{simple interest} = \text{principal} * \text{rate} * \text{time} / 100$ when rate is a percentage per matching time unit. Profit or loss can be represented by $\text{selling_price} - \text{cost_price}$, with the sign interpreted afterwards. Convert compound lengths by first moving into one unit: $\text{total_inches} = \text{feet} * 12 + \text{inches}$, then $\text{centimetres} = \text{total_inches} * 2.54$. These examples illustrate dimensional reasoning: choose consistent units before calculating. A formula with the right syntax can still use the wrong denominator or rate base.

Visual walkthrough: Expressions

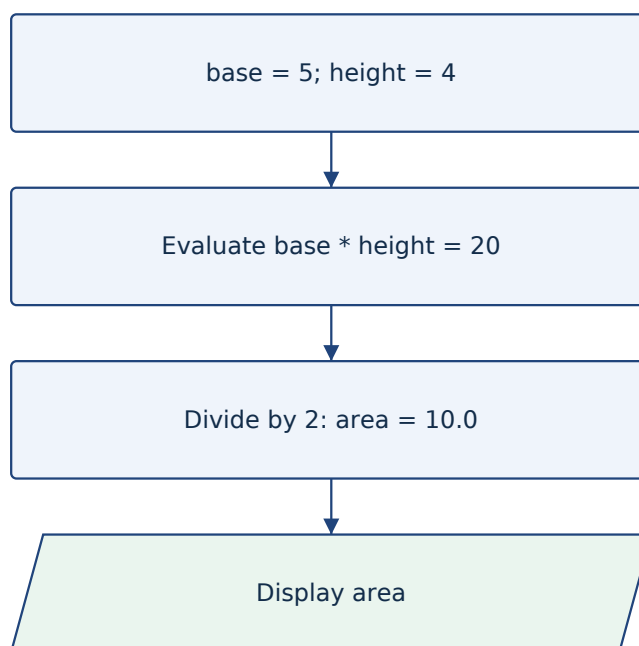


Figure 8.1. The parenthesised product is evaluated before division in the first example.

8.8 A first example

```
base = 5
height = 4
area = (base * height) / 2
print(area)
```

Output

```
10.0
```

Evaluation produces a value, then assignment binds `area` to that value. The final `print` is a separate statement.

Quick reference

Group, high to low	Operators
Grouping / access	(...), indexing, calls
Power then unary	**; unary +, -, ~ (see power caveat)
Multiplicative / additive	* / // %; + -
Shifts / bitwise	<< >>; &; ^;
Comparison / membership / identity	< <= > >= == !=; in, not in; is, is not
Boolean	not; and; or

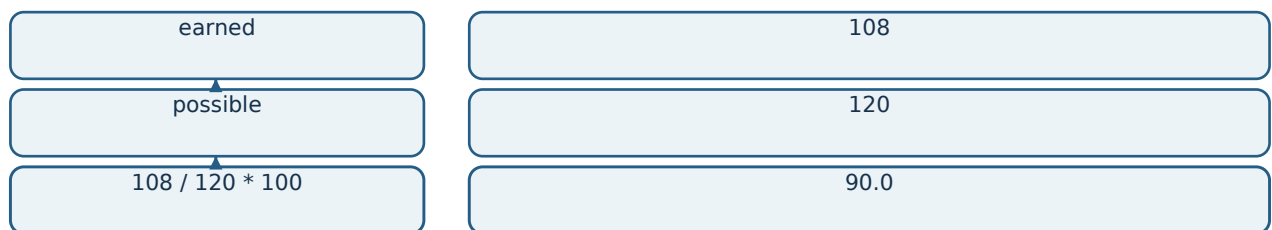
Concept bridge: Translate mathematics in stages

A formula on paper often uses a long horizontal fraction bar or implicit multiplication. Python does not see that visual grouping. Name the numerator and denominator or put each inside parentheses. Then test with numbers for which the intended answer is easy to calculate by hand. For percentages, distinguish an average of percentages from a percentage of aggregate marks: different denominators mean the questions are different.

8. Concept workshop: Translate mathematics in stages

A formula on paper often uses a long horizontal fraction bar or implicit multiplication. Python does not see that visual grouping. Name the numerator and denominator or put each inside parentheses. Then test with numbers for which the intended answer is easy to calculate by hand. For percentages, distinguish an average of percentages from a percentage of aggregate marks: different denominators mean the questions are different. A discount followed by a tax has two sequential bases. Intermediate variables are an audit trail, not wasted lines.

Execution model



Use the model before the code

- 8.1 Write the grouping before calculating.
- 8.2 Exponentiation associates right to left.
- 8.3 Translate the numerator and denominator as separate expressions.
- 8.4 Place the nonzero test to the left of and.
- 8.5 Different subject maxima require a weighted total: add earned marks, add possible marks, then divide and multiply by 100.

Boundary habit

For a 15% discount followed by 8% tax, calculate discounted price before tax; test an original price of 1200.

Challenge the model

- 8.1 Evaluate each subexpression independently and compare the two results.
- 8.3 Substitute simple numbers and calculate by hand; state the denominator must be nonzero.
- 8.5 Check an all-zero earned score and a perfect score. Averaging raw marks is not the same calculation.

8. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 8.1

Evaluate grouping with and without parentheses.

Reasoning and algorithm

Write the grouping before calculating. Parentheses can force subtraction before division; without them division binds more tightly.

Program

```
print(4 + 3 * 2)
print((4 + 3) * 2)
```

Output

```
10
14
```

Why this result follows

Multiplication first gives $4 + 6$. Parentheses first give $7 * 2$. Use intermediate values for expressions. A long expression is easier to verify if you name each meaningful subtotal. Percentage questions often require dividing the sum earned by the sum possible.

Boundary and error check

Evaluate each subexpression independently and compare the two results.

Solved problem 8.2

Explain exponentiation associativity.

Reasoning and algorithm

Exponentiation associates right to left. Compute the inner power first, then the outer one; do not evaluate strictly from left to right.

Program

```
print(2 ** 3 ** 2)
print((2 ** 3) ** 2)
```

Output

```
512
64
```

Why this result follows

The first means $2 ** 9$. The second explicitly squares 8.

Boundary and error check

Add explicit parentheses for both possible groupings and compare.

Solved problem 8.3

Translate a fraction formula.

Reasoning and algorithm

Translate the numerator and denominator as separate expressions. Parenthesise each so Python performs the intended fraction rather than a partial division.

Program

```
a, b, c, d = 6, 4, 3, 2
result = (a + b) / (c + d)
print(result)
```

Output

```
2.0
```

Why this result follows

Parentheses preserve the full numerator and denominator. The result is 10 / 5.

Boundary and error check

Substitute simple numbers and calculate by hand; state the denominator must be nonzero.

Solved problem 8.4

Guard an expression that would divide by zero.

Reasoning and algorithm

Place the nonzero test to the left of and. Python then skips division when the test is false.

Program

```
total, count = 15, 0
print(count != 0 and total / count > 5)
```

Output

```
False
```

Why this result follows

The first operand is false, so Python never evaluates the division.

Boundary and error check

Try a zero denominator and verify the program returns a Boolean rather than raising.

Solved problem 8.5

Calculate a percentage from unequal subject totals.

Reasoning and algorithm

Different subject maxima require a weighted total: add earned marks, add possible marks, then divide and multiply by 100.

Program

```
earned = 36 + 72
possible = 40 + 80
percentage = earned / possible * 100
print(percentage)
```

Output

```
90.0
```

Why this result follows

Add earned and possible marks separately. Averaging raw marks would answer a different question.

Boundary and error check

Check an all-zero earned score and a perfect score. Averaging raw marks is not the same calculation.

Trace numerator and denominator

Step	Calculation	Value
Earned	$36 + 72$	108
Possible	$40 + 80$	120
Fraction	$108 / 120$	0.9
Percent	$0.9 * 100$	90.0

A 36/40 subject and a 72/80 subject happen to have equal rates here; the weighted method also works when their rates differ.

8. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 8.1

Predict each result. Show the intermediate values and grouping.

```
30 - 12 / 3
(30 - 12) / 3
30 / 3 * 2
```

Exercise 8.2

Explain the difference between `-4 ** 2` and `(-4) ** 2`, then verify both.

Exercise 8.3

Translate the formula $(x + y)$ squared divided by $2z$ into Python, assuming z is nonzero.

Exercise 8.4

Predict the final expression value and add parentheses to show its grouping.

```
x = 8
not x < 5 and x < 10 or x == 20
```

Exercise 8.5

A price is reduced by 15% and then taxed at 8%. Write an expression; test 1,200.

CHAPTER 9

9. Errors

Build a repeatable method for finding mistakes. This chapter concentrates on the error categories and debugging expected in introductory Python.

9.1 Syntax and indentation errors

A syntax error prevents Python from parsing the program: examples include a missing colon after `if`, mismatched brackets or an unterminated string. `IndentationError` is a syntax-error subclass. Inconsistent tabs and spaces may cause `TabError`. Read the filename, line number, highlighted location and final message; the underlying mistake may be on the preceding line. Use four spaces per indentation level.

9.2 Runtime exceptions

Syntactically valid code can fail while executing. `NameError` indicates an unknown name; `TypeError` indicates an inappropriate type or combination of types; `ValueError` indicates an unsuitable value for an otherwise acceptable operation. `ZeroDivisionError`, `IndexError` and `KeyError` identify other common failures. A traceback shows the call path and location. Execution stops at an unhandled exception, so later statements do not run.

9.3 Logic errors

A logic error produces an incorrect answer without necessarily raising an exception. Examples include using `area = length + width`, forgetting to reset an accumulator and excluding the upper limit from a loop. Check the formula and expected behaviour before changing syntax. A passing test is evidence for that case, not proof that every case is correct. Boundary tests are particularly useful for comparisons and loop endpoints.

9.4 A practical debugging cycle

Reproduce the failure with the smallest useful input. Predict the correct result independently. Trace variable values or print labelled intermediate values. Locate the first point at which actual and expected values differ. Change one relevant cause, rerun the failing case and then rerun earlier passing cases to check for regressions. Remove temporary diagnostic output when finished.

9.5 Validation and assertions

Validation checks assumptions about input before using it: for example, reject a negative quantity or an empty list when an average is required. An assert statement can check an internal expectation while learning or testing, but should not be the only validation for real user input because assertions can be disabled. Import problems may appear as `ModuleNotFoundError` or `ImportError`; "linking error" is not the usual category for a basic Python script.

Visual walkthrough: Errors

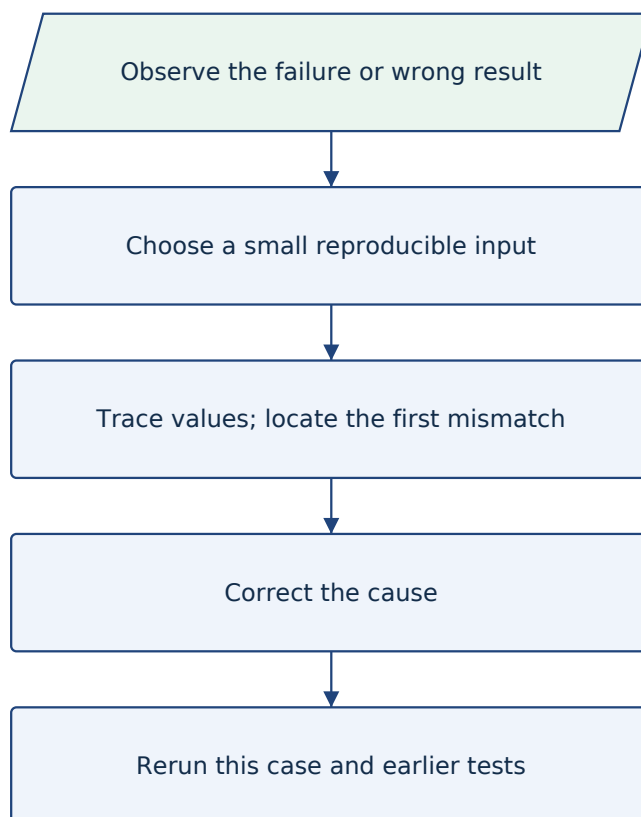


Figure 9.1. A debugging workflow. If the result is still wrong, repeat the trace rather than changing unrelated lines.

9.6 A first example

```
value = "12"  
print(int(value) + 3)
```

Output

```
15
```

The conversion prevents the `TypeError` that would arise from directly adding text and an integer. Errors are classified by cause, not merely by whether a message appears.

Quick reference

Symptom	Likely cause / repair
SyntaxError	Missing punctuation or invalid grammar
NameError	Misspelling or use before assignment
TypeError	Incompatible types; convert intentionally
ValueError	Conversion or operation rejects a value
IndexError / KeyError	Invalid sequence position / missing mapping key
Wrong result, no exception	Trace the algorithm and boundary cases

Concept bridge: Debug the first false assumption

Start from a smallest reproducible input and write the expected result. If the file cannot parse, fix its grammar before reasoning about values. If it raises, inspect the reached line and the types and values there. If it runs but gives the wrong result, trace the first point where observed state diverges from expected state. Off-by-one range errors become visible by listing visited values. A repaired program still needs a boundary test, since fixing the demonstrated case can leave an empty or one-item case broken.

9. Concept workshop: Debug the first false assumption

Start from a smallest reproducible input and write the expected result. If the file cannot parse, fix its grammar before reasoning about values. If it raises, inspect the reached line and the types and values there. If it runs but gives the wrong result, trace the first point where observed state diverges from expected state. Off-by-one range errors become visible by listing visited values. A repaired program still needs a boundary test, since fixing the demonstrated case can leave an empty or one-item case broken.

Execution model



Use the model before the code

- 9.1 Locate the intended fraction: total marks over total possible.
- 9.2 The average requires a nonzero count.
- 9.3 range excludes its stop.
- 9.4 Find where text enters a numeric expression.
- 9.5 Initialise total to zero, then update it once per payment.

Boundary habit

Check $n=0$ and $n=1$ after an off-by-one repair, not just a typical input.

Challenge the model

- 9.1 Use uneven subject maxima so the bad grouping produces a visibly wrong answer.
- 9.3 Check $n=0$ and $n=1$ after generalising the loop.
- 9.5 Check an empty payment list and a list containing zero.

9. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 9.1

Repair an average formula that forgot parentheses.

Reasoning and algorithm

Locate the intended fraction: total marks over total possible. Parentheses keep the entire denominator together before multiplication by 100.

Program

```
a, b, c = 10, 20, 30
wrong = a + b + c / 3
correct = (a + b + c) / 3
print(wrong, correct)
assert correct == 20
```

Output

```
40.0 20.0
```

Why this result follows

Division has higher precedence than addition. Test against an independently known average, not against the same faulty formula.

Boundary and error check

Use uneven subject maxima so the bad grouping produces a visibly wrong answer.

Solved problem 9.2

Prevent a division-by-zero case when computing an average.

Reasoning and algorithm

The average requires a nonzero count. Check the empty case before division and choose a documented response for it.

Program

```
total, count = 0, 0
if count == 0:
    print("No scores")
else:
    print(total / count)
```

Output

```
No scores
```

Why this result follows

Validation separates the undefined average of an empty collection from the valid numeric cases. Diagnose by phase and evidence Distinguish parsing, execution and logic. A syntax error prevents the program from starting; an exception arises on a reached state

Boundary and error check

Test [] and a one-item list; the latter should return that element.

Solved problem 9.3

Find an off-by-one error in the sum from 1 to 5.

Reasoning and algorithm

range excludes its stop. List the visited values, compare them with the target set 1 through 5, and move the stop one past 5.

Program

```
wrong = sum(range(1, 5))
correct = sum(range(1, 6))
print(wrong, correct)
assert correct == 15
```

Output

```
10 15
```

Why this result follows

range excludes its stop value. Including 5 requires stop 6.

Boundary and error check

Check n=0 and n=1 after generalising the loop.

Solved problem 9.4

Repair a missing numeric conversion.

Reasoning and algorithm

Find where text enters a numeric expression. Convert before doing arithmetic; printing alone can display text without converting it.

Program

```
raw_price = "25"
quantity = 3
print(raw_price * quantity)
print(int(raw_price) * quantity)
```

Output

```
252525
75
```

Why this result follows

String repetition is legal and therefore can hide a logic mistake. Conversion selects numeric multiplication.

Boundary and error check

Test a valid numeric spelling and a word, and point to the failing operation.

Solved problem 9.5

Trace the accumulation of three payments.

Reasoning and algorithm

Initialise total to zero, then update it once per payment. A state table after each addition reveals a skipped or doubled item.

Program

```
total = 0
for payment in [12, 8, 15]:
    total += payment
    print("payment", payment, "total", total)
assert total == 35
```

Output

```
payment 12 total 12
payment 8 total 20
payment 15 total 35
```

Why this result follows

The accumulator must be initialised before the loop. A labelled trace makes the progression 12, 20, 35 visible.

Boundary and error check

Check an empty payment list and a list containing zero.

Trace the accumulator after each item

Payment	Total before	Total after
12	0	12
8	12	20
15	20	35
End	assert total == 35	Passes

If the update were outside the loop, the assertion would expose the missing additions.

9. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 9.1

Treat each line independently. Classify the error, repair it and state whether it fails during parsing or execution.

```
print(unknown)
int("3.5")
8 / 0
"8" + 2
[7][1]
```

Exercise 9.2

A program uses $\text{perimeter} = \text{length} * \text{width}$. Give a test that exposes the logic error and write the correct formula.

Exercise 9.3

A condition uses $\text{marks} > 40$ for a pass rule of "40 or above". Supply boundary tests and repair it.

Exercise 9.4

A loop resets $\text{total} = 0$ inside its body before adding each mark. Explain its result and fix the placement.

Exercise 9.5

Design tests for an integer being inside 1 through 10 inclusive. Include both endpoints, outside values and a middle value.

CHAPTER 10

10. Conditional statements

Use selection to make a program choose an action. Translate precise rules into mutually consistent branches.

10.1 Sequence versus selection

Ordinary statements execute in sequence. An if statement conditionally executes an indented suite. The header ends with a colon, and every statement in that suite uses the same indentation. A later statement at the original indentation belongs outside the suite. Conditions are evaluated by truth value, so if name checks whether a string is nonempty; it does not check whether it contains only letters.

10.2 Choose the right structure

Use if for an optional action, if/else for two alternatives, and if/elif/else for an ordered set of alternatives. Only the first true branch in one elif chain executes. Separate if statements are independent and may all execute. An else belongs to the matching indentation level and does not have its own condition. Avoid an empty suite; pass can temporarily stand in for a statement.

10.3 Order thresholds carefully

For bands such as marks ≥ 90 , ≥ 75 and ≥ 50 , test the highest threshold first. Once a condition is true, later branches are skipped. Validate marks against 0 through 100 before classifying them. Overlapping tests can be intentional in an elif chain, but missing boundaries can leave values incorrectly classified. Write boundary examples alongside the rules.

10.4 Combine or nest conditions

and and or combine related Boolean conditions. Nest an if when a later question only matters after an earlier result; for example, first confirm that a customer is a member, then test their spending. Excessive nesting makes the logic harder to read. Use a clear combined condition when the resulting action is the same. Parentheses improve readability in mixed and/or expressions.

10.5 Reason through alternatives

Create a decision table with representative inputs and the intended branch. Test equality separately when the rule distinguishes it: the maximum of two values may have a tie. For a leap year, divisibility by 400 wins, otherwise divisible by 100 is not leap, otherwise divisible by 4 is leap. Translation must preserve the priority of exceptions in the original rule.

10.6 Menus and multi-part policies

A menu-driven calculator reads an operation and selects one calculation using an if/elif chain. Check division by zero inside the division branch; reject an unknown menu choice in the final else. For attendance rules, compute `attended / held * 100` only when held is positive, then compare with the threshold. For a bonus policy with several factors, write a decision table first: each row contains conditions and the resulting rate. This prevents an exception or boundary from disappearing inside a long chain.

Visual walkthrough: Conditional statements

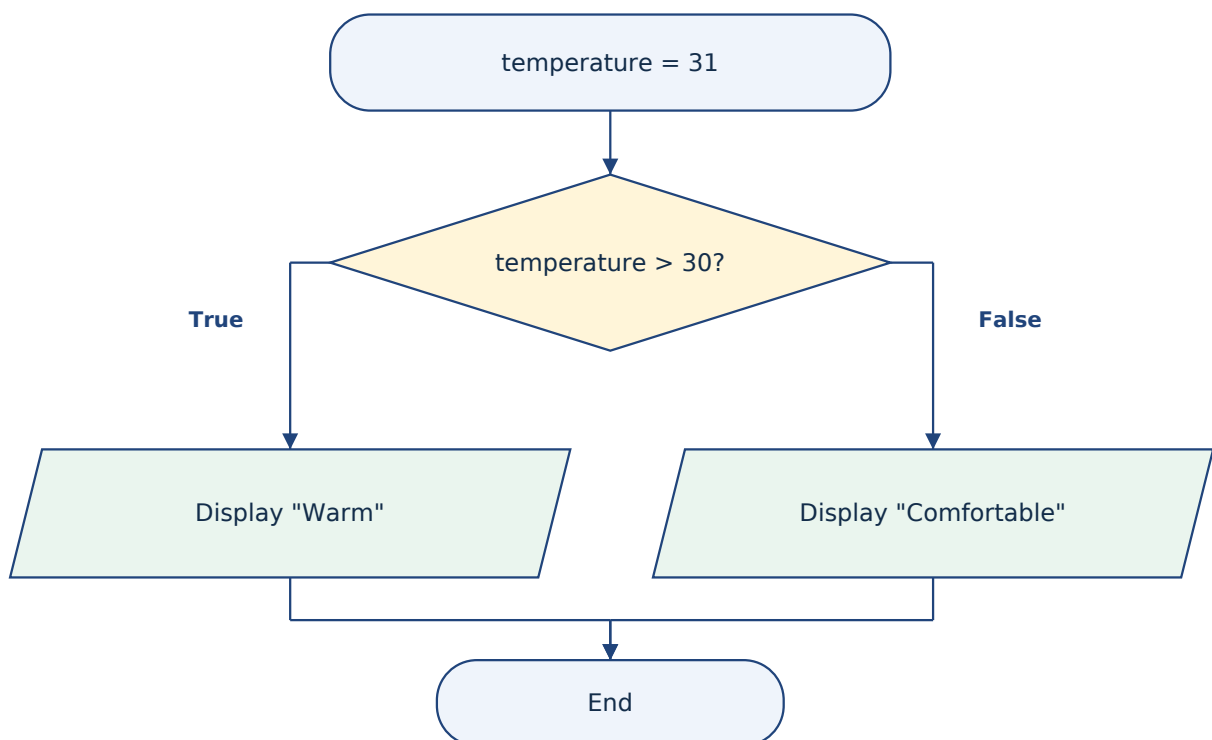


Figure 10.1. A decision diamond has two labelled exits. For 31, follow True; only the Warm branch executes.

10.7 A first example

```
temperature = 31
if temperature > 30:
    print("Warm")
else:
    print("Comfortable")
```

Output

```
Warm
```

The condition is true, so only the first indented suite executes. The else suite is skipped.

Quick reference

Pattern	Use
if condition:	Optional action
if ... else ...	Exactly one of two alternatives
if ... elif ... else ...	First matching alternative
Nested if	A decision dependent on another decision
Independent if statements	Several actions may apply
pass	Do nothing; syntactic placeholder

Concept bridge: Partition the input space

Conditions are easiest to verify as a partition. Write the allowed domain, then make cases with non-overlapping boundaries and a clear fallback. An if/elif chain stops at the first true branch, so order matters when conditions overlap. For a tiered charge, each band contributes only its own units; a final rate does not apply retroactively to earlier units. To check a policy, test one value below, at, and above every boundary and trace which branch executes. A decision diagram should show those same branches without adding extra execution steps.

10. Concept workshop: Partition the input space

Conditions are easiest to verify as a partition. Write the allowed domain, then make cases with non-overlapping boundaries and a clear fallback. An if/elif chain stops at the first true branch, so order matters when conditions overlap. For a tiered charge, each band contributes only its own units; a final rate does not apply retroactively to earlier units. To check a policy, test one value below, at, and above every boundary and trace which branch executes. A decision diagram should show those same branches without adding extra execution steps.

Execution model



Use the model before the code

10.1 The sign cases are exclusive and exhaustive.

10.2 Reject invalid scores first.

10.3 Track a current largest candidate or compare all three with $\geq$.

10.4 A year divisible by 400 is leap; divisible by 100 otherwise is not; divisible by 4 otherwise is leap.

10.5 Translate policy into ordered cases: membership may waive a fee, then an order threshold may waive it, otherwise charge it.

Boundary habit

Test -1, 0, 74, 75, 89, 90, and 101. State which branch runs each time.

Challenge the model

10.1 Check -1, 0 and 1, each of which must take exactly one branch.

10.3 Test all equal, two tied maxima, and all-negative inputs.

10.5 Test members and nonmembers on both sides of the threshold, including the exact boundary.

10. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 10.1

Classify an integer as positive, negative or zero.

Reasoning and algorithm

The sign cases are exclusive and exhaustive. Test greater than zero, less than zero, and use else for the remaining zero case.

Program

```
number = -4
if number > 0:
    print("Positive")
elif number < 0:
    print("Negative")
else:
    print("Zero")
```

Output

```
Negative
```

Why this result follows

For -4, the first comparison ($-4 > 0$) is false. The second ($-4 < 0$) is true, so Python prints Negative and skips else. Exactly zero reaches else.

Boundary and error check

Check -1, 0 and 1, each of which must take exactly one branch.

Solved problem 10.2

Award a grade after validating a score.

Reasoning and algorithm

Reject invalid scores first. Order thresholds from highest to lowest so an earlier lower threshold cannot swallow a higher grade.

Program

```
marks = 75
if not 0 <= marks <= 100:
    print("Invalid marks")
elif marks >= 90:
    print("A")
elif marks >= 75:
    print("B")
elif marks >= 50:
    print("C")
else:
    print("Needs improvement")
```

Output

```
B
```

Why this result follows

Descending thresholds prevent a high score from being captured by a lower band. Exactly 75 belongs to B.

Boundary and error check

Test values directly below, at and above every cutoff, plus -1 and 101.

Solved problem 10.3

Find the greatest of three values, allowing ties.

Reasoning and algorithm

Track a current largest candidate or compare all three with $\geq$. Ties must still produce a greatest value.

Program

```
a, b, c = 14, 14, 9
if a >= b and a >= c:
    greatest = a
elif b >= c:
    greatest = b
else:
    greatest = c
print(greatest)
```

Output

```
14
```

Why this result follows

Using `>=` handles equality. The task asks for the greatest value, not a unique winning variable.

Boundary and error check

Test all equal, two tied maxima, and all-negative inputs.

Solved problem 10.4

Determine whether a year is a leap year.

Reasoning and algorithm

A year divisible by 400 is leap; divisible by 100 otherwise is not; divisible by 4 otherwise is leap. Express the exception order clearly.

Program

```
year = 2100
if year % 400 == 0:
    leap = True
elif year % 100 == 0:
    leap = False
else:
    leap = year % 4 == 0
print(leap)
```

Output

False

Why this result follows

Century years are exceptions to the divisibility-by-four rule unless divisible by 400.

Boundary and error check

Test 1900, 2000, 2024 and 2023.

Century-year decision trace

Year	First true check	Leap?
2100	<code>2100 % 100 == 0</code>	False
2000	<code>2000 % 400 == 0</code>	True
2024	Neither century test; <code>%4 == 0</code>	True
2023	Neither; <code>%4 != 0</code>	False

Order matters: testing divisibility by 4 first would incorrectly accept 2100.

Solved problem 10.5

Calculate delivery cost using membership and order value.

Reasoning and algorithm

First select the membership branch. A member pays 0 when the order is at least 500 and 20 otherwise. A nonmember pays 50 at every order value.

Program

```
member = True
order = 450
if member:
    if order >= 500:
        delivery = 0
    else:
        delivery = 20
else:
    delivery = 50
print(delivery)
```

Output

20

Why this result follows

The outer condition selects the membership policy. Only members need the second threshold test.

Boundary and error check

Test members and nonmembers on both sides of the threshold, including the exact boundary.

Decision trace for the delivery rule

Decision	Evaluation	Outcome
member?	True	Enter member branch
order >= 500?	450 >= 500 is False	delivery = 20
print	20	One line: 20

Boundary check: member with 500 pays 0; nonmember with 500 pays 50. The nonmember branch never tests the threshold.

Test the policy as a 2 by 2 table

Member?	Order value	Delivery fee
Yes	499	20
Yes	500	0
No	499	50
No	500	50

10. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 10.1

Explain the difference between three independent if statements and an if/elif/else chain. Give a case where the number of executed branches differs.

Exercise 10.2

Calculate an electricity bill: first 100 units at 2 per unit, next 100 at 3, and remaining units at 5. Reject negative units.

Exercise 10.3

Determine whether three positive lengths form a triangle. If valid, classify it as equilateral, isosceles or scalene.

Exercise 10.4

A museum admits children under 12 for 40, people aged 12-59 for 100, and people aged 60 or above for 60. Reject negative ages.

Exercise 10.5

Find the median of three integers using decisions, without sorting. Include repeated values in your tests.

CHAPTER 11

11. For loops

Repeat an operation over a known collection or sequence of numbers. Trace iteration carefully and build accumulators correctly.

11.1 Iteration over a sequence

A for loop obtains successive elements from an iterable such as a string, list, tuple or range. The loop variable receives each element and the indented body executes. Iterating directly over values is usually clearer than indexing when positions are not needed. An empty iterable executes the body zero times, so variables assigned only in the body might never be created.

11.2 Understand range

`range(stop)` starts at zero; `range(start, stop)` chooses a start; `range(start, stop, step)` also chooses the increment. The stop value is excluded. A negative step supports descending sequences, but the endpoints must have the correct direction. A zero step raises `ValueError`. `range` is a compact range object, not an already-built list; `list(range(...))` displays its elements.

11.3 Counters and accumulators

A counter counts matching items; an accumulator combines values. Initialise a sum to 0 and a product to 1 before the loop. Update only when the task requires it. To compute an average, track both total and count and handle a zero count. To find a maximum manually, start with a real first element or track whether one has been seen; starting at zero fails when all values are negative.

11.4 Tracing a counted loop

For each iteration, write the current loop value, any test result and the updated total or count. Example: summing values 2, 4 and 6 starts total at 0 and produces totals 2, 6 and 12. The initial value is not an extra iteration. Place print outside the loop when only the final answer is required; place it inside when every step should be displayed. Nested loops have their own later chapter.

11.5 Common range mistakes

`range(1, n)` excludes `n`, while `range(1, n + 1)` includes it for a positive step. `range(5, 0)` is empty because its default step is positive; use `range(5, 0, -1)` to count down. A loop over `range(0)` executes zero times. Changing the loop variable inside the body does not alter the range object or prevent the for loop from assigning its next value. Prefer while when you need direct control of a changing stopping condition.

11.6 Series as a term rule plus accumulation

A numerical series can be written as a rule for its k th term. For squares, $\text{term} = k ** 2$; for reciprocal terms, $\text{term} = 1 / k$ with k starting at 1. Accumulate one term on each iteration. For alternating signs, maintain $\text{sign} = 1$ and negate it after each term, or use a parity test. Keep a count separate from a value limit: "first n terms" and "all terms up to value n " are different specifications. To test, expand the first three terms manually and compare with the loop.

Visual walkthrough: For loops

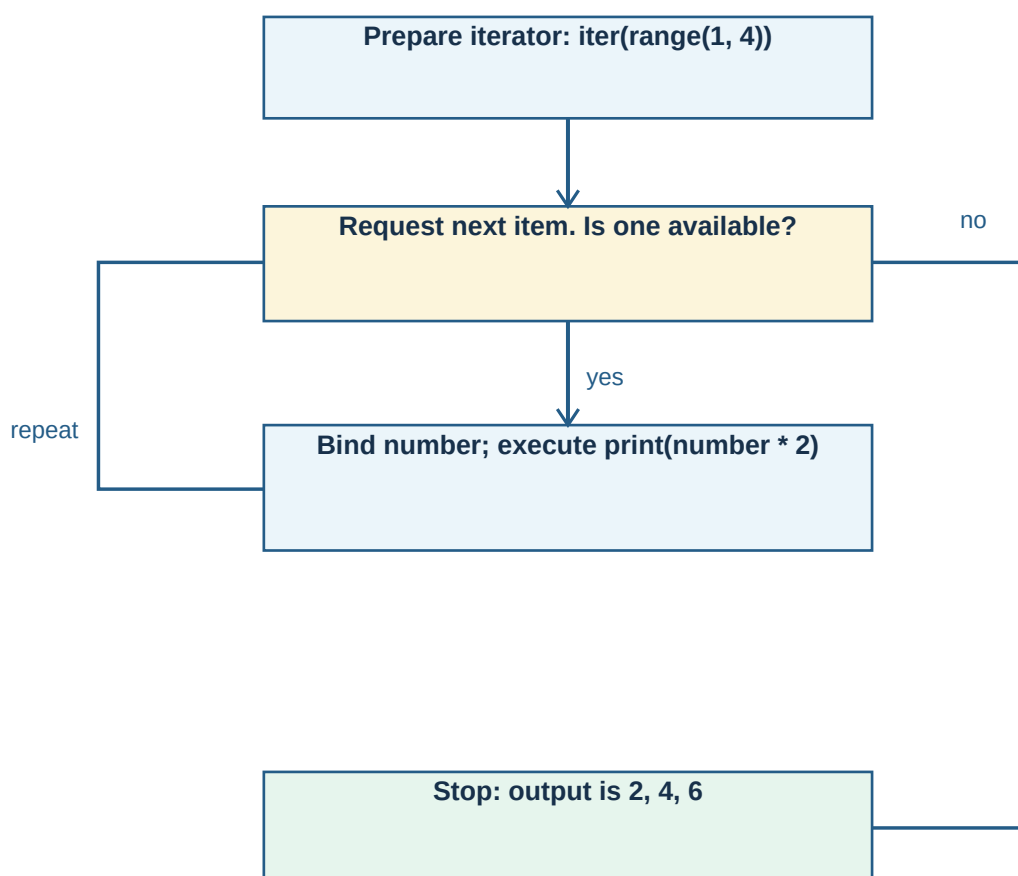


Fig Figure 11.1. Each request advances the iterator once; the body has no manual advance.

11.7 A first example

```
for number in range(1, 4):  
    print(number * 2)
```

Output

```
2  
4  
6
```

The loop variable takes 1, then 2, then 3. The stop value 4 is excluded, so the output has three lines.

Execution trace

Iteration / value	Running total for 2, 4, 6
Before loop	0
First / 2	2
Second / 4	6
Third / 6	12

Quick reference

Range	Visited values
range(4)	0, 1, 2, 3
range(2, 7, 2)	2, 4, 6
range(5, 0, -2)	5, 3, 1
range(5, 0)	No values
range(1, n + 1)	1 through n for non-negative n

CHAPTER 11 / WORKED WALKTHROUGH

Build a loop invariant for a sum

A loop invariant says what is true before every iteration. For sum of 1 through n , total equals the sum of values already visited. Set total to 0 before the first visit; add the current item; after the final visit the invariant gives the answer.

Trace the program

```
total = 0
for k in range(1, 4):
    total += k
print(total)
```

State trace

Before loop	total = 0
After k = 1	total = 1
After k = 2	total = 3
After k = 3	total = 6

A common wrong turn

The iterator supplies the next value automatically. There is no second manual advance step in a Python for statement.

Check your understanding

Use `range(1, n + 1)` for an inclusive positive end. Check $n=0$: the loop body is skipped and total stays 0.

CHAPTER 11 / WORKED WALKTHROUGH

Alternating series: derive the sign and term

For $S(n) = 1 - 1/2 + 1/3 - 1/4 + \dots$ through n terms, separate three roles: k chooses the magnitude $1/k$; sign chooses plus or minus; total accumulates. Start sign at $+1$, then flip it after each term. The invariant after k visits is the sum of the first k signed fractions.

Trace the program

```
def alternating(n):
    total = 0.0
    sign = 1
    for k in range(1, n + 1):
        total += sign / k
        sign = -sign
    return total
```

State trace

k=1	+1/1; total=1
k=2	-1/2; total=0.5
k=3	+1/3; total=0.8333...

A common wrong turn

Do not flip sign before adding the first term. For $n=0$, the empty sum is 0.0 ; reject negative n or state the assumption $n \geq 0$.

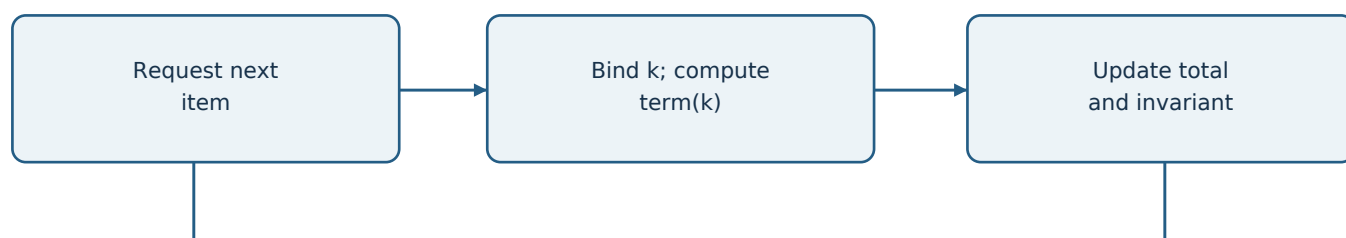
Check your understanding

Manually expand $n=4$ and compare with the function. Which variable controls count versus sign?

11. Concept workshop: Loop invariant and series rule

A counted loop needs an iterable, an initial accumulator, an update, and a stopping rule. Write what the accumulator means before each iteration. For a sum it is the total of visited terms; for a product it is the product of visited factors. A series adds a separate term rule: the k th term may be k squared, $1/k$, or an alternating signed fraction. Keep the term number separate from the term value. The Python for statement requests and binds the next item automatically, so there is no extra manual advance in the body.

Execution model



Use the model before the code

- 11.1 Start total at zero.
- 11.2 A counter increments only for values whose remainder modulo 2 is zero.
- 11.3 The product identity is one.
- 11.4 Accumulate the measurements, then divide by the number actually processed.
- 11.5 The starting value is odd and the step is two, so no parity test is needed.

Boundary habit

Use `range(1, n + 1)` for an inclusive positive end. Check $n=0$: the loop body is skipped and total stays 0.

Challenge the model

- 11.1 Trace totals 1, 5, 14, 30, 55 and check the $n=0$ empty sum.
- 11.3 Test $n=0$ and $n=1$; reject or define behavior for negative n .
- 11.5 Check an empty or reversed interval and verify 9 appears but 11 does not.

11. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 11.1

Sum the squares from 1 through 5.

Reasoning and algorithm

Start total at zero. Visit the inclusive range 1 through 5 and add the square of the current number, not the number itself.

Program

```
total = 0
for number in range(1, 6):
    total += number ** 2
print(total)
```

Output

55

Why this result follows

The stop is 6 so that 5 is included. A sum starts at zero and adds one square per iteration.

Boundary and error check

Trace totals 1, 5, 14, 30, 55 and check the n=0 empty sum.

Solved problem 11.2

Count the even values in a list.

Reasoning and algorithm

A counter increments only for values whose remainder modulo 2 is zero. Zero is even under that test.

Program

```
count = 0
for value in [3, 8, 0, 11, 4]:
    if value % 2 == 0:
        count += 1
print(count)
```

Output

3

Why this result follows

Zero is even because it has remainder zero when divided by two.

Boundary and error check

Try an empty list, all odd values, and repeated even values.

Solved problem 11.3

Calculate a factorial, including the definition of zero factorial.

Reasoning and algorithm

The product identity is one. Multiplying values 1 through n therefore gives 1 for n=0 because the loop is empty.

Program

```
n = 5
product = 1
for factor in range(1, n + 1):
    product *= factor
print(product)
```

Output

```
120
```

Why this result follows

The identity for multiplication is 1. For n=5, the running products are 1, 2, 6, 24 and 120. For n=0, the loop does not run and the result stays 1.

Boundary and error check

Test n=0 and n=1; reject or define behavior for negative n.

Solved problem 11.4

Find the total and average of four measurements.

Reasoning and algorithm

Accumulate the measurements, then divide by the number actually processed. Keep total and count conceptually distinct.

Program

```
values = [6, 8, 10, 12]
total = 0
for value in values:
    total += value
print(total, total / len(values))
```

Output

```
36 9.0
```

Why this result follows

The total is updated for each value. This example assumes the list is nonempty so its length can be a divisor.

Boundary and error check

For empty input avoid division by zero; for one value the average equals it.

Solved problem 11.5

Print odd numbers from 3 through 9.

Reasoning and algorithm

The starting value is odd and the step is two, so no parity test is needed. The stop is one past the desired last value.

Program

```
for value in range(3, 10, 2):  
    print(value)
```

Output

```
3  
5  
7  
9
```

Why this result follows

Starting with an odd number and stepping by two visits only odd values. The stop 10 is excluded.

Boundary and error check

Check an empty or reversed interval and verify 9 appears but 11 does not.

Trace the exclusive stop

Iteration	Current value	Printed
First	3	3
Second	5	5
Third	7	7
Fourth	9	9
Next	11 >= stop 10	Stop

`range(3, 10, 2)` includes 9 because the stop is excluded; it never yields 10 or 11.

11. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 11.1

Predict the output. Explain how to include the value 6 in the loop.

```
for value in range(2, 6):  
    print(value * 3)
```

Exercise 11.2

Calculate n factorial for a non-negative integer using a for loop. Verify that 0 factorial is 1.

Exercise 11.3

Count vowels in a string, ignoring case. Count every occurrence, not just distinct vowels.

Exercise 11.4

List the visited values. Explain why the stop does not appear.

```
range(8, 1, -2)
```

Exercise 11.5

Sum multiples of 3 from 3 through a non-negative limit. Test a limit below 3.

CHAPTER 12

12. While loops

Use condition-controlled repetition when the number of steps is not known in advance. Prove to yourself that every loop makes progress.

12.1 Anatomy of a while loop

A while loop tests its condition before each iteration. If the initial condition is false, the body never runs. Initialise the controlling state before the loop, test it in the header, and update it in the body. For a countdown, initialise `n` to the starting value, repeat while `n > 0` and decrease `n` each time. A missing or misplaced update can cause an infinite loop.

12.2 Sentinel-controlled input

A sentinel is a special value that ends a sequence of inputs, such as `-1` ending a list of non-negative marks. Test for the sentinel before including it in a total or count. Choose a sentinel outside the valid data domain. A while `True` loop can be appropriate when an explicit break follows input or another check, but its exit conditions must be clear.

12.3 Digit-processing algorithms

For a non-negative integer, `n % 10` extracts the final decimal digit and `n // 10` removes it. Repeating these operations supports digit sums, digit counts and reversal. Preserve an original value if needed for a later comparison. Zero requires attention: while `n > 0` executes no times for zero, although the number `0` has one decimal digit. Define how negative inputs should be handled, often by processing `abs(n)`.

12.4 Search and stopping conditions

A loop can stop because a target was found or because all possibilities were exhausted. Keep these meanings distinct using a flag, a break or a loop else. A primality test only needs candidate divisors through the square root, because factors occur in pairs. For integer code, `divisor * divisor <= n` avoids an unnecessary floating-point square root. Numbers below 2 are not prime.

12.5 Trace progress and boundary cases

List the state before and after each iteration. A suitable measure should move towards termination: `n` decreases, a divisor increases, or the list of remaining inputs shrinks. `continue` can accidentally skip an update in a while loop, so consider where the update occurs. For any iterative algorithm, test a case requiring zero iterations, one iteration and several iterations when those cases are meaningful.

12.6 Recognising digit and divisor problem families

An Armstrong test compares a non-negative integer with the sum of its digits raised to the number of digits. Preserve the original value, determine the digit count, extract digits with `% 10`, accumulate `digit ** count` and discard digits with `// 10`. A perfect-number test instead sums positive proper divisors and compares that sum with the original positive integer. Try candidate divisors from 1 through `n // 2`; `n` itself must be excluded. A palindrome reverses the digits and compares with the original. The shared skill is selecting the right state and accumulator, not memorising unrelated programs.

Visual walkthrough: While loops

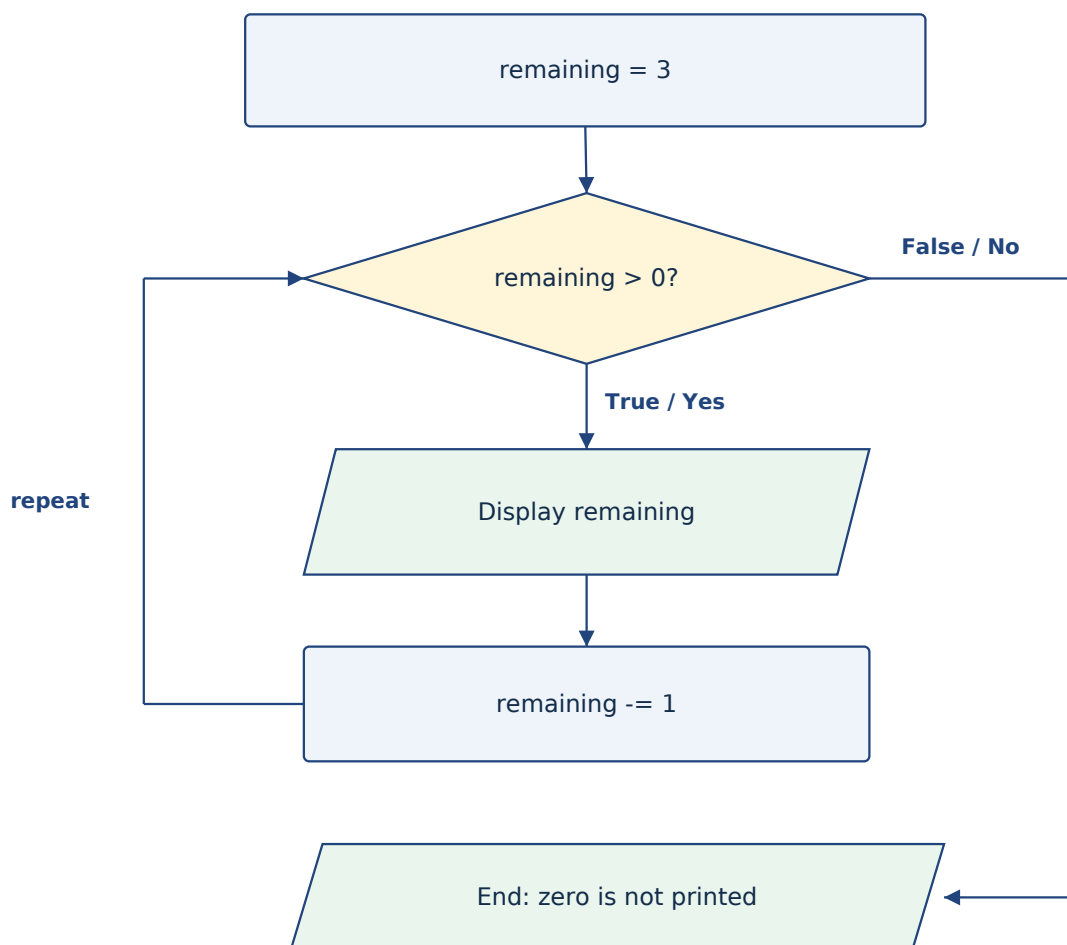


Figure 12.1. The update makes progress towards a false test. Follow the backward arrow after every update.

12.7 A first example

```
remaining = 3
while remaining > 0:
    print(remaining)
    remaining -= 1
```

Output

```
3
2
1
```

The controlling value starts at 3 and decreases after each print. The fourth test is false, so zero is not printed.

Execution trace

n before iteration	Last digit / new total / n after
5074	4 / 4 / 507
507	7 / 11 / 50
50	0 / 11 / 5
5	5 / 16 / 0

Quick reference

Pattern	Key invariant or caution
Countdown	Control value decreases towards zero
Running total	Total equals sum of processed data
Sentinel input	Sentinel is excluded from results
Digit removal	$n //= 10$ removes one decimal digit
Primality search	Only test while $\text{divisor} * \text{divisor} \leq n$
Loop termination	Every non-exiting route must make progress

CHAPTER 12 / WORKED WALKTHROUGH

A while loop must make progress

Choose state, condition and update together. For extracting digits, the number of unprocessed digits is the progress measure. `divmod(value, 10)` returns the shorter prefix and last digit. Special-case zero if the task counts its single digit.

Trace the program

```
value = 253
while value > 0:
    value, digit = divmod(value, 10)
    print(digit)
```

State trace

253	value=25, digit=3
25	value=2, digit=5
2	value=0, digit=2

A common wrong turn

If a `continue` can skip the only update, the loop may never terminate. Put progress before such a `continue` or restructure the branch.

Check your understanding

Trace `value=0` and `value=10`. Explain how a digit-counting version handles zero.

12. Concept workshop: Progress proves termination

For a while loop, state why the condition eventually becomes false. In a digit algorithm, replacing n by $n//10$ removes one decimal digit per iteration. In a sentinel loop, consuming input makes progress; in a divisor loop, the trial divisor grows. A branch containing `continue` must not bypass the only progress update. Trace zero explicitly: `while n > 0` skips its body for zero, which may or may not match the problem. Preserve the original number when the final result needs to be compared with it.

Execution model



Use the model before the code

12.1 Repeated `divmod` by 10 peels off the last digit while making the unprocessed prefix smaller.

12.2 Build the reversed number by shifting the previous result one decimal place and appending the removed last digit.

12.3 A composite number has a divisor no larger than its square root.

12.4 Keep the two most recent terms.

12.5 First determine the digit count, then sum each digit raised to that count.

Boundary habit

Trace `value=0` and `value=10`. Explain how a digit-counting version handles zero.

Challenge the model

12.1 A number ending in zero still contributes digit zero. Treat input zero as one digit when counting digits.

12.3 0 and 1 are not prime; test 2, a square such as 49, and a prime.

12.5 Test 0, a single digit, 370, and a nearby non-Armstrong value.

12. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 12.1

Calculate the sum of the digits of 5074.

Reasoning and algorithm

Repeated divmod by 10 peels off the last digit while making the unprocessed prefix smaller. Accumulate each digit once.

Program

```
n = 5074
total = 0
while n > 0:
    total += n % 10
    n //= 10
print(total)
```

Output

```
16
```

Why this result follows

The loop removes 4, 7, 0 and 5 in that order. The running totals are 4, 11, 11 and 16; after *n* becomes 0, the condition is false.

Boundary and error check

A number ending in zero still contributes digit zero. Treat input zero as one digit when counting digits.

Solved problem 12.2

Reverse a non-negative integer.

Reasoning and algorithm

Build the reversed number by shifting the previous result one decimal place and appending the removed last digit.

Program

```
n = 1204
reverse = 0
while n > 0:
    reverse = reverse * 10 + n % 10
    n //= 10
print(reverse)
```

Output

```
4021
```

Why this result follows

Each step shifts reverse left one decimal place and appends $n \% 10$. For 1204 the successive reverse values are 4, 40, 402 and 4021.

Boundary and error check

Check 0 and an input such as 120, whose numeric reverse drops the leading zero.

Solved problem 12.3

Determine whether 29 is prime.

Reasoning and algorithm

A composite number has a divisor no larger than its square root. Try divisors from 2 while $d*d \leq n$; stop on the first factor.

Program

```
n = 29
divisor = 2
prime = n >= 2
while prime and divisor * divisor <= n:
    if n % divisor == 0:
        prime = False
    divisor += 1
print(prime)
```

Output

True

Why this result follows

If no factor at or below the square root exists, no larger factor pair can exist either. The initial flag rejects values below 2.

Boundary and error check

0 and 1 are not prime; test 2, a square such as 49, and a prime.

Trace the prime candidate

Divisor	$\text{divisor} * \text{divisor} \leq 29$	$29 \% \text{divisor}$
2	$4 \leq 29$: True	1: continue
3	$9 \leq 29$: True	2: continue
4	$16 \leq 29$: True	1: continue
5	$25 \leq 29$: True	4: continue
6	$36 \leq 29$: False	Stop; prime True

For 49, divisor 7 reaches $7*7 \leq 49$ and divides evenly. Stopping before the square root would be wrong.

Solved problem 12.4

Generate the first seven Fibonacci terms beginning with 0 and 1.

Reasoning and algorithm

Keep the two most recent terms. Emit the current term, then update the pair simultaneously so the old right value is not lost.

Program

```
a, b = 0, 1
count = 0
terms = []
while count < 7:
    terms.append(a)
    a, b = b, a + b
    count += 1
print(terms)
```

Output

```
[0, 1, 1, 2, 3, 5, 8]
```

Why this result follows

Simultaneous assignment uses both old values. A separate counter controls the requested number of terms.

Boundary and error check

Test zero terms, one term, and the requested seven-term prefix.

Trace both Fibonacci registers

Count before	(a, b) before append	Terms after append
0	(0, 1)	[0]
1	(1, 1)	[0, 1]
2	(1, 2)	[0, 1, 1]
3	(2, 3)	[0, 1, 1, 2]
6	(8, 13)	[0, 1, 1, 2, 3, 5, 8]

At count 7 the condition fails. In `a, b = b, a + b`, both right-hand values use the old pair.

Solved problem 12.5

Check whether 370 is an Armstrong number.

Reasoning and algorithm

First determine the digit count, then sum each digit raised to that count. Compare the sum with the original value, which must be preserved.

Program

```
original = 370
n = original
power = len(str(original))
total = 0
while n > 0:
    digit = n % 10
    total += digit ** power
    n //= 10
print(total == original)
```

Output

```
True
```

Why this result follows

The removed digits are 0, 7 and 3. Their cubes contribute 0, 343 and 27, so the total is 370. Zero also compares equal to its empty-loop total 0.

Boundary and error check

Test 0, a single digit, 370, and a nearby non-Armstrong value.

Trace digit extraction

n before	Digit / contribution	Total after
370	0 / $0^{**3} = 0$	0
37	7 / $7^{**3} = 343$	343
3	3 / $3^{**3} = 27$	370
0	Condition false	$370 == \text{original}$

Keep original unchanged while n shrinks. If the program compared total with n at the end, it would compare with zero.

12. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 12.1

Trace the countdown. State the printed values and the number of condition tests.

```
n = 4
while n > 0:
    print(n)
    n -= 2
```

Exercise 12.2

Check whether a non-negative integer is a palindrome by reversing its digits. Preserve the original value.

Exercise 12.3

Read non-negative marks until -1 is entered. Print the count and average; handle an immediate sentinel without dividing by zero.

Exercise 12.4

Determine whether a positive integer is perfect by summing its proper divisors. Test 6, 12 and 1. Explain why the number itself is excluded.

Exercise 12.5

Print all prime numbers from 2 through 50 using nested iteration. Explain why testing a number against itself would be incorrect.

CHAPTER 13

13. Nested loops

A nested loop places one loop inside another. It describes a repeated task that itself contains repetition, such as visiting the cells of a table.

13.1 Two levels of repetition

For each outer-loop iteration, the inner loop starts again and runs through its own values. If the outer loop has three iterations and the inner loop has four each time, the inner body executes twelve times. Use different variable names such as `row` and `column`. Reusing the same name can hide the outer value and make the trace misleading.

13.2 Rows and columns

Think of a rectangular grid as rows, with several columns in each row. The outer loop chooses a row; the inner loop visits its columns. `print(value, end=" ")` can keep several entries on one output line. A `print()` after the inner loop ends the row. Its indentation is critical: if placed inside the inner loop, every cell would end its own line.

13.3 Variable-length rows

A triangle has a different number of cells in each row. If row numbers begin at 1, `range(row)` performs exactly `row` iterations. The total number of inner iterations for `n` rows is $1 + 2 + \dots + n$, rather than n squared. Decide what one row should look like before coding. Then express its length and contents using the row number.

13.4 Nested collections

A list of row lists stores a table. `table[r][c]` first selects row `r` and then item `c` within it. A nested tuple works similarly for fixed data. Lists are explained fully later; here focus on the two-level traversal. Rows need not all have the same length, so traversing each row directly is safer when rectangular shape has not been guaranteed.

13.5 Reasoning about cost and correctness

A nested loop is not automatically wrong or slow; its cost depends on the number of iterations. For small classroom tables, direct traversal is clear and useful. Trace one complete outer iteration before reasoning about all rows. If you use `break` inside an inner loop, it exits only that inner loop. The loop-control chapter explains how to represent a search result explicitly.

13.6 Designing new patterns

Separate the row shape from the printed value. For row r of a right-aligned triangle with n rows, first print $n-r$ spaces and then r stars. For a descending numeric row, choose a negative-step range. For a repeated row label, print r on each inner iteration. For consecutive numbers continuing across rows, place a counter before the outer loop and increment it inside the inner loop. Draw three rows by hand first; derive spaces, item count and item value separately.

Visual walkthrough: Nested loops

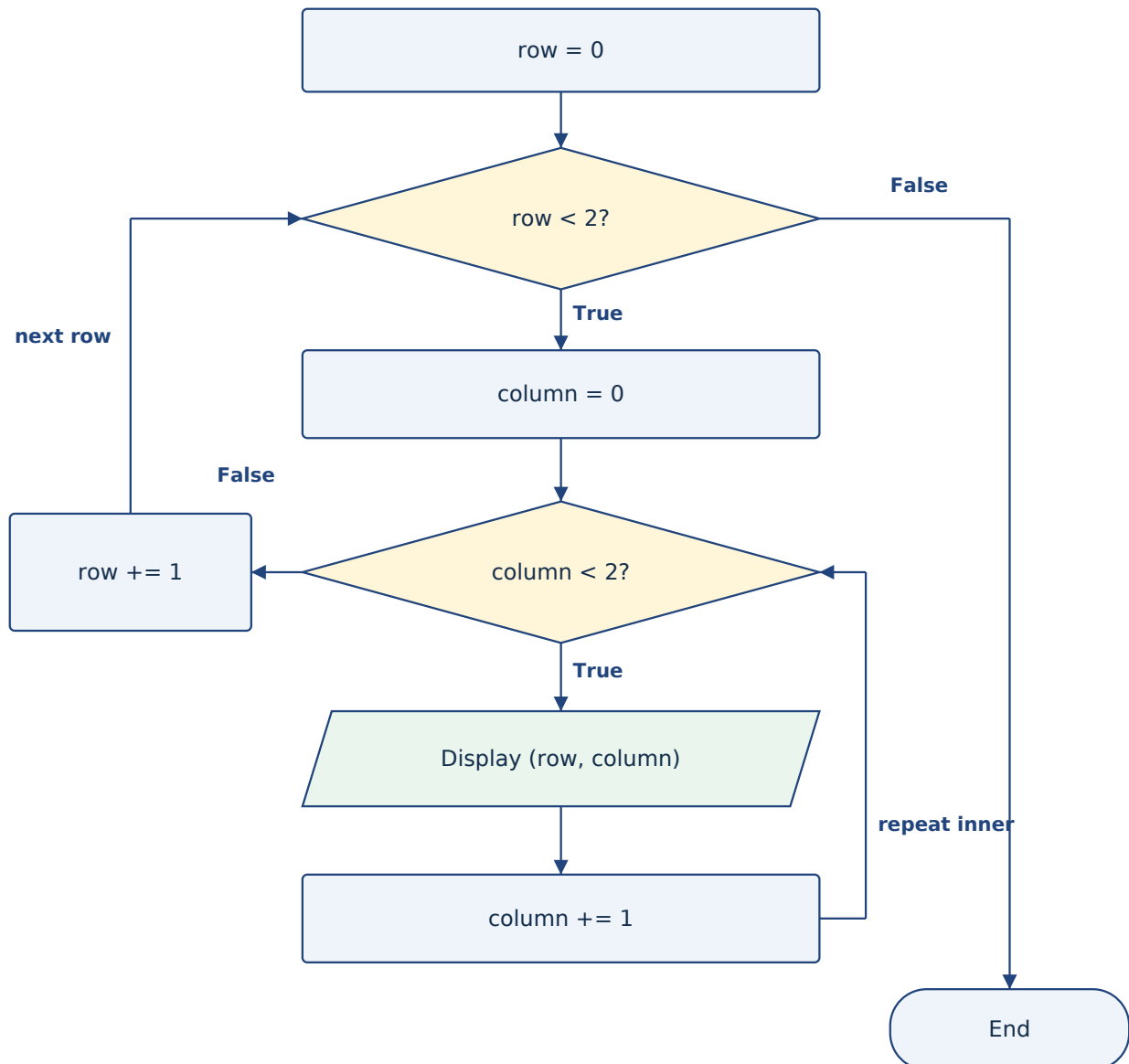


Figure 13.1. The inner counter resets for each outer iteration. Trace the pairs (0,0), (0,1), (1,0), (1,1).

13.7 A first example

```
for row in range(2):  
    for column in range(2):  
        print(row, column)
```

Output

```
0 0  
0 1  
1 0  
1 1
```

The pair sequence is (0,0), (0,1), (1,0), (1,1). The inner traversal completes before the outer value changes.

Concept bridge: A grid of iterations

Nested loops form a grid: each outer value gets a complete inner traversal. A row-oriented output task has three levels of action: choose a row, choose each column within that row, then end the row. Build a row table with desired widths before writing Python. In a numeric triangle, a counter may persist across rows, even though the inner column count resets each row. Runtime grows roughly with the number of visited row-column pairs, not just the number of outer rows.

13. Concept workshop: A grid of iterations

Nested loops form a grid: each outer value gets a complete inner traversal. A row-oriented output task has three levels of action: choose a row, choose each column within that row, then end the row. Build a row table with desired widths before writing Python. In a numeric triangle, a counter may persist across rows, even though the inner column count resets each row. Runtime grows roughly with the number of visited row-column pairs, not just the number of outer rows.

Execution model



Use the model before the code

13.1 The outer loop fixes the first coordinate; the inner loop runs through every second coordinate before the outer loop advances.

13.2 For row r , print r stars and then one newline.

13.3 Each row has one entry per column.

13.4 Reset the row subtotal for each row, then add it to the grand total; alternatively sum each element once in the nested traversal.

13.5 Carry a next-number counter across row boundaries.

Boundary habit

Predict output for three rows, then change `range(row)` to `range(4-row)`.

Challenge the model

13.1 Count rows times columns and verify the exact visit order.

13.3 Check the first row and column separately to catch off-by-one indexing.

13.5 For four rows the total printed count is $1+2+3+4=10$.

13. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 13.1

Trace all pairs from two short ranges.

Reasoning and algorithm

The outer loop fixes the first coordinate; the inner loop runs through every second coordinate before the outer loop advances.

Program

```
for row in range(1, 3):  
    for column in range(1, 4):  
        print(row, column)
```

Output

```
1 1  
1 2  
1 3  
2 1  
2 2  
2 3
```

Why this result follows

For row 1, the inner loop prints (1,1), (1,2), (1,3). It restarts at column 1 for row 2, printing (2,1), (2,2), (2,3).

Boundary and error check

Count rows times columns and verify the exact visit order.

Solved problem 13.2

Print a left-aligned triangle of four rows.

Reasoning and algorithm

For row *r*, print *r* stars and then one newline. Put the newline after, not inside, the inner loop.

Program

```
for row in range(1, 5):  
    for column in range(row):  
        print("*", end="")  
    print()
```

Output

```
*
**
***
****
```

Why this result follows

On row 1, the inner loop prints one star, then `print()` ends the line. On row 4, it prints four stars before the single newline.

Boundary and error check

Check one row and zero rows; use a row-count table before writing the loops.

Solved problem 13.3

Build a small multiplication grid.

Reasoning and algorithm

Each row has one entry per column. Compute `row*column` at the inner level and end the row only after all columns are printed.

Program

```
for row in range(1, 4):
    line = ""
    for column in range(1, 4):
        line += str(row * column) + " "
    print(line.strip())
```

Output

```
1 2 3
2 4 6
3 6 9
```

Why this result follows

Each row starts with empty text. `str` converts each product so it can be concatenated; `strip` removes the final outside space.

Boundary and error check

Check the first row and column separately to catch off-by-one indexing.

Trace one complete grid row

Column	Product (row = 2)	line after append
1	2 * 1 = 2	"2 "
2	2 * 2 = 4	"2 4 "
3	2 * 3 = 6	"2 4 6 "

After column 3, `strip` removes the trailing space and `print` ends row 2. The next row resets `line` to "".

Solved problem 13.4

Calculate the total of a nested list.

Reasoning and algorithm

Reset the row subtotal for each row, then add it to the grand total; alternatively sum each element once in the nested traversal.

Program

```
table = [[2, 3], [4, 5], [6, 7]]
total = 0
for row in table:
    for value in row:
        total += value
print(total)
```

Output

```
27
```

Why this result follows

The accumulator belongs outside both loops because the task asks for one grand total.

Boundary and error check

Test an empty table and rows of unequal lengths if the algorithm allows them.

Solved problem 13.5

Print an increasing numeric pattern.

Reasoning and algorithm

The inner range starts at 1 for every new row and ends at that row number. This program restarts the digits on each row; it has no counter carried across rows.

Program

```
for row in range(1, 4):
    for value in range(1, row + 1):
        print(value, end="")
    print()
```

Output

```
1
12
123
```

Why this result follows

Each row begins again at 1 and includes the row number as its final digit.

Boundary and error check

For four rows the total printed count is $1+2+3+4=10$.

13. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 13.1

How many times does the inner body execute for `range(4)` outside and `range(3)` inside? Explain rather than just multiply.

Exercise 13.2

Predict the order of pairs for outer values 0, 1 and inner values 2, 3. Write a trace table.

Exercise 13.3

Print a five-row descending star triangle with row lengths 5, 4, 3, 2 and 1.

Exercise 13.4

Write a nested-loop program that calculates each row total separately for `[[3, 5], [2, 8], [4, 1]]`. Explain where the row accumulator resets.

Exercise 13.5

Print a 4-by-4 multiplication table and count how many products exceed 8. Include the final count once.

CHAPTER 14

14. Loop control

Use `break`, `continue`, `pass` and `loop else` precisely. Each changes a different part of the program's flow.

14.1 `break`: finish the nearest loop

`break` immediately exits the nearest surrounding `for` or `while` loop. Statements after `break` in the current body do not execute. Execution continues with the first statement after that loop. A search can stop when a match is found, avoiding unnecessary work. In nested loops, breaking the inner loop does not automatically stop the outer one.

14.2 `continue`: skip the current remainder

`continue` skips the rest of the current body and starts the next iteration. In a `for` loop, the next element is obtained. In a `while` loop, the condition is tested again. Be careful that `continue` does not bypass the update that makes a `while` loop terminate. A clear condition around the useful action may be simpler than `continue` for a very short body.

14.3 `pass`: do nothing

`pass` is a valid statement that takes no action. It can fill a suite that cannot be left syntactically empty. A comment alone does not count as a statement. `pass` does not exit a loop or skip subsequent statements: code below it still runs. Use it as a temporary placeholder, then replace it with the intended work.

14.4 The meaning of `loop else`

An `else` attached to a loop runs when the loop completes without `break`. For a `for` loop, completion means the iterable is exhausted; for a `while` loop, it means the condition became false. It can also run when the body executes zero times. The `else` is associated with the loop by indentation, not with the last `if` inside it. Read it as "no `break` occurred".

14.5 Choose an explicit stopping rule

In a sentinel-input loop, a special value means stop and must not be counted as normal data. `while True` makes the condition permanently true, so the body must contain a reachable `break` or some other explicit exit. Prefer a natural `while` condition when it expresses the task clearly. Test a match at the start, at the end and absent from the data to verify a search.

Visual walkthrough: Loop control

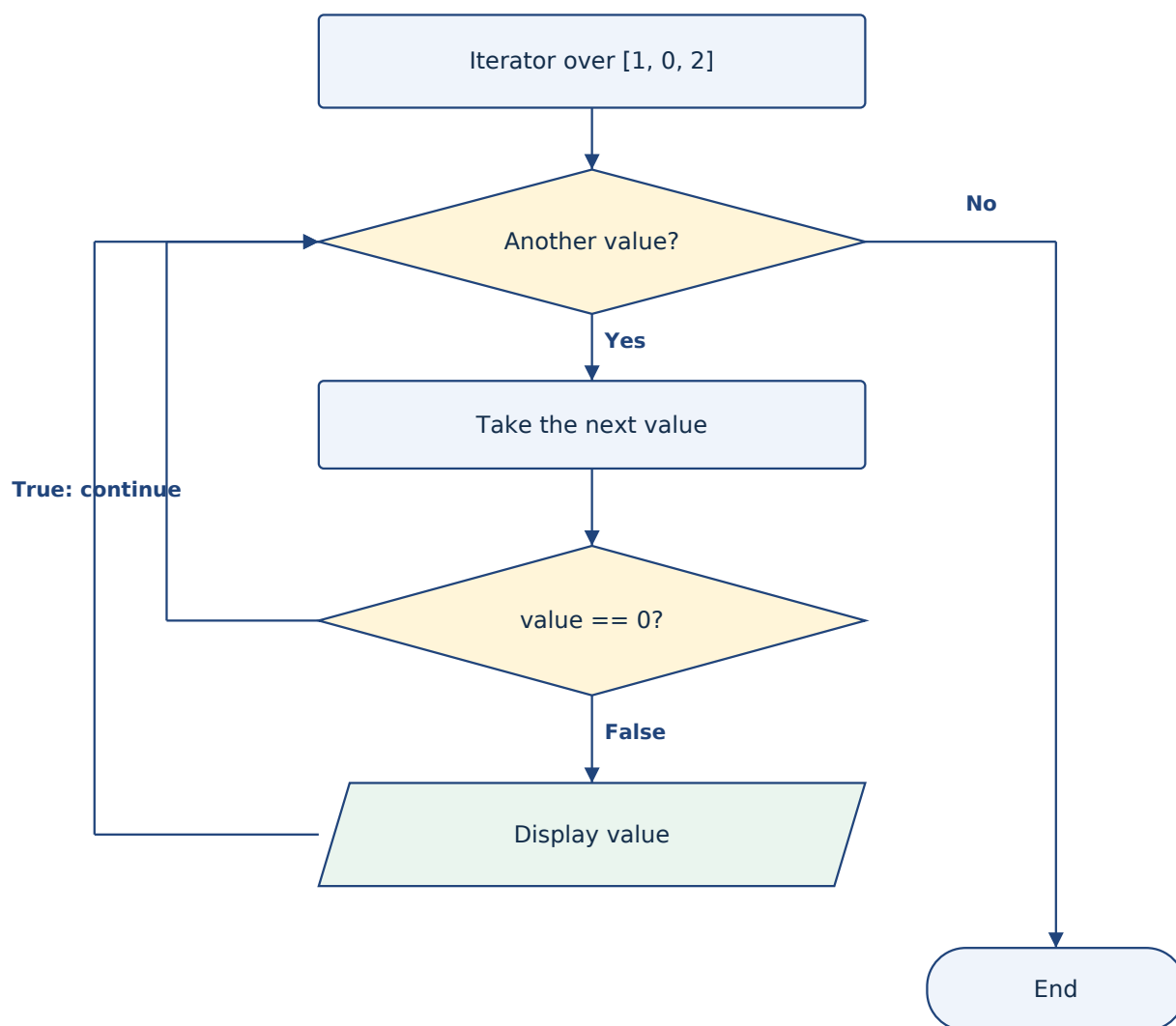


Figure 14.1. `continue` skips `Display` for the value zero and returns to the next-item test. Exhausting the iterator ends the loop.

14.6 A first example

```
for value in [1, 0, 2]:
    if value == 0:
        continue
    print(value)
```

Output

```
1
2
```

The zero is visited but the `print` statement is skipped for that iteration. The following 2 still prints.

CHAPTER 14 / WORKED WALKTHROUGH

Control flow changes which work runs

Use `continue` to skip the rest of one iteration, `break` to end the current loop, and `pass` as an empty statement. A `loop else` runs only if the loop finishes without `break`. In nested loops, `break` ends only the innermost loop.

Trace the program

```
for x in [2, 4, 7]:
    if x % 2:
        print("odd", x)
        break
    else:
        print("none")
```

State trace

x=2	No break
x=4	No break
x=7	Print odd 7; break
loop else	Skipped

A common wrong turn

A search for a nested-table value needs an outer-loop signal or a return from a function to stop both loops.

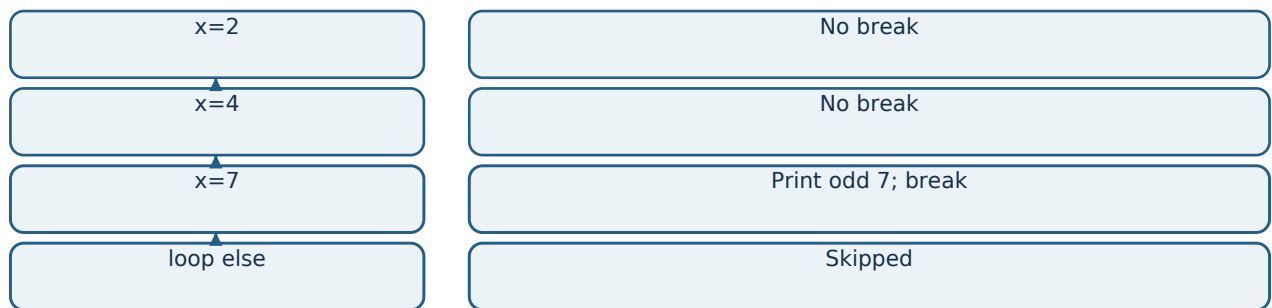
Check your understanding

Change the list to `[2, 4, 6]`. Which line prints? What happens for `[]`?

14. Concept workshop: Three distinct control decisions

continue skips remaining work for the current visit and then tries the next one. break abandons the current loop altogether. pass does nothing and continues with the following statement. For a search, loop else is a convenient completion signal because it runs only when no break happened. A break inside the inner loop does not magically stop an outer loop; communicate the found state outward, or return from a function. When reading a loop, mark every path that can skip an update or exit early.

Execution model



Use the model before the code

- 14.1 Scan in order and break at the first divisible value.
- 14.2 Use continue only for invalid negative payments, and add other values exactly once.
- 14.3 pass does nothing.
- 14.4 A for/else reports failure only if the scan finishes without break.
- 14.5 Check the sentinel before adding it.

Boundary habit

Change the list to [2, 4, 6]. Which line prints? What happens for []?

Challenge the model

- 14.1 Test no matches and a match at the first element.
- 14.3 Replace pass with continue and compare the outputs to see the difference.
- 14.5 Test sentinel first, sentinel absent, and a second sentinel later in the list.

14. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 14.1

Find the first multiple of 5.

Reasoning and algorithm

Scan in order and break at the first divisible value. An early break prevents later matching values from replacing the first answer.

Program

```
for value in [3, 7, 10, 15]:
    if value % 5 == 0:
        print(value)
        break
```

Output

```
10
```

Why this result follows

The loop stops at 10. The later matching value 15 is never visited.

Boundary and error check

Test no matches and a match at the first element.

Solved problem 14.2

Skip negative values when adding payments.

Reasoning and algorithm

Use continue only for invalid negative payments, and add other values exactly once. The update should be below the skip test.

Program

```
total = 0
for value in [4, -3, 6, -1]:
    if value < 0:
        continue
    total += value
print(total)
```

Output

```
10
```

Why this result follows

Only 4 and 6 reach the addition statement. `continue` does not stop the complete loop.

Boundary and error check

Test all negative values, zero, and a mixed list.

Solved problem 14.3

Show that `pass` does not skip the next statement.

Reasoning and algorithm

`pass` does nothing. Execution continues at the next statement in the same block; it is a placeholder, not a skip instruction.

Program

```
for value in range(3):
    if value == 1:
        pass
    print(value)
```

Output

```
0
1
2
```

Why this result follows

All three values print. `pass` is a no-operation statement, not a synonym for `continue`.

Boundary and error check

Replace `pass` with `continue` and compare the outputs to see the difference.

Solved problem 14.4

Report an unsuccessful search using `loop else`.

Reasoning and algorithm

A `for/else` reports failure only if the scan finishes without `break`. Place the success action before `break` and the failure action in `else`.

Program

```
for value in [2, 4, 6]:
    if value == 9:
        print("Found")
        break
else:
    print("Not found")
```

Output

```
Not found
```

Why this result follows

The values 2, 4 and 6 are each unequal to 9. The loop finishes without break, so its else suite prints Not found.

Boundary and error check

Test a first-element match, a final-element match, and an empty list.

Solved problem 14.5

Process a sentinel-terminated sequence.

Reasoning and algorithm

Check the sentinel before adding it. break ends the scan immediately, so all values after the sentinel are ignored.

Program

```
total = 0
for value in [8, 5, -1, 20]:
    if value == -1:
        break
    total += value
print(total)
```

Output

13

Why this result follows

The sentinel is tested before addition. It and every later value are excluded.

Boundary and error check

Test sentinel first, sentinel absent, and a second sentinel later in the list.

Trace the sentinel before adding

Value	Sentinel test	Total
8	False	8
5	False	13
-1	True; break	13
20	Never visited	13

Moving the sentinel test below total += value would add -1 incorrectly.

14. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 14.1

Explain the difference between `break`, `continue` and `pass` using one sentence and one small example for each.

Exercise 14.2

Predict the output. Explain separately the effects of `continue` and `break`.

```
for value in range(5):  
    if value == 1:  
        continue  
    if value == 3:  
        break  
    print(value)
```

Exercise 14.3

Write a search for the first negative value in a list, using `loop else` when no negative value exists.

Exercise 14.4

Repair a `while` loop where `continue` occurs before the only counter update. Show why the original can repeat forever.

Exercise 14.5

Find the first table match with nested loops; stop both loops after it is found.

CHAPTER 15

15. Strings

Represent and transform text using indexing, slicing, traversal and the string methods covered in the KIPS string chapter.

15.1 Indexing and immutability

A string is an immutable sequence of Unicode code points. Index `0` selects the first character; `-1` selects the last. A string of length `n` has non-negative valid indices `0` through `n-1`. An invalid single index raises `IndexError`. You cannot assign to a character position; create a new string instead. `len` counts code points, which is not always the number of visually perceived symbols for complex Unicode text.

15.2 Slicing and operators

`s[start:stop:step]` selects characters up to, but excluding, `stop`. Missing bounds default according to the step direction. `s[::-1]` reverses a string; a zero step is invalid. Slice bounds outside the string are clipped, unlike a single out-of-range index. `+` concatenates, `*` repeats and `in` tests for a substring. String comparisons are lexicographic and case-sensitive; do not assume they follow a human dictionary's ordering.

15.3 Searching and checking text

`find` returns the first matching index or `-1`; `index` raises `ValueError` when the substring is absent. `count` counts non-overlapping occurrences. `startswith` and `endswith` check boundaries. `isalpha`, `isdigit`, `isalnum` and `isspace` require at least one character and check all characters. These methods are Unicode-aware: `isdigit` is not a guarantee that every matching string can be converted by `int`. `islower` and `isupper` require at least one cased character; digits do not make them false.

15.4 Changing case and whitespace

`lower` and `upper` return converted copies. `capitalize` uppercases the first character and lowercases the remainder; `title` uses word-like boundaries. `istitle` checks title-case structure. `strip` removes leading and trailing whitespace by default, while `lstrip` and `rstrip` affect one end. An argument to `strip` is a set of characters to remove repeatedly, not an exact prefix. `replace(old, new, count)` returns a copy with replacements, optionally limited by `count`.

15.5 Split, join and partition

`split()` without an argument groups whitespace and omits leading/trailing empty fields. `split(",")` uses an exact separator and preserves empty fields between adjacent commas. `separator.join(strings)` joins an iterable of strings; numbers must be converted first. `partition(separator)` returns (before, separator, after) at the first occurrence; if absent, it returns (original, "", ""). Traverse characters directly when counting or filtering, and avoid repeatedly modifying the same immutable string for very large workloads.

15.6 Character codes and small transformations

`ord(character)` gives the Unicode code-point number for one character; `chr(number)` constructs the corresponding one-character string. This is useful for studying character representation, but adding one to a code point is not a general way to move to the next letter in every language. For counting character categories, decide whether spaces belong to special symbols or their own category. For a name-initials task, split into words, take each word's first character and join those initials deliberately.

15.7 From a text rule to a new string

Because strings are immutable, a transformation builds another string. Start result as empty text, traverse the original and choose what to append for each character. A substitution table can map selected letters to replacement symbols; later, dictionaries provide a natural representation. To swap commas and dots, decide from each original character which replacement to append. Two naive chained replace calls can accidentally replace newly inserted characters a second time. To combine selected parts of two strings, calculate their index boundaries separately before concatenating.

Visual walkthrough: Strings

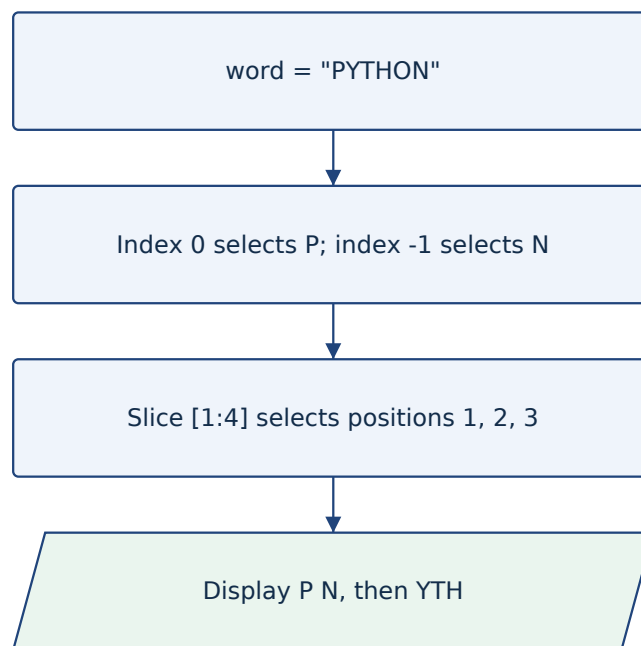


Figure 15.1. Indexing selects one position; a slice selects a range whose stop is excluded. The original string remains unchanged.

15.8 A first example

```
word = "PYTHON"
print(word[0], word[-1])
print(word[1:4])
```

Output

```
P N
YTH
```

Indices select positions, beginning at zero. The slice contains positions 1, 2 and 3; position 4 is excluded.

Quick reference

Method / operation	Result or important behaviour
<code>s[i]</code> , <code>s[a:b:c]</code> , <code>len(s)</code>	Single character, slice, length
<code>find</code> / <code>index</code> / <code>count</code>	Position or -1 / position or error / occurrence count
<code>isalpha</code> / <code>isdigit</code> / <code>isalnum</code>	Letters / digits / letters-or-digits; nonempty required
<code>isspace</code> / <code>islower</code> / <code>isupper</code> / <code>istitle</code>	Whitespace / lower / upper / title checks
<code>lower</code> / <code>upper</code> / <code>capitalize</code> / <code>title</code>	New strings with changed case
<code>strip</code> / <code>lstrip</code> / <code>rstrip</code>	Trim both / left / right edges
<code>replace</code> / <code>split</code> / <code>join</code> / <code>partition</code>	Transform / tokenize / combine / split once
<code>startswith</code> / <code>endswith</code>	Prefix / suffix Boolean checks

Concept bridge: Indexes, slices and transformations

An index denotes one existing position; a slice describes a range and tolerates an end beyond the string. A negative step reverses the traversal direction, so start and stop must be interpreted accordingly. Strings are immutable: cleaning text builds a new string or a list of pieces to join. To find overlapping occurrences, resume searching one position after the previous match rather than after the whole match. Normalising case and whitespace before a comparison makes the chosen equivalence relation explicit.

15. Concept workshop: Indexes, slices and transformations

An index denotes one existing position; a slice describes a range and tolerates an end beyond the string. A negative step reverses the traversal direction, so start and stop must be interpreted accordingly. Strings are immutable: cleaning text builds a new string or a list of pieces to join. To find overlapping occurrences, resume searching one position after the previous match rather than after the whole match. Normalising case and whitespace before a comparison makes the chosen equivalence relation explicit.

Execution model



Use the model before the code

- 15.1 Find the final dot, then slice the prefix and suffix.
- 15.2 Normalise case once, then inspect each character.
- 15.3 Build a cleaned comparison string by keeping only relevant alphanumeric characters in one case.
- 15.4 Split names into words.
- 15.5 split without an argument collapses runs of whitespace.

Boundary habit

Test an empty string and a string with no match. Predict the output before running.

Challenge the model

- 15.1 Test no dot, a leading dot, and multiple dots; state the chosen policy for each.
- 15.3 Test punctuation-only text and a near-palindrome with one mismatched letter.
- 15.5 Test leading whitespace, newlines, and an input containing no words.

15. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 15.1

Extract a filename stem and extension using slicing.

Reasoning and algorithm

Find the final dot, then slice the prefix and suffix. This works differently from splitting on every dot when a filename contains several.

Program

```
filename = "lesson.py"
print(filename[:6])
print(filename[-2:])
print(filename[::-1])
```

Output

```
lesson
py
yp.nossel
```

Why this result follows

The stop index is excluded. Negative indexing counts from the end; a step of -1 reverses the sequence. Track indexes and slices distinctly Indexing asks for one position and raises `IndexError` if it does not exist. Slicing asks for a range

Boundary and error check

Test no dot, a leading dot, and multiple dots; state the chosen policy for each.

Solved problem 15.2

Normalise a name and count its vowels.

Reasoning and algorithm

Normalise case once, then inspect each character. A vowel count increments for every occurrence, not merely every distinct vowel.

Program

```
name = "  Anika Rao  "
clean = name.strip().title()
count = 0
for character in clean.lower():
    if character in "aeiou":
        count += 1
print(clean, count)
```

Output

```
Anika Rao 5
```

Why this result follows

strip removes outside whitespace; the internal space remains. Converting to lowercase avoids duplicating vowel tests.

Boundary and error check

Check an empty name, mixed case, and repeated vowels.

Solved problem 15.3

Check a phrase palindrome while ignoring spaces and punctuation.

Reasoning and algorithm

Build a cleaned comparison string by keeping only relevant alphanumeric characters in one case. Compare it with its reverse.

Program

```
phrase = "Never odd, or even!"
clean = ""
for character in phrase.lower():
    if character.isalnum():
        clean += character
print(clean == clean[::-1])
```

Output

```
True
```

Why this result follows

Filtering and case normalisation define the comparison rule explicitly. For this task, an empty cleaned phrase would compare equal to its reverse. Lowercase phrase; clean = ""
No More characters? Yes Take next character False isalnum()?

Boundary and error check

Test punctuation-only text and a near-palindrome with one mismatched letter.

Solved problem 15.4

Create initials for all names except the final surname.

Reasoning and algorithm

Split names into words. Take the first letter of each given name while keeping the last word as the surname.

Program

```
full_name = "Anika Meera Rao"
parts = full_name.split()
initials = ""
for part in parts[:-1]:
    initials += part[0].upper() + ". "
print(initials + parts[-1])
```

Output

A. M. Rao

Why this result follows

Whitespace splitting gives three words. The first two supply initials, and the last word is retained. This version assumes at least one non-whitespace word; a one-word name remains unchanged.

Boundary and error check

Test one-word and multiword names; decide whether extra whitespace is ignored.

Solved problem 15.5

Normalise whitespace into a hyphen-separated label.

Reasoning and algorithm

split without an argument collapses runs of whitespace. Join those words with hyphens, then normalise case.

Program

```
text = "  learn  Python\nwell  "
words = text.split()
label = "-".join(words).lower()
print(label)
print(label.find("python"), label.count("-"))
```

Output

```
learn-python-well
6 2
```

Why this result follows

Whitespace splitting handles repeated spaces and newlines. join introduces exactly one separator between each pair of words.

Boundary and error check

Test leading whitespace, newlines, and an input containing no words.

Trace the string pipeline

Stage	Value	Reason
Original	Spaces and newline	Raw input
split()	[learn, Python, well]	Whitespace removed
join	learn-Python-well	Two separators
lower	learn-python-well	Case normalised

The final find("python") is 6 and count("-") is 2. On all-whitespace input the label is empty.

15. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 15.1

Predict the four slices below. Explain why indexing with `s[8]` behaves differently from `s[8:]`.

```
s = "COMPUTER"  
s[1:6:2]  
s[-3:]  
s[::-2]  
s[8:]
```

Exercise 15.2

Count uppercase letters, lowercase letters, digits and whitespace separately in a string. Leave punctuation outside these counts.

Exercise 15.3

Swap commas and dots in a string by reading each original character once. Explain why two unrestricted chained replace calls can fail.

Exercise 15.4

Read comma-separated names, trim each field and join nonempty names with "; ". Preserve the original order.

Exercise 15.5

Find all starting indexes of "ana" in "banana", including overlaps.

CHAPTER 16

16. Lists

Use mutable ordered collections to collect, edit, search and summarise data. Pay particular attention to shared references and method return values.

16.1 Create and traverse lists

A list holds an ordered sequence of references and can contain mixed types, although uniform data is easier to process. Create one with `[]` or `list(iterable)`. Indexing, negative indices, slicing, concatenation, repetition and membership resemble string operations. A slice produces a new outer list. Assigning items or slices changes a list in place. Traverse values directly; use `range(len(values))` when positions must be updated.

16.2 Add and remove elements

`append(x)` adds one object; `extend(iterable)` adds each of its elements; `insert(i, x)` inserts before position `i`. `remove(x)` deletes the first equal value and raises `ValueError` if missing. `pop(i)` removes and returns an item; `pop()` uses the final position and fails on an empty list. `del` can delete an index or slice, and `clear` removes all elements. Avoid removing elements from the same list while traversing it forwards; build a filtered result instead.

16.3 Search, sort and summarise

`count(x)` counts equal values and `index(x)` returns the first position or raises `ValueError`. `sort` changes the list and returns `None`; `sorted` creates a new sorted list from an iterable. `reverse` changes order in place; it does not sort. `min`, `max` and `sum` work with suitable element types. `min/max` need a nonempty iterable unless a default is supplied. Incomparable mixed values such as integers and strings cannot normally be sorted together.

16.4 Aliasing and copying

`b = a` creates another reference to the same list. `b = a.copy()`, `b = a[:]` and `b = list(a)` create a shallow copy: the outer list is new but nested objects remain shared. Repeating a nested list repeats references, so `[[0] * 3] * 2` creates two references to one row. Build each row separately in a loop when independent rows are required. A shallow copy is not a full independent copy of nested data.

16.5 Nested lists and basic algorithms

A table can be represented as a list of row lists. `table[r][c]` selects a cell. Validate row lengths before assuming a rectangle. For each row, `sum(row)` gives its total; for a column, accumulate `row[column]` across rows. A linear search checks values in order and works without sorting. An introductory manual sort repeatedly selects or swaps values; use `sort` or `sorted` in ordinary programs when the algorithm itself is not the lesson.

16.6 Comprehensions and reversed traversal

A list comprehension is a compact list-building expression: `[x * x for x in range(4)]` creates `[0, 1, 4, 9]`. It has the same basic idea as initialising an empty list and appending a result inside a loop. Learn the loop version first. `reversed(values)` provides a reverse iterator; `list(reversed(values))` materialises a new list. Neither operation means sorting. For nested rows, evaluate a fresh row construction for each outer iteration.

16.7 Search, filtering and list comparison

A linear search can track the first matching index or collect all matching positions. Decide which result the task needs. A filter creates a new list of elements satisfying a condition; initialise that new list once, before traversal. List equality compares corresponding values in order, so `[1, 2]` and `[2, 1]` are unequal even though they contain the same distinct values. Repetition can repeat references to nested objects rather than independently copying those objects. Test with an empty list, one item and repeated values.

Visual walkthrough: Lists

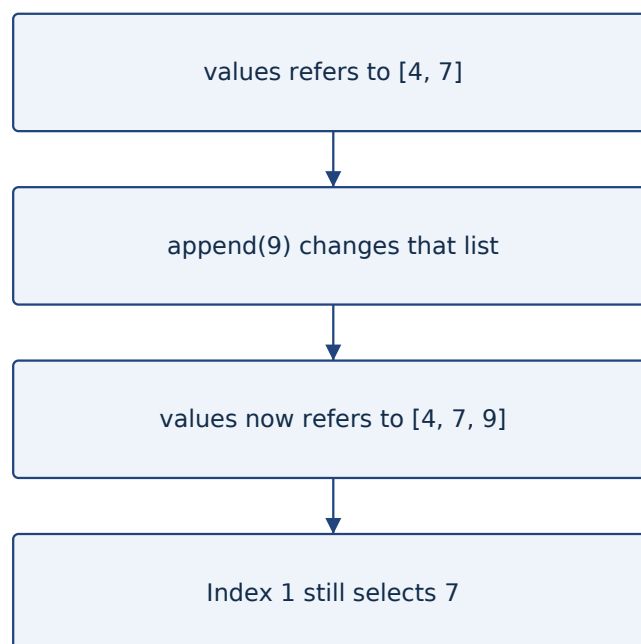


Figure 16.1. Appending increases the list length without shifting existing elements. This follows the first example.

16.8 A first example

```
values = [4, 7]
values.append(9)
print(values)
print(values[1])
```

Output

```
[4, 7, 9]
7
```

`append` changes the existing list by adding one item. Index `1` then selects the second item, `7`.

Quick reference

Operation	Effect / return
<code>append(x)</code> / <code>extend(xs)</code>	Add one object / add all elements
<code>insert(i, x)</code>	Insert before index <code>i</code>
<code>remove(x)</code> / <code>pop(i)</code>	Delete by value / remove and return by index
<code>index(x)</code> / <code>count(x)</code>	First position / number of equal entries
<code>sort()</code> / <code>reverse()</code>	Mutate list; return <code>None</code>
<code>sorted(xs)</code>	Return a new sorted list
<code>min</code> / <code>max</code> / <code>sum</code> / <code>len</code>	Smallest / largest / total / count
<code>copy()</code> / <code>a[:]</code>	Shallow copy of outer list

CHAPTER 16 / WORKED WALKTHROUGH

List names can share the same object

An alias is another name for one list. A shallow copy makes a new outer list but keeps references to the same nested objects. Draw one box per list object and arrows from names or outer-list positions. This prevents surprises when mutating a nested row.

Trace the program

```
a = [[1], [2]]
b = a.copy()
b[0].append(9)
b[1] = [7]
print(a, b)
```

State trace

After copy	Two outer lists; same inner rows
Append 9	Both see first row [1, 9]
Replace b[1]	Only b now points to [7]

A common wrong turn

The result is `[[1, 9], [2]]` and `[[1, 9], [7]]`. `[[0]*3]*3` repeats one row reference; use a comprehension for independent rows.

Check your understanding

Draw the four list boxes and arrows. Then test `a[0]` is `b[0]` and `a` is `b`.

CHAPTER 16 / WORKED WALKTHROUGH

Selection sort: develop the missing algorithm

Maintain a sorted prefix. On outer pass i , find the smallest value in positions i through the end. Swap it into position i . The invariant is that positions before i contain the smallest values in sorted order; the suffix remains to be processed. It takes quadratic comparisons even if the input is nearly sorted.

Trace the program

```
def selection_sort(values):
    a = values.copy()
    for i in range(len(a)):
        smallest = i
        for j in range(i + 1, len(a)):
            if a[j] < a[smallest]:
                smallest = j
        a[i], a[smallest] = a[smallest], a[i]
    return a
```

State trace

Start	[3, 1, 2]
Pass i=0	smallest at 1 -> [1, 3, 2]
Pass i=1	smallest at 2 -> [1, 2, 3]

A common wrong turn

Do not swap during the inner scan: it complicates the invariant. Copying the input makes the function return a new list; remove copy only when in-place mutation is intended.

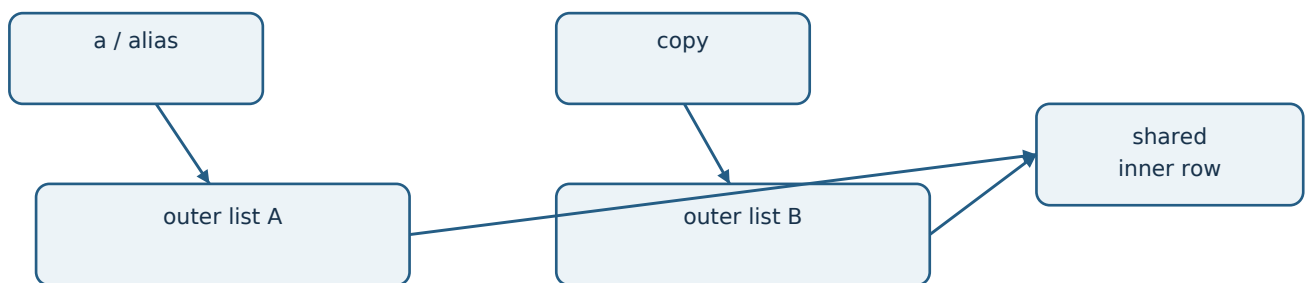
Check your understanding

Trace [], [4], [2, 2, 1], and an already sorted list. Compare with `sorted(values)`.

16. Concept workshop: Aliases, shallow copies and rows

List variables hold references. Assignment creates an alias; `copy` makes a new outer list; nested mutable members may still be shared. Picture two boxes for the outer lists and one shared box for each copied inner row. Replacing a slot in one outer list changes one arrow, while mutating a shared row changes the object both arrows reach. This also explains the multiplication trap: `[[0]*3]*3` repeats one row reference. Use `[[0 for _ in range(3)] for _ in range(3)]` to create independent rows. For selection sort, keep a separate sorted-prefix invariant so the inner search and outer swap have clear roles.

Execution model



Use the model before the code

16.1 `append` inserts one object as a single element, while `extend` iterates its argument and inserts each item.

16.2 Seed largest from a real first element rather than zero.

16.3 Keep a result list and a seen set.

16.4 An alias and the original are one object; a shallow copy is another outer list.

16.5 A table is a list of rows.

Boundary habit

Draw the four list boxes and arrows. Then test `a[0]` is `b[0]` and `a` is `b`.

Challenge the model

16.1 Try extending with an empty list; compare `append([3,4])` with `extend([3,4])`.

16.3 Test empty input, all repeats and mixed types that are hashable.

16.5 Test an empty table and a row missing that column; define whether ragged tables are allowed.

16. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 16.1

Build a list using append and extend.

Reasoning and algorithm

append inserts one object as a single element, while extend iterates its argument and inserts each item. Track list length after each operation.

Program

```
values = [1, 2]
values.append([3, 4])
print(values)
values.extend([5, 6])
print(values)
```

Output

```
[1, 2, [3, 4]]
[1, 2, [3, 4], 5, 6]
```

Why this result follows

append adds the nested list as one element. extend adds two separate integer elements.

Boundary and error check

Try extending with an empty list; compare append([3,4]) with extend([3,4]).

Solved problem 16.2

Find the largest value manually, including all-negative data.

Reasoning and algorithm

Seed largest from a real first element rather than zero. Compare each remaining value and replace the candidate only when greater.

Program

```
values = [-8, -3, -12, -5]
largest = values[0]
for value in values[1:]:
    if value > largest:
        largest = value
print(largest)
```

Output

```
-3
```

Why this result follows

Start largest at -8. Comparing -3 replaces it; -12 and -5 do not. The final largest is -3. This version needs a nonempty list.

Boundary and error check

Test all-negative input and decide what an empty list should do.

Solved problem 16.3

Remove repeated values while preserving first-occurrence order.

Reasoning and algorithm

Keep a result list. For each source value, scan that list; append only when the value is absent. The list retains first-appearance order.

Program

```
values = [4, 2, 4, 1, 2, 3]
unique = []
for value in values:
    if value not in unique:
        unique.append(value)
print(unique)
```

Output

```
[4, 2, 1, 3]
```

Why this result follows

Membership checks prevent duplicate insertion. This simple approach preserves order but repeatedly scans the result.

Boundary and error check

Test empty input, repeats, and inner lists; list membership accepts unhashable values.

Solved problem 16.4

Compare an alias with a shallow copy of a flat list.

Reasoning and algorithm

An alias and the original are one object; a shallow copy is another outer list. Draw arrows before each append or assignment.

Program

```
a = [10, 20]
alias = a
copy = a.copy()
alias.append(30)
copy[0] = 99
print(a)
print(copy)
```

Output

```
[10, 20, 30]
[99, 20]
```

Why this result follows

The alias modifies a itself. The copied outer list can replace its first reference without altering a.

Boundary and error check

For nested rows, test whether the copied outer lists still share inner row objects.

Solved problem 16.5

Calculate row totals and the second-column total of a table.

Reasoning and algorithm

A table is a list of rows. Sum each row; to sum one column, visit that index in every row.

Program

```
table = [[2, 4, 6], [1, 3, 5]]
for row in table:
    print(sum(row))
column_total = 0
for row in table:
    column_total += row[1]
print(column_total)
```

Output

```
12
9
7
```

Why this result follows

Each row is an independent list. The second column has index 1 and contains 4 and 3.

Boundary and error check

Test an empty table and a row missing that column; define whether ragged tables are

Trace the second column

Row	row[1]	Running total
[2, 4, 6]	4	4
[1, 3, 5]	3	7
After loop	No more rows	print(7)

The first loop separately prints row totals 12 and 9. A row shorter than two items would make row[1] fail.

16. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 16.1

Predict the values of `a` and `result` after execution. Explain how `sorted(a)` differs.

```
a = [3, 1, 2]
result = a.sort()
```

Exercise 16.2

Create a new list containing only the non-negative values from a source list. Do not mutate the source during traversal.

Exercise 16.3

Find the second-largest distinct number in a list. Handle fewer than two distinct values with an explanatory message.

Exercise 16.4

Build a 3-by-3 zero table with independent rows, change its centre to 1 and show that no other row changes.

Exercise 16.5

Implement selection sort on a list of integers using loops. Trace the list after each outer pass and compare the result with `sorted`.

CHAPTER 17

17. Tuples

Use immutable sequence structure to represent fixed collections and records. Distinguish an immutable container from the mutability of an object inside it.

17.1 Tuple construction

A tuple is ordered and immutable. Commas create tuple structure: `(7,)` is a one-element tuple whereas `(7)` is just the integer `7`. `()` creates an empty tuple. `tuple(iterable)` constructs a tuple from an iterable. Parentheses improve readability and are necessary in some contexts, but a comma-separated assignment such as `point = 3, 4` also packs a tuple.

17.2 Sequence operations

Tuples support indexing, negative indices, slicing, concatenation, repetition, membership and traversal. These operations do not allow replacement of a tuple item. `count` and `index` are their principal searching methods. `len`, `min`, `max` and `sum` work when values are suitable. `sorted(tuple_value)` returns a list, not a tuple; wrap it in `tuple(...)` when a sorted tuple is required.

17.3 Packing and unpacking

Packing groups values into one tuple. Unpacking assigns corresponding elements to separate targets: `x, y = point`. The number of ordinary targets must match the number of values, otherwise `ValueError` occurs. Starred unpacking, such as `first, *middle, last = values`, collects remaining elements into a list. Treat this as an optional convenience; simple fixed-size unpacking is enough for most introductory records.

17.4 Immutability and reconstruction

You cannot use `t[0] = value`, `append` or `remove` on a tuple. To express an updated collection, construct a new tuple using slices and concatenation, then rebind the name. This is not mutation of the original tuple. A tuple can contain a mutable list; the list's contents can change even though the tuple's reference to that list cannot be replaced.

17.5 Choosing a representation

Use a tuple for a fixed positional record such as `(name, roll_number)`, and a list for a collection that changes length or order. Tuple immutability alone does not guarantee hashability: a tuple containing a list cannot be a dictionary key. A tuple of hashable values, such as an integer coordinate pair, can be a key. For large records with many fields, positional meaning becomes harder to remember; a dictionary can label the fields.

17.6 Nested tuples and extrema positions

A nested tuple stores fixed records or a fixed table. `len` counts top-level items, not all nested elements. For `t = ((2, 4), (6, 8))`, `len(t)` is 2 and `t[1][0]` is 6. To find a largest value and its first position in a flat nonempty tuple, use `largest = max(t)` and `position = t.index(largest)`. A tie returns the first index. Summing a nested tuple directly is not the same as summing each numeric row.

17.7 Combining and rebuilding records

Concatenating two tuples places all items of the first before all items of the second; it does not pair corresponding elements. If a task wants pairs of roll numbers and marks, traverse matching positions and build pairs. An insertion at position `i` can be expressed as `t[:i] + (new_value,) + t[i:]`. A deletion can be expressed as `t[:i] + t[i+1:]`. Always distinguish "construct a changed tuple and rebind the name" from changing a tuple in place.

Visual walkthrough: Tuples

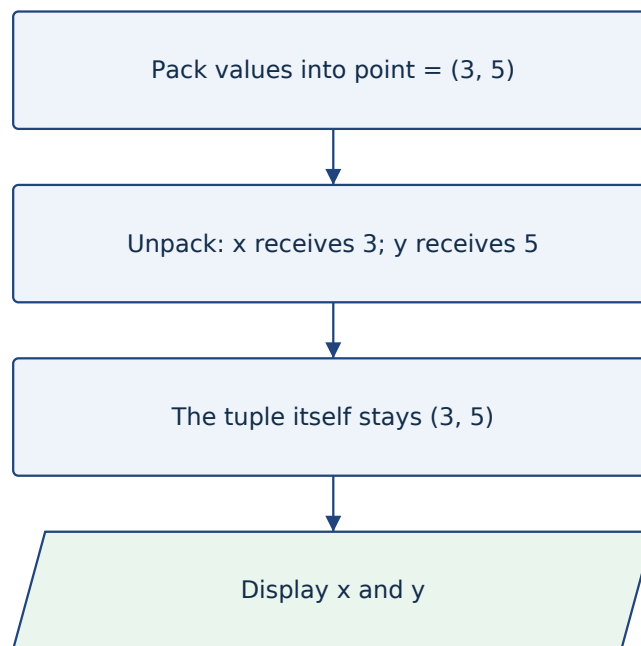


Figure 17.1. Unpacking binds names to the tuple's elements; it does not remove elements from the tuple.

17.8 A first example

```
point = (3, 5)
x, y = point
print(x, y)
```

Output

```
3 5
```

Packing groups two values into a fixed sequence; unpacking binds corresponding names.

Quick reference

Expression	Meaning
(5,) / ()	Singleton / empty tuple
tuple("cat")	("c", "a", "t")
t.count(x) / t.index(x)	Frequency / first position
a, b = (2, 3)	Unpack two values
t[:i] + (x,) + t[i+1:]	Construct a replacement tuple
tuple(sorted(t))	New tuple in sorted order

Concept bridge: Fixed structure and contained mutation

Tuple immutability applies to the tuple's slots. Reconstruct a new tuple when replacing a field; do not attempt an item assignment. If a slot points to a list, methods on that list can still change its contents. Use tuples when field order and count are stable, and unpack into names that explain each position. The comma creates a one-element tuple.

17. Concept workshop: Fixed structure and contained mutation

Tuple immutability applies to the tuple's slots. Reconstruct a new tuple when replacing a field; do not attempt an item assignment. If a slot points to a list, methods on that list can still change its contents. Use tuples when field order and count are stable, and unpack into names that explain each position. The comma creates a one-element tuple. A nested tuple can describe a record or fixed table, but algorithms still need a policy for empty input and for positions that tie.

Execution model



Use the model before the code

17.1 The comma, not parentheses alone, creates a singleton tuple.

17.2 Unpack record fields into meaningful names, then compute the average from numeric fields.

17.3 A tuple slot cannot be assigned.

17.4 Count repeats from the original tuple, then sort into a new sequence and convert back to tuple if that is the required output.

17.5 The tuple holds a reference to an inner mutable list.

Boundary habit

Compare `type((3))` and `type((3,))`; predict which object can be a dictionary key.

Challenge the model

17.1 Also test the empty tuple `()` and a two-item tuple.

17.3 Check first, middle and last replacement positions and whether the original tuple stays unchanged.

17.5 Contrast `append` with direct assignment to `record[1]`, which raises `TypeError`.

17. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 17.1

Create and identify a singleton tuple.

Reasoning and algorithm

The comma, not parentheses alone, creates a singleton tuple. Compare (7,) with (7) by examining their types.

Program

```
single = (7,)
not_tuple = (7)
print(type(single).__name__)
print(type(not_tuple).__name__)
```

Output

```
tuple
int
```

Why this result follows

The comma distinguishes tuple packing from ordinary grouping parentheses. Tuples fix positions, not all contents. A tuple cannot have its items replaced. It can still contain a mutable list whose contents change. Unpack named fields when their meaning is stable.

Boundary and error check

Also test the empty tuple () and a two-item tuple.

Solved problem 17.2

Unpack a student record and calculate an average.

Reasoning and algorithm

Unpack record fields into meaningful names, then compute the average from numeric fields. The fixed field order is part of the record contract.

Program

```
record = ("Mira", 78, 84, 90)
name, first, second, third = record
average = (first + second + third) / 3
print(name, average)
```

Output

```
Mira 84.0
```

Why this result follows

Unpacking makes each fixed position's meaning explicit. There must be exactly four corresponding targets.

Boundary and error check

Test equal marks and a record with the wrong number of fields.

Solved problem 17.3

Create a tuple with a revised middle value.

Reasoning and algorithm

A tuple slot cannot be assigned. Reconstruct a new tuple from the prefix, replacement value and suffix.

Program

```
original = (10, 20, 30)
revised = original[:1] + (99,) + original[2:]
print(original)
print(revised)
```

Output

```
(10, 20, 30)
(10, 99, 30)
```

Why this result follows

Slicing and concatenation construct a new tuple. The singleton replacement needs a comma.

Boundary and error check

Check first, middle and last replacement positions and whether the original tuple stays unchanged.

Solved problem 17.4

Find repeated values and create a sorted tuple.

Reasoning and algorithm

Count repeats from the original tuple, then sort into a new sequence and convert back to tuple if that is the required output.

Program

```
values = (5, 2, 5, 1)
print(values.count(5), values.index(5))
ordered = tuple(sorted(values))
print(ordered)
```

Output

```
2 0
(1, 2, 5, 5)
```

Why this result follows

count includes both occurrences; index returns the first. sorted returns a list, which tuple converts back.

Boundary and error check

Test empty and already sorted tuples; sorted alone returns a list.

Solved problem 17.5

Show the effect of a mutable object inside a tuple.

Reasoning and algorithm

The tuple holds a reference to an inner mutable list. Mutating that list changes what the tuple appears to contain without replacing a tuple slot.

Program

```
record = ("Team A", [10, 12])
record[1].append(14)
print(record)
```

Output

```
('Team A', [10, 12, 14])
```

Why this result follows

The tuple still contains the same two references. The list referred to by its second element changes internally.

Boundary and error check

Contrast append with direct assignment to record[1], which raises TypeError.

Separate tuple slot from list object

Step	Tuple slot 1 refers to	List contents
Construct	One list object	[10, 12]
append(14)	Same list object	[10, 12, 14]
record[1] = []	Forbidden	TypeError

The tuple prevents rebinding its slot; it does not freeze the object stored in that slot.

17. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 17.1

Convert a list of five integers to a tuple and display its first, last and middle elements.

Exercise 17.2

Predict the result and type of each slice.

```
t = (3, 6, 9, 12, 15)
t[1:4]
t[-2:]
t[::-1]
```

Exercise 17.3

Store three (name, mark) records in a tuple. Traverse them using unpacking and display names with marks at least 80.

Exercise 17.4

Construct a new tuple after removing the element at a valid index, using slices. Leave the original unchanged.

Exercise 17.5

Explain whether each can be a dictionary key: (1, 2), ([1], 2), "point", [1, 2]. Connect your explanation to hashability.

CHAPTER 18

18. Dictionaries

Model labelled data and fast lookups using key-value pairs. Build practical frequency tables and learn the behaviour of dictionary methods.

18.1 Keys and values

A dictionary maps unique, hashable keys to values. Strings, integers and tuples of hashable items are common keys; lists and dictionaries are not hashable. Values may be any objects and may repeat. Use `{"name": "Asha", "mark": 88}` or `dict(name="Asha", mark=88)`. If a key appears more than once during construction, the last value wins. Modern Python preserves insertion order, but this order is not automatic sorted order.

18.2 Access and mutation

`d[key]` returns a value or raises `KeyError`. `d.get(key, default)` returns a default when the key is missing without inserting it. Assignment adds a new key or replaces an existing value. `update` merges another mapping or key-value iterable, overwriting matching keys. `setdefault` returns the existing value, or inserts and returns a supplied default for an absent key. Membership tests keys: `key in d` does not search the values.

18.3 Traversal and summaries

Iterating over a dictionary yields keys. `keys()`, `values()` and `items()` provide dynamic views; `items` is convenient for key/value unpacking. Do not change the dictionary's size during iteration over it. `sorted(d)`, `min(d)` and `max(d)` work on keys, provided those keys are comparable. `sum(d)` sums numeric keys; `sum(d.values())` sums numeric values. `len(d)` counts key-value pairs.

18.4 Deletion and copies

`del d[key]` deletes an entry and errors if absent. `pop(key, default)` removes and returns the value, or returns its default; omitting a default raises `KeyError` if absent. `popitem` removes and returns the most recently inserted remaining pair in modern Python, and fails if empty. `clear` empties the mapping. `copy` creates a shallow copy, so nested mutable values remain shared.

18.5 Frequency patterns and fromkeys

A frequency table increments the value associated with each observed item: `counts[item] = counts.get(item, 0) + 1`. Decide whether case and punctuation should be normalised before counting. `dict.fromkeys(keys, value)` assigns the same value object to every key. This is safe for immutable zero counters, but a shared list default can surprise you. Create a fresh list for each key when independent lists are required.

18.6 Renaming keys and nested records

There is no general positional slice of a dictionary. To rename an existing key, retrieve or pop its value and assign it to a new key; decide first what to do if the new key already exists. For a nested employee record, `d["employee"]["salary"]` follows two successive mappings. `enumerate(d)` can number keys during traversal, but `len(d)` is the direct way to count entries. `enumerate` yields pairs of a counter and an item, starting at zero unless given a start.

18.7 Merging, ranking and totals

`update` overwrites matching keys. If the task instead asks to combine quantities, iterate through the second dictionary and add each amount to `result.get(key, 0)`. To obtain the two highest distinct values, first remove duplicate values and then sort the remaining values; check that two distinct values exist. To retain all names tied at the maximum, find the maximum score and then collect matching keys. A total over keys and a total over values answer different questions. State which one is requested before using `sum`.

Visual walkthrough: Dictionaries

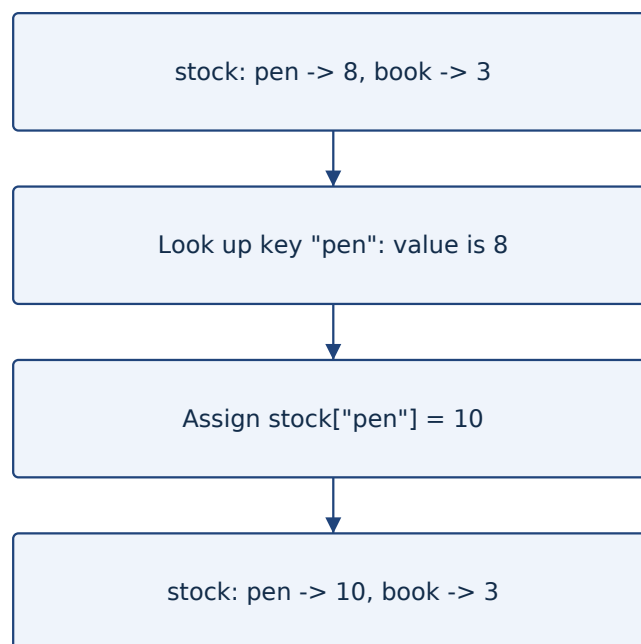


Figure 18.1. A key identifies the entry to update. Other key-value pairs are retained.

18.8 A first example

```
stock = {"pen": 8, "book": 3}
print(stock["pen"])
stock["pen"] = 10
print(stock)
```

Output

```
8
{'pen': 10, 'book': 3}
```

The string key `pen` identifies the quantity. Updating its value retains the same key.

Quick reference

Operation	Behaviour
<code>d[k] / d.get(k, default)</code>	Error if absent / return default if absent
<code>d[k] = v / d.update(other)</code>	Insert or replace / merge
<code>keys() / values() / items()</code>	Dynamic views of keys / values / pairs
<code>setdefault(k, v)</code>	Get value, inserting <code>v</code> only if absent
<code>pop(k) / popitem()</code>	Remove one key / latest inserted pair
<code>clear() / copy()</code>	Empty dictionary / shallow copy
<code>dict.fromkeys(keys, value)</code>	All keys refer to the same value object
<code>sorted(d) / sum(d.values())</code>	Sorted keys / total numeric values

Concept bridge: Keys, counts and independent records

A dictionary answers a key lookup, while a list answers a positional lookup. For frequency counting, after processing `k` characters, the dictionary must contain exactly their accepted counts. `get(key, 0)` supplies the missing starting count, but it does not create a key until assigned.

18. Concept workshop: Keys, counts and independent records

A dictionary answers a key lookup, while a list answers a positional lookup. For frequency counting, after processing k characters, the dictionary must contain exactly their accepted counts. `get(key, 0)` supplies the missing starting count, but it does not create a key until assigned. When values are mutable lists, two keys may accidentally share one list object; construct a separate value for each group. To find a maximum, scan actual entries and state how ties and empty input are handled.

Execution model



Use the model before the code

- 18.1 Normalise each character, skip spaces, and increment its dictionary count.
- 18.2 Dictionary assignment replaces the value for an existing key and creates a key when absent.
- 18.3 Seed the best student from a real item and scan the rest.
- 18.4 Use `pop` with a default when the stock item may be missing, then sum the remaining values.
- 18.5 Construct one list for each group rather than sharing a single mutable default or alias.

Boundary habit

Try the empty string, then compare membership testing with `get` for a key whose stored value is 0.

Challenge the model

- 18.1 Test empty text, mixed case, and repeated spaces.
- 18.3 Test all-negative marks, ties and empty input; define a tie policy.
- 18.5 Append to one group and check that the other group does not change.

18. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 18.1

Count letter frequencies, ignoring spaces and case.

Reasoning and algorithm

Lowercase each character, then accept letters with `isalpha()`. This also skips digits and punctuation, not just spaces.

Program

```
counts = {}
for character in "Data Data".lower():
    if character.isalpha():
        counts[character] = counts.get(character, 0) + 1
print(counts)
```

Output

```
{'d': 2, 'a': 4, 't': 2}
```

Why this result follows

For the first distinct letter, get supplies zero before adding one. Later occurrences increase its count. Keys retain first-appearance order.

Boundary and error check

Test empty text, mixed case, and repeated spaces.

Solved problem 18.2

Update a student mark and add a new student.

Reasoning and algorithm

Dictionary assignment replaces the value for an existing key and creates a key when absent. Model the two operations separately.

Program

```
marks = {"Asha": 82, "Dev": 75}
marks.update({"Dev": 79, "Mira": 91})
for name, mark in marks.items():
    print(name, mark)
```

Output

```
Asha 82
Dev 79
Mira 91
```

Why this result follows

Dev's value is replaced without making a duplicate key. Mira is appended as a newly inserted key.

Boundary and error check

Check len before and after each update; replacing should not change the number of keys.

Solved problem 18.3

Find the highest-scoring student without sorting.

Reasoning and algorithm

Seed the best student from a real item and scan the rest. Compare marks rather than sorting the whole dictionary.

Program

```
marks = {"Asha": 82, "Dev": 79, "Mira": 91}
best_name = None
best_mark = None
for name, mark in marks.items():
    if best_mark is None or mark > best_mark:
        best_name, best_mark = name, mark
print(best_name, best_mark)
```

Output

```
Mira 91
```

Why this result follows

The None sentinel avoids assuming scores are positive. This version keeps the first encountered name in a tie and assumes nonempty data.

Boundary and error check

Test all-negative marks, ties and empty input; define a tie policy.

Solved problem 18.4

Remove a stock item safely and total the remaining quantities.

Reasoning and algorithm

Use pop with a default when the stock item may be missing, then sum the remaining values. Deletion should not crash on an absent key.

Program

```
stock = {"pen": 12, "book": 5, "eraser": 8}
removed = stock.pop("ruler", 0)
stock.pop("eraser")
print(removed, sum(stock.values()))
```

Output

```
0 17
```

Why this result follows

The default prevents a missing ruler from causing `KeyError`. Summing values totals quantities rather than keys.

Boundary and error check

Test removal of a present key and a missing key.

Solved problem 18.5

Create independent lists for two groups.

Reasoning and algorithm

Construct a new empty list in each loop iteration. The red and blue keys then refer to different list objects.

Program

```
groups = {}
for name in ["red", "blue"]:
    groups[name] = []
groups["red"].append("Asha")
print(groups)
```

Output

```
{'red': ['Asha'], 'blue': []}
```

Why this result follows

Each evaluation of `[]` creates a fresh list. `fromkeys` with one list argument would share that list across both keys.

Boundary and error check

Append to one group and check that the other group does not change.

Trace independent lists

Step	groups["red"]	groups["blue"]
Create red	[]	Absent
Create blue	[]	[] (different list)
Append to red	["Asha"]	[]

Using `dict.fromkeys(["red", "blue"], [])` would reuse one list and change both entries when either is appended.

18. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 18.1

Evaluate these operations in order. Explain which operation changes the dictionary.

```
d = {"a": 1}
d.get("b", 0)
d.setdefault("b", 0)
"b" in d
```

Exercise 18.2

Build a dictionary mapping integers 1 through 10 to their squares, then display the pairs in descending key order.

Exercise 18.3

Merge two dictionaries of item quantities by adding quantities for common keys. Do not merely overwrite them.

Exercise 18.4

Given a nonempty name-to-score dictionary, print every name tied for the highest score, in alphabetic order.

Exercise 18.5

Invert a dictionary whose values may repeat, producing a dictionary of value-to-list-of-keys. Keep independent lists for each value.

CHAPTER 19

19. Sets

A set stores unique hashable elements. It is useful when membership and overlap matter more than positions or repetition.

19.1 Unique elements

Create a set with braces containing values, such as `{2, 4, 6}`. Repeated equal values collapse to one element. `set()` creates an empty set; `{}` instead creates an empty dictionary. A set does not provide sequence positions, so `s[0]` is invalid. Never rely on the printed or traversal order as a meaningful sorting of the elements.

19.2 Hashable members

Elements must be hashable: integers, strings and tuples of hashable values are common examples. A list cannot be an element of a set. `"Immutable"` is a useful beginner clue but not a complete test: a tuple containing a list is not hashable. Membership with `in` asks whether an equal element is present. `len` gives the number of distinct stored elements.

19.3 Union and intersection

The union `a | b` contains elements found in either set; the intersection `a & b` contains elements present in both. Difference `a - b` keeps elements of `a` absent from `b`. Symmetric difference `a ^ b` keeps elements present in exactly one set. These symbols also have bitwise meanings for integers; the operand types determine which operation is performed.

19.4 Mutation

`add(x)` inserts one element. `remove(x)` deletes an element and raises `KeyError` when it is missing; `discard(x)` removes it if present and otherwise does nothing. `clear` empties the set. These methods modify the set and should not be assigned back as if they returned an updated collection. Avoid adding or removing elements while directly iterating over the same set.

19.5 When to use a set

Sets are useful for counting distinct names, checking shared interests or finding missing registrations. They are unsuitable for remembering how many times each value occurred; use a frequency dictionary for that. Converting a list to a set discards duplicates and list order. `sorted(a_set)` can produce a predictable list for presentation when its elements are mutually comparable.

Visual walkthrough: Sets

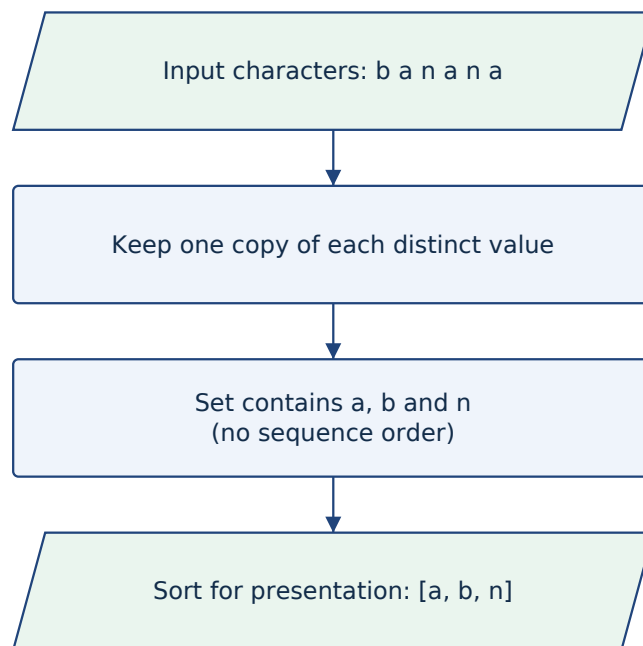


Figure 19.1. A set removes repeated values. `sorted` provides a separate predictable presentation order.

19.6 A first example

```
letters = set("banana")
print(sorted(letters))
```

Output

```
['a', 'b', 'n']
```

The six-character string becomes a set of three distinct characters. Sorting is only for presentation.

CHAPTER 19 / WORKED WALKTHROUGH

Sets answer membership questions

A set stores each distinct hashable value once. Use it when you care whether something occurs, not how many times or at which index. Set difference models missing items: expected - submitted. Convert to sorted output when a stable display order is required.

Trace the program

```
expected = {1, 2, 3}
submitted = {2, 2, 3}
print(sorted(expected - submitted))
```

State trace

submitted set	{2, 3}
difference	{1}
sorted display	[1]

A common wrong turn

Sets are unordered; do not index one or promise its print order. For counts, use a dictionary instead.

Check your understanding

Try no submissions and an extra ID 4. Distinguish missing from unexpected IDs.

19. Concept workshop: Set algebra as a question

Translate wording into membership operations: both groups means intersection, either means union, expected but absent means difference, and exactly one means symmetric difference. A set discards duplicates and does not promise display order. If frequency or first position matters, a set alone loses necessary information. For a reproducible printed answer, sort the resulting set at the display boundary. Always check the empty-set cases; they often make the meaning of the operation clearest.

Execution model



Use the model before the code

19.1 Convert values to a set because only uniqueness matters; len then gives the number of distinct values.

19.2 Intersection keeps values present in both registration sets.

19.3 Union collects values present in either group without duplicates.

19.4 Expected minus submitted yields missing identifiers.

19.5 discard safely ignores a missing element; remove would raise KeyError.

Boundary habit

Try no submissions and an extra ID 4. Distinguish missing from unexpected IDs.

Challenge the model

19.1 Test an empty input and repeated copies of one value.

19.3 Test an empty group and a student registered in both.

19.5 Test present and absent values and verify the resulting set.

19. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 19.1

Count distinct values.

Reasoning and algorithm

Convert values to a set because only uniqueness matters; len then gives the number of distinct values.

Program

```
values = [3, 1, 3, 2, 1]
unique = set(values)
print(len(unique))
print(sorted(unique))
```

Output

```
3
[1, 2, 3]
```

Why this result follows

The set stores 1, 2 and 3 once each. sorted returns a list for stable output.

Boundary and error check

Test an empty input and repeated copies of one value.

Solved problem 19.2

Find common course registrations.

Reasoning and algorithm

Intersection keeps values present in both registration sets. State whether order matters before printing.

Program

```
python = {"Asha", "Dev", "Mira"}
maths = {"Dev", "Omar"}
print(sorted(python & maths))
```

Output

```
['Dev']
```

Why this result follows

Intersection retains only names in both sets.

Boundary and error check

Test disjoint and identical groups.

Solved problem 19.3

Find names registered in either course.

Reasoning and algorithm

Union collects values present in either group without duplicates. Sort only if a stable printed order is required.

Program

```
first = {"Asha", "Dev"}
second = {"Dev", "Mira"}
print(sorted(first | second))
```

Output

```
['Asha', 'Dev', 'Mira']
```

Why this result follows

Union includes every distinct name once.

Boundary and error check

Test an empty group and a student registered in both.

Solved problem 19.4

Find missing submissions.

Reasoning and algorithm

Expected minus submitted yields missing identifiers. Reverse the subtraction to find unexpected submissions instead.

Program

```
expected = {1, 2, 3, 4}
submitted = {1, 3}
print(sorted(expected - submitted))
```

Output

```
[2, 4]
```

Why this result follows

Difference is directional: expected minus submitted finds missing entries.

Boundary and error check

Test complete, empty and extra submissions.

Solved problem 19.5

Remove a possibly absent value safely.

Reasoning and algorithm

`discard` safely ignores a missing element; `remove` would raise `KeyError`. Choose based on whether absence is normal.

Program

```
values = {2, 4}
values.discard(9)
values.add(6)
print(sorted(values))
```

Output

```
[2, 4, 6]
```

Why this result follows

`discard` does not fail when 9 is absent. `add` then inserts 6.

Boundary and error check

Test present and absent values and verify the resulting set.

Trace an absent removal

Step	Set before	Set after
<code>discard(9)</code>	{2, 4}	{2, 4}
<code>add(6)</code>	{2, 4}	{2, 4, 6}
<code>sorted(values)</code>	{2, 4, 6}	[2, 4, 6]

`remove(9)` would raise `KeyError`; `discard(9)` explicitly accepts the missing value.

19. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 19.1

Explain why `{}` and `set()` produce different empty collection types.

Exercise 19.2

Predict union, intersection, each directional difference and symmetric difference.

```
a = {1, 2, 3}
b = {3, 4}
a | b
a & b
a - b
b - a
a ^ b
```

Exercise 19.3

Count the distinct letters in a lowercase word, ignoring repeated occurrences. Print them in sorted order.

Exercise 19.4

Explain why converting a list of marks into a set before calculating an average can change the meaning of the result.

Exercise 19.5

Find students who joined exactly one of two clubs. Use a set operation and test overlapping as well as disjoint groups.

CHAPTER 20

20. Modules

A module is a reusable collection of definitions. Import it once and use its documented names instead of rewriting the same tools.

20.1 Organisation and reuse

A module usually corresponds to a Python file, though some modules are implemented differently. It groups related names and code. The standard library contains modules supplied with Python, including `math`, `random` and `statistics`. A built-in function such as `len` is available without an import; a standard-library function may need one. A module is not itself just another word for a single function.

20.2 The import statement

`import math` binds the module name `math` in your program. Access one of its attributes with a dot, such as `math.sqrt(16)`. The prefix identifies where the function comes from. A module may expose constants as well as functions: `math.pi` is a value and therefore does not require call parentheses. Writing `math.pi()` would attempt to call a float and fail.

20.3 Selective imports

`from math import sqrt` binds `sqrt` directly. You then call `sqrt(16)`, not necessarily `math.sqrt(16)`, because the module name `math` has not been introduced by that statement alone. You may import several named items together. Avoid wildcard imports while learning: they make it difficult to see where names originated and may overwrite names already in use.

20.4 Aliases

`import statistics as stats` creates a shorter local name for the module. It does not rename the installed module on disk. `from math import sqrt as root` creates a local alias for the selected function. Choose aliases that improve readability. If an example uses an alias, use that same name in later calls. An import should normally appear near the start of the file.

20.5 Finding and diagnosing names

A missing module can raise `ModuleNotFoundError`; asking for a nonexistent attribute can raise `AttributeError`. Forgetting to import a name before use usually raises `NameError`. Do not call a file `math.py` or `statistics.py` when you intend to import the standard modules. `help(math.sqrt)` describes a function after import. For unfamiliar behaviour, consult the official Python documentation rather than guessing from the name.

Visual walkthrough: Modules

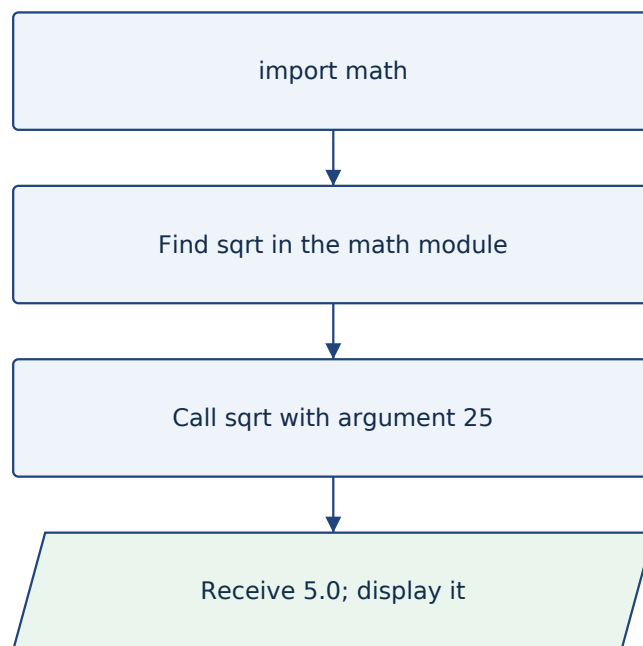


Figure 20.1. The dot selects the function in the imported module. Parentheses call that function.

20.6 A first example

```
import math
print(math.sqrt(25))
```

Output

```
5.0
```

The dot is attribute access: find `sqrt` inside the `math` module, then call it with the argument `25`.

Concept bridge: The import determines the legal name

Python executes an import statement and binds one or more names. `import math` gives a module name from which `math.sqrt` can be looked up. `from math import sqrt` binds `sqrt` directly. `import math as m` binds `m`, so `math.sqrt` is unavailable unless `math` was separately imported.

20. Concept workshop: The import determines the legal name

Python executes an import statement and binds one or more names. `import math` gives a module name from which `math.sqrt` can be looked up. `from math import sqrt` binds `sqrt` directly. `import math as m` binds `m`, so `math.sqrt` is unavailable unless `math` was separately imported. A constant such as `math.pi` is retrieved, whereas a function such as `math.sqrt` must be called. Trace the namespace before debugging an `AttributeError` or `NameError`; an incorrect call style is often the entire problem.

Execution model



Use the model before the code

20.1 `import math` binds the module name; attribute lookup then finds `sqrt`.

20.2 `from math import sqrt` binds the function directly.

20.3 An alias binds another module name.

20.4 A module constant is a value, not a function.

20.5 Match call syntax to the import statement: either `module.function` after `import module`, or `function` after selective import.

Boundary habit

Rewrite using `from math import sqrt` and explain why `math.sqrt` would then need a separate import.

Challenge the model

20.1 Call with zero and a perfect square; a negative real input is outside `sqrt`'s real domain.

20.3 Rename the alias and locate all dependent calls.

20.5 Run both valid versions and check that the computed value agrees.

20. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 20.1

Use a module-qualified function.

Reasoning and algorithm

`import math` binds the module name; attribute lookup then finds `sqrt`. Keep the qualifier in the call.

Program

```
import math
print(math.sqrt(49))
```

Output

```
7.0
```

Why this result follows

The import supplies the module name; the dot selects its `sqrt` function. Imports create names in a namespace. An import is a naming decision. `import math` binds `math`, so call `math.sqrt`. `from math import sqrt` binds `sqrt` directly. Both use the same library function.

Boundary and error check

Call with zero and a perfect square; a negative real input is outside `sqrt`'s real domain.

Solved problem 20.2

Use a selectively imported function.

Reasoning and algorithm

`from math import sqrt` binds the function directly. Do not prefix it with `math` unless `math` was separately imported.

Program

```
from math import ceil
print(ceil(4.2))
```

Output

```
5
```

Why this result follows

Only `ceil` is bound by this import statement. Calling it does not require a `math` prefix.

Boundary and error check

Compare the namespace after both import styles.

Solved problem 20.3

Use a module alias.

Reasoning and algorithm

An alias binds another module name. Use the chosen alias consistently at every call site.

Program

```
import statistics as stats
print(stats.mean([2, 4, 9]))
```

Output

```
5
```

Why this result follows

The alias stats identifies the statistics module within this program.

Boundary and error check

Rename the alias and locate all dependent calls.

Solved problem 20.4

Access a constant rather than calling it.

Reasoning and algorithm

A module constant is a value, not a function. Read math.pi without parentheses and use it inside a calculation.

Program

```
import math
print(round(math.pi, 3))
```

Output

```
3.142
```

Why this result follows

pi is a number. round is a built-in function used here to shorten its displayed numeric value.

Boundary and error check

Check its type and compare it with a function such as math.sqrt.

Solved problem 20.5

Repair a mismatch between import style and call style.

Reasoning and algorithm

Match call syntax to the import statement: either `module.function` after `import module`, or `function` after selective import.

Program

```
from math import sqrt as root
print(root(81))
```

Output

```
9.0
```

Why this result follows

After the chosen import, the local function name is `root`. A bare call to `sqrt` would not be defined by this statement.

Boundary and error check

Run both valid versions and check that the computed value agrees.

Resolve the imported name

Step	Available name	Call result
Import	<code>root</code>	Alias for <code>math.sqrt</code>
<code>root(81)</code>	<code>root</code>	<code>9.0</code>
<code>sqrt(81)</code>	Not imported here	<code>NameError</code>

Alternative: `import math` followed by `math.sqrt(81)`. Match the call to the names introduced by the import.

20. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 20.1

Distinguish a module, a function and a constant. Give one example of each from math.

Exercise 20.2

Rewrite `import math; print(math.sqrt(64))` using a selective import.

Exercise 20.3

Import statistics with a readable alias and calculate the mean of three values.

Exercise 20.4

Explain the error when `sqrt` has no other binding. Repair the call or the import.

```
import math
print(sqrt(16))
```

Exercise 20.5

Explain why saving your program as `random.py` can interfere with `import random`. Choose a safer filename.

CHAPTER 21

21. The math module

Use the standard mathematical library for real-number calculations, rounding, powers and angles.

21.1 Real-valued mathematics

Import `math` before using its functions. `sqrt(x)` returns a real square root and requires `x` to be non-negative for ordinary real input. `log(x)` requires a positive argument. A domain error means the requested real operation is undefined for the supplied value. Complex-valued mathematics uses other tools and is outside this chapter's scope.

21.2 Rounding directions

`floor(x)` is the greatest integer no larger than `x`; `ceil(x)` is the smallest integer no smaller than `x`. For `-2.7` they return `-3` and `-2` respectively. `int(-2.7)` truncates towards zero and returns `-2`. `round` is a built-in, not a `math`-only function; for exact halfway cases it uses ties-to-even behaviour, subject to representation details for floats. Formatting to two decimal places is usually clearer when the task is only display.

21.3 Powers and factorials

The `**` operator calculates powers; `math.pow` converts its inputs to floats and returns a float. `factorial(n)` multiplies the positive integers through `n`, with `factorial(0)` defined as 1. It expects a non-negative integer. `fabs` gives a floating-point absolute value. The built-in `abs` also works for ordinary real numbers and has other numeric uses. Select the function by its specification, not by a familiar-sounding name.

21.4 Constants and angles

`math.pi` approximates the circle constant and `math.e` approximates the natural exponential base. `sin`, `cos` and `tan` expect radians, not degrees. `radians(degrees_value)` converts degrees to radians; `degrees(radians_value)` goes the other way. For example, a right angle is `pi/2` radians. Never infer the required unit only from a number such as 30; document its meaning.

21.5 Logarithms and floating results

`log(x)` is the natural logarithm; `log10(x)` is base ten. A logarithm is the power to which a base must be raised to obtain a positive value. Floating-point trigonometric results may display long approximations such as `0.4999999999999994`. This does not mean the algorithm is fundamentally wrong. Round or format for presentation, and use a tolerance when comparing approximate real results.

Visual walkthrough: The math module

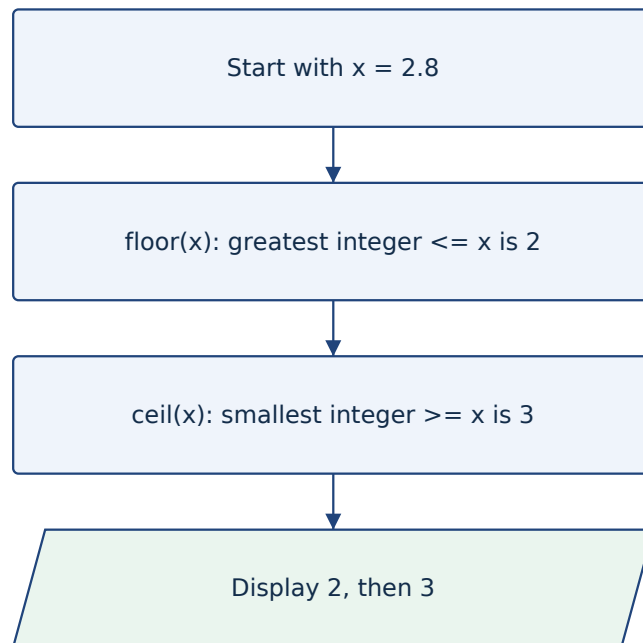


Figure 21.1. Compare rounding down with rounding up. Neither operation asks for the nearest integer.

21.6 A first example

```
import math
print(math.floor(2.8))
print(math.ceil(2.8))
```

Output

```
2
3
```

The floor is 2 and the ceiling is 3; neither function means "round to the nearest integer".

CHAPTER 21 / WORKED WALKTHROUGH

Distinguish angle units and rounding

The math trigonometric functions take radians, while many word problems give degrees. Convert at the boundary. `ceil` and `floor` round toward positive and negative infinity respectively; truncation toward zero is different for negative values.

Trace the program

```
import math
angle = math.radians(90)
print(round(math.sin(angle), 6))
```

State trace

90 degrees	pi/2 radians
<code>sin(pi/2)</code>	approximately 1.0
rounded display	1.0

A common wrong turn

Floating-point outputs near a mathematical integer can differ in the last bits. Use a tolerance when comparing calculated real values.

Check your understanding

Compare `floor(-2.8)`, `ceil(-2.8)`, and `trunc(-2.8)` by drawing them on a number line.

21. Concept workshop: Domains and units in math

The mathematics formula and the library call both carry assumptions. A right triangle has non-negative lengths; factorial accepts non-negative integers; logarithm requires a positive argument; trigonometric calls accept radians. Convert degrees before calling sin or cos. floor, ceil and trunc differ most clearly for negative fractions. Floating-point computations can be close to a mathematical value without having identical binary representation, so compare calculated real values within an appropriate tolerance.

Execution model



Use the model before the code

21.1 The legs form a right triangle, so hypotenuse is the square root of the sum of squares.

21.2 floor goes toward negative infinity, ceil toward positive infinity, and trunc toward zero.

21.3 Square the radius before multiplying by pi; keep the geometry formula distinct from display rounding.

21.4 math.sin expects radians.

21.5 Use factorial for nonnegative integers, pow for exponentiation and log for positive inputs.

Boundary habit

Compare floor(-2.8), ceil(-2.8), and trunc(-2.8) by drawing them on a number line.

Challenge the model

21.1 Test a 3-4-5 triangle and a zero leg.

21.3 Test radius zero and reject a negative radius if it violates the task contract.

21.5 Try factorial(0), pow(2,3), log(1); document invalid negative or zero-domain calls.

21. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 21.1

Find the length of a right triangle's hypotenuse.

Reasoning and algorithm

The legs form a right triangle, so hypotenuse is the square root of the sum of squares. `math.hypot` expresses that operation directly.

Program

```
import math
a, b = 5, 12
print(math.hypot(a*a + b*b))
```

Output

```
13.0
```

Why this result follows

The squared sides sum to 169; the square root is 13.

Boundary and error check

Test a 3-4-5 triangle and a zero leg.

Solved problem 21.2

Compare floor, ceil and truncation.

Reasoning and algorithm

floor goes toward negative infinity, ceil toward positive infinity, and trunc toward zero. Negative fractions separate the three.

Program

```
import math
x = -3.6
print(math.floor(x), math.ceil(x), int(x))
```

Output

```
-4 -3 -3
```

Why this result follows

Floor moves towards negative infinity, ceil towards positive infinity and int towards zero.

Boundary and error check

Compare -2.8 and +2.8 rather than using only positive data.

Solved problem 21.3

Compute a circle's area.

Reasoning and algorithm

Square the radius before multiplying by pi; keep the geometry formula distinct from display rounding.

Program

```
import math
radius = 4
area = math.pi * radius ** 2
print(f"{area:.2f}")
```

Output

```
50.27
```

Why this result follows

Use pi as a constant and square the radius. .2f controls only the displayed precision.

Boundary and error check

Test radius zero and reject a negative radius if it violates the task contract.

Solved problem 21.4

Calculate a sine from a degree angle.

Reasoning and algorithm

math.sin expects radians. Convert the degree angle with radians before calling sin.

Program

```
import math
angle = math.radians(30)
print(f"{math.sin(angle):.3f}")
```

Output

```
0.500
```

Why this result follows

The explicit conversion makes the units correct before the trigonometric call.

Boundary and error check

Test 0 and 90 degrees, allowing tiny floating-point error.

Solved problem 21.5

Compare factorial, power and logarithm.

Reasoning and algorithm

Use factorial for nonnegative integers, pow for exponentiation and log for positive inputs. Each function has a domain.

Program

```
import math
print(math.factorial(5))
print(math.pow(2, 4))
print(math.log10(1000))
```

Output

```
120
16.0
3.0
```

Why this result follows

These are different operations: product through 5, 2 raised to 4, and the power of 10 needed to obtain 1000.

Boundary and error check

Try factorial(0), pow(2,3), log(1); document invalid negative or zero-domain calls.

Trace operation and domain

Expression	Question answered	Result
factorial(5)	1 * 2 * 3 * 4 * 5	120
pow(2, 4)	2 raised to 4	16.0
log10(1000)	10 to what power?	3.0
factorial(0)	Empty product	1

For log10, the input must be positive. math.pow returns a float here, even though the power is mathematically integral.

21. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 21.1

Predict `floor(2.1)`, `ceil(2.1)`, `floor(-2.1)` and `ceil(-2.1)`. Draw their positions on a number line.

Exercise 21.2

Use `math.pi` to calculate a circle circumference for a supplied non-negative radius.

Exercise 21.3

Calculate `sin(90 degrees)` after converting the angle. Explain why `sin(90)` alone is a different request.

Exercise 21.4

Write code that rejects a negative number before calling `math.sqrt`. Test zero as well as a positive number.

Exercise 21.5

Use `factorial` to calculate the number of ways to arrange five distinct objects in a line. Explain why `factorial(0)` is 1.

CHAPTER 22

22. The random module

Generate varied sample inputs and simple simulations. Focus on possible outcomes and endpoints rather than predicting a random result.

22.1 Pseudorandomness

A pseudorandom generator uses an algorithm and internal state to produce a sequence that is useful for simulations. It is not a source of cryptographic secrets. `import random` makes the standard module available. A seed can repeat an experiment; do not reseed before every draw or you may repeat the same part of the sequence instead of generating varied trials.

22.2 Float and integer ranges

`random.random()` returns a float at least `0.0` and less than `1.0`. `random.randint(a, b)` chooses an integer with both endpoints included. `random.randrange(start, stop, step)` follows range-style endpoint rules, excluding `stop`. For example, `randrange(2, 11, 2)` can return `2`, `4`, `6`, `8` or `10`. An empty range cannot produce a choice.

22.3 Choosing and rearranging

`choice(sequence)` selects one element from a nonempty sequence. `sample(sequence, k)` selects `k` positions without replacement and returns a list. If the source itself contains duplicate values, equal values can still appear in a sample. `shuffle(list_value)` changes the list order in place and returns `None`. If you want to preserve the original list, copy it first.

22.4 Simulation and expectations

To simulate a die, use `randint(1, 6)`. Repeat the draw and count outcomes. The expected counts are not guaranteed counts: even a fair die need not give exactly `100` of each face in `600` rolls. Tests should verify properties such as range and number of draws. Avoid claiming a single simulation proves fairness or predicts the next result.

22.5 Reproducible examples

`random.Random(seed)` creates a dedicated generator object with its own state. Calling `rng.randint(...)` then uses that generator. This book prints stable checks such as the length of a result and whether every outcome lies in range. Exact high-level sequences can depend on Python version. Four-digit classroom practice codes can be generated here, but real authentication codes require a security-focused generator such as secrets.

Visual walkthrough: The random module

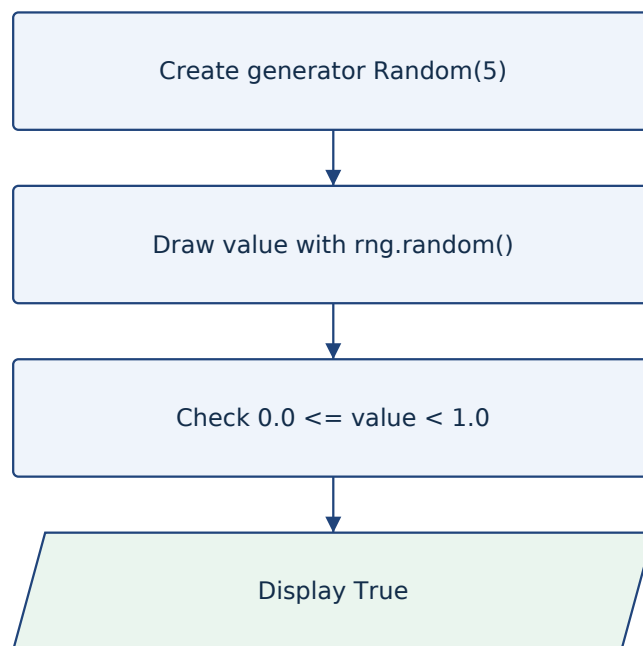


Figure 22.1. A property check verifies the documented interval without depending on a particular decimal draw.

22.6 A first example

```
import random
rng = random.Random(5)
value = rng.random()
print(0.0 <= value < 1.0)
```

Output

```
True
```

The result must satisfy the documented interval. A property check is more useful here than memorising a random decimal.

CHAPTER 22 / WORKED WALKTHROUGH

Specify what randomness means

A pseudorandom generator produces reproducible draws from a seed. `randint(a, b)` includes both endpoints. `choice` selects one element; `shuffle` changes a list in place and returns `None`. A seeded generator is useful for tests, not for security-sensitive secrets.

Trace the program

```
import random
rng = random.Random(5)
values = [1, 2, 3]
rng.shuffle(values)
print(sorted(values))
```

State trace

Before	Three distinct elements
After shuffle	Order may change
After sorted	[1, 2, 3]

A common wrong turn

Do not test an unseeded program against one exact draw. Test invariants such as range, length and preservation of elements.

Check your understanding

For 100 coin tosses, check that heads + tails = 100 regardless of the actual draws.

22. Concept workshop: Test invariants of random code

Pseudorandom output is an outcome of a generator state, not a promise of one particular value. A seed makes a local sequence reproducible; it does not make it secure. Assert properties that hold for every valid draw: a die is within 1..6, a sample contains distinct source members, and a shuffle preserves the multiset. Understand whether an endpoint is included and whether a function mutates its input. Keeping a source copy separate from a shuffled list makes both behavior and tests easier to explain.

Execution model



Use the model before the code

22.1 `randint(1,6)` includes both endpoints, making it a direct die model.

22.2 `range` in `randrange` excludes its stop.

22.3 `sample` selects distinct positions without replacement.

22.4 `shuffle` mutates its list and returns `None`.

22.5 Choose each character from the stated alphabet, repeat exactly four times, and join.

Boundary habit

For 100 coin tosses, check that `heads + tails = 100` regardless of the actual draws.

Challenge the model

22.1 Assert `1 <= roll <= 6` across draws; do not demand one particular unseeded roll.

22.3 Check that the result has length three and each name comes from the source.

22.5 Check length and membership of every character; use secrets for real security tokens.

22. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 22.1

Generate a die result and verify its bounds.

Reasoning and algorithm

`randint(1,6)` includes both endpoints, making it a direct die model. A random result is not one fixed expected value.

Program

```
import random
rng = random.Random(17)
roll = rng.randint(1, 6)
print(1 <= roll <= 6)
```

Output

```
True
```

Why this result follows

The output checks the allowed range without depending on one particular random face.

Boundary and error check

Assert $1 \leq \text{roll} \leq 6$ across draws; do not demand one particular unseeded roll.

Solved problem 22.2

Generate an even value with an excluded stop.

Reasoning and algorithm

`range` in `randrange` excludes its stop. Choose a start and step that generate exactly the intended even values.

Program

```
import random
rng = random.Random(9)
value = rng.randrange(2, 12, 2)
print(value in [2, 4, 6, 8, 10])
```

Output

```
True
```

Why this result follows

The possible values are the elements of `range(2, 12, 2)`; 12 itself cannot be chosen.

Boundary and error check

Enumerate all possible values and ensure the upper stop is excluded.

Solved problem 22.3

Select three different names from a unique list.

Reasoning and algorithm

sample selects distinct positions without replacement. Require a source with at least three distinct names for the intended result.

Program

```
import random
rng = random.Random(11)
names = ["Asha", "Dev", "Mira", "Omar"]
chosen = rng.sample(names, 3)
print(len(chosen), len(set(chosen)))
```

Output

```
3 3
```

Why this result follows

sample chooses without replacement. Unique source names ensure three distinct selected names.

Boundary and error check

Check that the result has length three and each name comes from the source.

Solved problem 22.4

Shuffle a copy while retaining the original.

Reasoning and algorithm

shuffle mutates its list and returns None. Copy first if the original order must be retained.

Program

```
import random
rng = random.Random(4)
original = [1, 2, 3, 4]
shuffled = original.copy()
rng.shuffle(shuffled)
print(original)
print(sorted(shuffled) == original)
```

Output

```
[1, 2, 3, 4]
True
```

Why this result follows

The copy may be rearranged, while the original stays unchanged. Sorting verifies that no values were lost.

Boundary and error check

Compare sorted contents before and after and verify the original list remains unchanged.

Solved problem 22.5

Generate a four-character classroom practice code.

Reasoning and algorithm

Choose each character from the stated alphabet, repeat exactly four times, and join. This is classroom pseudorandomness, not secure token generation.

Program

```
import random
rng = random.Random(21)
code = ""
for position in range(4):
    code += str(rng.randint(0, 9))
print(len(code), code.isdigit())
```

Output

```
4 True
```

Why this result follows

Keeping the code as a string preserves possible leading zeroes. This is a simulation, not secure authentication code.

Boundary and error check

Check length and membership of every character; use secrets for real security tokens.

Trace a generated code

Step	Action	Invariant
Initial	code = ""	Length 0
Each of 4 turns	Append one digit	Length grows by 1
After loop	print checks	Length 4; all digits

A leading zero would survive because code is a string. The seeded demonstration is reproducible but is not a security token.

22. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 22.1

List every possible result of `randrange(5, 20, 5)`. Compare it with `randint(5, 20)`.

Exercise 22.2

Simulate 100 coin tosses and count heads and tails in a dictionary. Verify that the two counts add to 100.

Exercise 22.3

Select one name from a nonempty tuple using `choice`. Explain what should happen if the tuple is empty.

Exercise 22.4

Explain why `result` becomes `None`. Write the correct approach to shuffle and then display the list.

```
import random
values = [1, 2, 3]
result = random.shuffle(values)
```

Exercise 22.5

Generate a six-digit integer whose first digit cannot be zero. State the inclusive `randint` bounds before coding.

CHAPTER 23

23. The statistics module

Summarise a collection of numbers with measures of centre and spread. Choose a statistic that answers the actual question.

23.1 Data and the question

A statistic compresses several observations into a summary. Before computing one, decide whether all measurements represent comparable quantities and whether missing values have been handled. A list of exam marks may be suitable; combining marks out of 20 with marks out of 100 without adjustment is not. `import statistics` gives access to the standard functions discussed below.

23.2 Mean

The arithmetic mean is the sum of the values divided by their count. `statistics.mean(data)` performs this calculation for a nonempty collection. Every observation contributes, so one unusually large value can pull the mean upwards. Repeated values matter; do not convert the data into a set before averaging. The arithmetic mean is not necessarily one of the observations.

23.3 Median

Sort the values conceptually and locate the middle. With an odd number of observations, the median is the central value; with an even number, it is the average of the middle two. `statistics.median` handles the ordering internally and does not require you to mutate the source list. Because it depends on positions rather than the magnitude of every outlier, it can better represent the centre of skewed data.

23.4 Mode and ties

The mode is a most frequent value. In modern Python, `statistics.mode` returns the first encountered value among tied modes. `statistics.multimode` returns all modes in first-occurrence order. There may be several modes, and with all-distinct data every value is tied in frequency. Explain the tie rule rather than assuming mode always reports a unique peak.

23.5 Spread and valid inputs

Population standard deviation treats the data as the entire population; sample standard deviation estimates variation from a sample and uses a different denominator. `statistics.pstdev` accepts one observation, while `stdev` needs at least two. Mean, median and mode reject empty data with `StatisticsError`. For this introductory course, first master centre measures; use spread functions only after understanding what the data represents.

Visual walkthrough: The statistics module

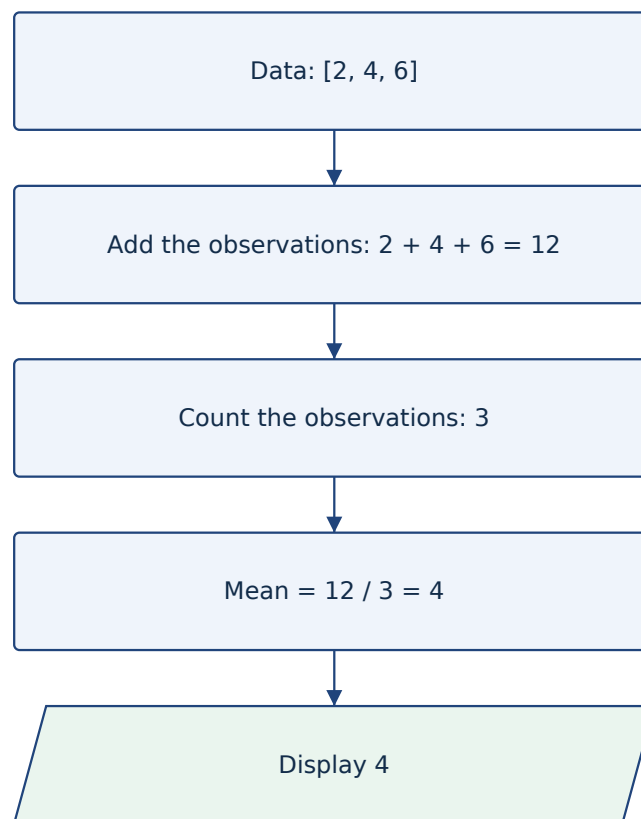


Figure 23.1. Compute the total and count before dividing. The arrows connect the same quantities used by `statistics.mean`.

23.6 A first example

```
import statistics as stats
values = [2, 4, 6]
print(stats.mean(values))
```

Output

```
4
```

The total is 12 and the count is 3. The mean is 4, a measure of centre rather than a claim about every observation.

CHAPTER 23 / WORKED WALKTHROUGH

Check the data contract first

Before a statistic, inspect empty input and the meaning of ties. The arithmetic mean is total/count and is undefined for count zero. Median sorts conceptually and selects the middle value or averages the two middle values. Multimode returns all most frequent values.

Trace the program

```
from statistics import mean, median, multimode
data = [2, 4, 4, 6]
print(mean(data), median(data), multimode(data))
```

State trace

Sum / count	$16 / 4 = 4$
Middle pair	$(4 + 4) / 2 = 4$
Most frequent	[4]

A common wrong turn

A one-mode answer can hide tied modes. An empty input needs a deliberate policy before calling mean.

Check your understanding

Predict `median([2, 9, 3])` and `multimode([1, 2, 1, 2])`.

23. Concept workshop: A statistic answers a question

Mean responds to total magnitude, median to middle order, and mode to frequency. Decide which question the data calls for before choosing a function. A tie may yield several modes. Spread also depends on interpretation: population statistics describe all observations under study, while sample variance estimates a larger population from observations. Empty data and one-item samples are domain questions, not formatting problems. Guard them deliberately and explain the chosen policy.

Execution model



Use the model before the code

23.1 Mean uses every magnitude, median uses sorted middle position, and mode uses frequency.

23.2 Sort conceptually and average the two central observations for even length.

23.3 A tied highest frequency means multiple modes.

23.4 Mean divides by count, so empty input needs a guard or a documented exception.

23.5 Population variance divides by n ; sample variance divides by $n-1$ and estimates spread from a sample.

Boundary habit

Predict `median([2, 9, 3])` and `multimode([1, 2, 1, 2])`.

Challenge the model

23.1 Test an outlier and a constant list; state what happens on empty data.

23.3 Test all-unique data and two equally frequent peaks.

23.5 Check one element: population spread is defined as zero, sample spread is not.

23. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 23.1

Compare the three common centre measures.

Reasoning and algorithm

Mean uses every magnitude, median uses sorted middle position, and mode uses frequency. Apply all three to the same data to see why they differ.

Program

```
import statistics as stats
values = [4, 6, 6, 8, 16]
print(stats.mean(values))
print(stats.median(values))
print(stats.mode(values))
```

Output

```
8
6
6
```

Why this result follows

The mean is 8, the median is 6 and the most frequent value is 6. Each answers a different summary question.

Boundary and error check

Test an outlier and a constant list; state what happens on empty data.

Solved problem 23.2

Calculate an even-length median.

Reasoning and algorithm

Sort conceptually and average the two central observations for even length. Do not take one middle element.

Program

```
import statistics as stats
values = [9, 1, 5, 3]
print(stats.median(values))
```

Output

```
4.0
```

Why this result follows

The sorted order is 1, 3, 5, 9. The middle two values average to 4.0.

Boundary and error check

Check two elements and four elements with distinct middle values.

Solved problem 23.3

Handle tied modes explicitly.

Reasoning and algorithm

A tied highest frequency means multiple modes. `multimode` makes that ambiguity explicit rather than hiding all but one.

Program

```
import statistics as stats
values = [7, 2, 7, 2, 9]
print(stats.mode(values))
print(stats.multimode(values))
```

Output

```
7
[7, 2]
```

Why this result follows

7 and 2 each occur twice. `mode` chooses the first encountered tied value; `multimode` returns both.

Boundary and error check

Test all-unique data and two equally frequent peaks.

Solved problem 23.4

Avoid calculating a mean for empty data.

Reasoning and algorithm

`Mean` divides by count, so empty input needs a guard or a documented exception. Choose the policy before calling the function.

Program

```
import statistics as stats
values = []
if values:
    print(stats.mean(values))
else:
    print("No observations")
```

Output

```
No observations
```

Why this result follows

An empty collection is falsey. The guard prevents an undefined empty-data mean.

Boundary and error check

Test [] and a singleton; avoid treating zero as if it were missing data.

Solved problem 23.5

Compare population and sample spread.

Reasoning and algorithm

Population variance divides by n ; sample variance divides by $n-1$ and estimates spread from a sample. Name the data interpretation first.

Program

```
import statistics as stats
values = [2, 4, 6]
print(f"{stats.pstdev(values):.3f}")
print(f"{stats.stdev(values):.3f}")
```

Output

```
1.633
2.000
```

Why this result follows

The functions use different denominators. Select the one matching the interpretation of the data, not the smaller result.

Boundary and error check

Check one element: population spread is defined as zero, sample spread is not.

Compute both spread definitions

Quantity	Population	Sample
Mean	4	4
Squared deviations	$4 + 0 + 4 = 8$	8
Variance	$8 / 3$	$8 / 2 = 4$
Standard deviation	1.633	2.000

A one-element sample makes $n - 1$ equal zero; sample spread is undefined there.

23. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 23.1

Compute mean, median and mode by hand for [3, 5, 5, 7, 10], then verify them with statistics.

Exercise 23.2

Predict the median of [12, 2, 8, 4]. Explain why averaging only the first and last input values is wrong.

Exercise 23.3

Create data with two tied modes and compare mode with multimode. State the tie order.

Exercise 23.4

Add a very large outlier to a small list and compare how the mean and median change.

Exercise 23.5

Write a program that prints count, minimum, maximum, mean and median for nonempty data and a clear message for empty data.

CHAPTER 24

24. Printing patterns

Patterns turn loop logic into something you can see. Build each line by deciding its spaces, number of items and item values before writing the loops.

24.1 A row is the unit of design

A pattern is a sequence of output rows. Start by drawing a small target on paper and numbering its rows from 1. For each row, answer three questions: how many leading spaces, how many symbols, and which symbol or number? Keep these decisions separate. A useful row table for a left triangle has row numbers 1, 2, 3 and symbol counts 1, 2, 3. This converts a picture into simple arithmetic.

24.2 Control the newline

`print` normally adds a newline. `print("*", end="")` prints a star while staying on the same line. Call `print()` after completing a row to move down once. In a nested-loop solution, the row-ending print belongs inside the outer loop and outside the inner loop. If it is inside both loops, every star appears on a new line. If it is outside both, all rows run together.

24.3 Rectangles and triangles

For a rectangle with rows `r` and columns `c`, every row prints `c` symbols. A left triangle prints row symbols on row number `row`. A descending triangle prints `n-row+1` symbols. You can build these with an inner loop or with string repetition such as `"*" * row`. Both are useful: the loop exposes the mechanism, while repetition expresses a repeated string compactly. First learn why the row length is correct.

24.4 Alignment and pyramids

A right-aligned triangle of height `n` uses `n-row` leading spaces and `row` stars. A centred pyramid of height `n` uses `n-row` leading spaces and `2*row-1` stars. These formulas assume each star occupies one column and there are no spaces inserted between stars. If you print `"* "` instead, the width changes and the spacing formula must change too. Monospace text makes alignment predictable.

24.5 Numbers inside a pattern

The printed value need not equal the row length. To print 1, 12, 123, use the inner column number. To print 1, 22, 333, use the outer row number. To continue counting across rows, initialise a separate value before the outer loop and increase it after every printed number. Values with two digits occupy more columns than values with one digit; use formatting such as `f"{value:3d}"` when aligned numeric columns are required.

24.6 Hollow shapes and testing

For a hollow rectangle, print a star when row is the first or last row, or column is the first or last column; otherwise print a space. Test dimensions 1, 2 and a larger size. Negative row counts are invalid for these tasks, and a zero-row pattern can reasonably print nothing if specified. Debug one row at a time. Examine whitespace with `repr(line)` when you cannot see why two outputs look different.

Visual walkthrough: Printing patterns

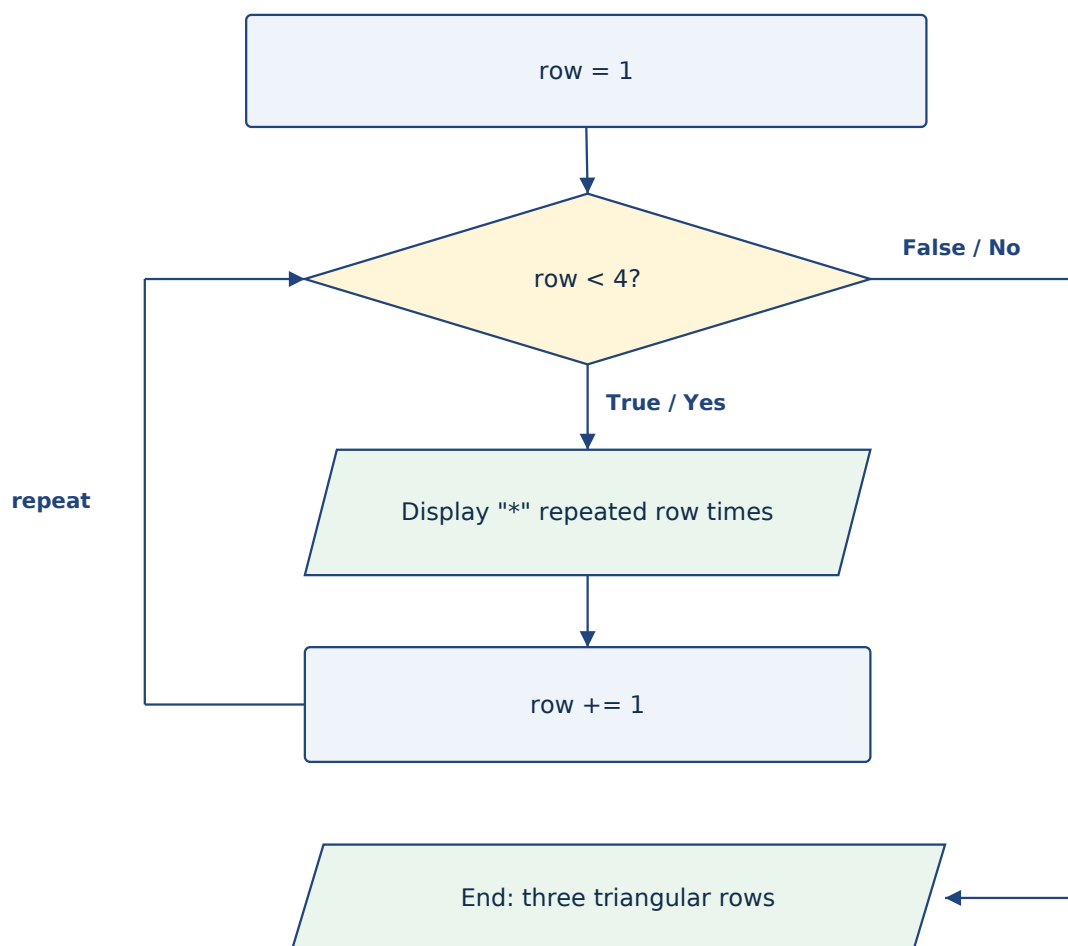


Figure 24.1. The row number controls the star count. This counter chart is equivalent to the range-based first example.

24.7 A first example

```
for row in range(1, 4):  
    print("*" * row)
```

Output

```
*  
**  
***
```

The outer loop chooses the row, and repetition produces that row's symbols. Each print ends one line.

Concept bridge: Derive counts for every row

Pattern printing is a small algebra problem. Start with a table of row number, leading spaces, interior spaces if any, and printed symbols. For a pyramid of n rows, row r has $n-r$ leading spaces and $2r-1$ stars. For a hollow rectangle, print a border when row is first or last or column is first or last. The final newline happens once per row. Test $n=1$ because first and last row coincide, then test a typical shape.

24. Concept workshop: Derive counts for every row

Pattern printing is a small algebra problem. Start with a table of row number, leading spaces, interior spaces if any, and printed symbols. For a pyramid of n rows, row r has $n-r$ leading spaces and $2r-1$ stars. For a hollow rectangle, print a border when row is first or last or column is first or last. The final newline happens once per row. Test $n=1$ because first and last row coincide, then test a typical shape.

Execution model

r=1	2 spaces, 1 star
r=2	1 space, 3 stars
r=3	0 spaces, 5 stars

Use the model before the code

24.1 The outer loop counts rows; the inner loop supplies four columns per row.

24.2 Row r contains r stars.

24.3 Right alignment needs leading spaces $n-r$ followed by r stars.

24.4 A centred pyramid uses $n-r$ leading spaces and $2r-1$ stars.

24.5 Carry the next integer across rows while row lengths grow.

Boundary habit

Derive an inverted pyramid rule for star counts 5, 3, 1; check $n=1$.

Challenge the model

24.1 Count twelve stars total and test a one-row rectangle.

24.3 Check the first and last rows; trailing spaces are unnecessary.

24.5 For n rows the total printed count is $n(n+1)/2$; verify a small n by hand.

24. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 24.1

Print a rectangle with three rows and four columns.

Reasoning and algorithm

The outer loop counts rows; the inner loop supplies four columns per row. End each row with one newline.

Program

```
for row in range(3):
    for column in range(4):
        print("*", end="")
    print()
```

Output

```
****
****
****
```

Why this result follows

Every outer iteration prints four stars. The final print ends that row, so the result has three lines.

Boundary and error check

Count twelve stars total and test a one-row rectangle.

Solved problem 24.2

Print a left-aligned triangle with four rows.

Reasoning and algorithm

Row r contains r stars. Generate the row text or use an inner loop, then print one complete line.

Program

```
for row in range(1, 5):
    print("*" * row)
```

Output

```
*
**
***
****
```

Why this result follows

Each row repeats the star character exactly row times. The counts for rows 1 through 4 are 1, 2, 3 and 4, each followed by a newline.

Boundary and error check

Check one row and the four-row star-count sequence 1,2,3,4.

Solved problem 24.3

Print a right-aligned triangle with four rows.

Reasoning and algorithm

Right alignment needs leading spaces $n-r$ followed by r stars. Derive the two counts from the row before printing.

Program

```
n = 4
for row in range(1, n + 1):
    spaces = n - row
    stars = row
    print(" " * spaces + "*" * stars)
```

Output

```
  *
 **
 ***
****
```

Why this result follows

Each row has width four. As one leading space disappears, one star is added.

Boundary and error check

Check the first and last rows; trailing spaces are unnecessary.

Solved problem 24.4

Print a centred pyramid with four rows.

Reasoning and algorithm

A centred pyramid uses $n-r$ leading spaces and $2r-1$ stars. Both sequences change predictably with r .

Program

```
n = 4
for row in range(1, n + 1):
    spaces = n - row
    stars = 2 * row - 1
    print(" " * spaces + "*" * stars)
```

Output

```

*
***
*****
*****

```

Why this result follows

For rows 1 through 4, the (spaces, stars) pairs are (3, 1), (2, 3), (1, 5), (0, 7). Each iteration prints one whole row, then advances.

Boundary and error check

For $n=4$ verify star counts 1,3,5,7 and space counts 3,2,1,0.

Solved problem 24.5

Print consecutive integers in triangular rows.

Reasoning and algorithm

Carry the next integer across rows while row lengths grow. The counter update belongs inside the inner loop.

Program

```

value = 1
for row in range(1, 4):
    line = ""
    for column in range(row):
        line += f"{value:3d}"
        value += 1
    print(line)

```

Output

```

1
2  3
4  5  6

```

Why this result follows

The counter belongs before the outer loop so it continues across row boundaries. Each number occupies three character positions.

Boundary and error check

For n rows the total printed count is $n(n+1)/2$; verify a small n by hand.

Trace the counter across rows

Row	Values printed	value afterward
1	1	2
2	2, 3	4
3	4, 5, 6	7

24. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 24.1

Print a five-row descending triangle whose star counts are 5, 4, 3, 2 and 1. Write its row-length rule first.

Exercise 24.2

Print four rows containing 1, 22, 333 and 4444. Explain why the row variable controls both value and repetition count.

Exercise 24.3

Print a hollow rectangle with five rows and seven columns. Test your border condition separately for a one-row rectangle.

Exercise 24.4

Print a centred inverted pyramid with star counts 7, 5, 3 and 1. Derive the leading-space formula.

Exercise 24.5

This code deliberately places a newline after every star. Repair the indentation and explain where the newline belongs.

```
for row in range(1, 4):
    for column in range(row):
        print("*", end="")
        print()
```

CHAPTER 25

25. Functions

A function gives a name to a reusable piece of behaviour. Learn to separate inputs, computation and returned results so that you can test one task at a time.

25.1 Why define a function?

Suppose several parts of a program need to calculate a rectangle area. Copying the same formula repeatedly makes corrections harder. A function stores the calculation once and lets each caller supply different dimensions. Begin with one clear responsibility per function. A good function name describes that responsibility, such as `rectangle_area`. Functions also make testing easier because you can call the same logic with normal, boundary and invalid values.

25.2 Definition and call

The keyword `def` starts a function definition. Write `def rectangle_area(length, width):` and indent the body beneath it. The names `length` and `width` are parameters: placeholders receiving values at call time. `rectangle_area(7, 4)` is a call; 7 and 4 are arguments. Executing `def` creates the function but does not run its body. The body runs when the function is called. Define the function before the first executed call that needs it.

25.3 Return versus print

`return` expression evaluates a value and sends it back to the caller, ending that function call immediately. The caller can store the result, print it or combine it with other results. `print` merely displays something; its return value is `None`. A function that reaches its end without `return`, or uses `return` without a value, returns `None`. Prefer returning a calculation result and letting the caller decide how to display it.

25.4 Arguments and defaults

Positional arguments match parameters by position. Keyword arguments such as `rectangle_area(width=4, length=7)` match by name. A default parameter, such as `tax_rate=0.05`, supplies a value when the caller omits that argument. Put ordinary required parameters before defaulted parameters. Avoid a mutable default such as `items=[]`: it is created once when `def` executes and can be shared across later calls. Use `None` and create a fresh list inside when a fresh collection is needed.

25.5 Local names and object references

Parameters and names assigned inside a function are normally local to that function. A local name does not automatically exist in the caller. Rebinding a numeric parameter leaves the caller's name unchanged. Mutating a list passed as an argument can change the same list seen by the caller, because both refer to that object. Return new data when mutation would be surprising, or document the mutation explicitly. You do not need global variables for the programs in this chapter.

25.6 Multiple results and early return

A function can return a tuple such as `return total, count`. The caller can unpack it into two names. This is one returned object containing several values. Early return is useful when a case already has a complete answer, such as an empty input or an invalid condition. Every intended path should have a clear result. Avoid a function that sometimes returns a number and sometimes accidentally falls through to `None`.

25.7 Document and test behaviour

A string immediately inside the function body can serve as its docstring, describing purpose, parameters and result. State assumptions such as non-negative dimensions or a nonempty collection. Test the function directly with small known cases before using it in a larger program. Separate input prompts from core calculations so that tests do not depend on interactive typing. Recursive functions call themselves and require a stopping case, but recursion is not necessary for the beginner tasks here.

Visual walkthrough: Functions

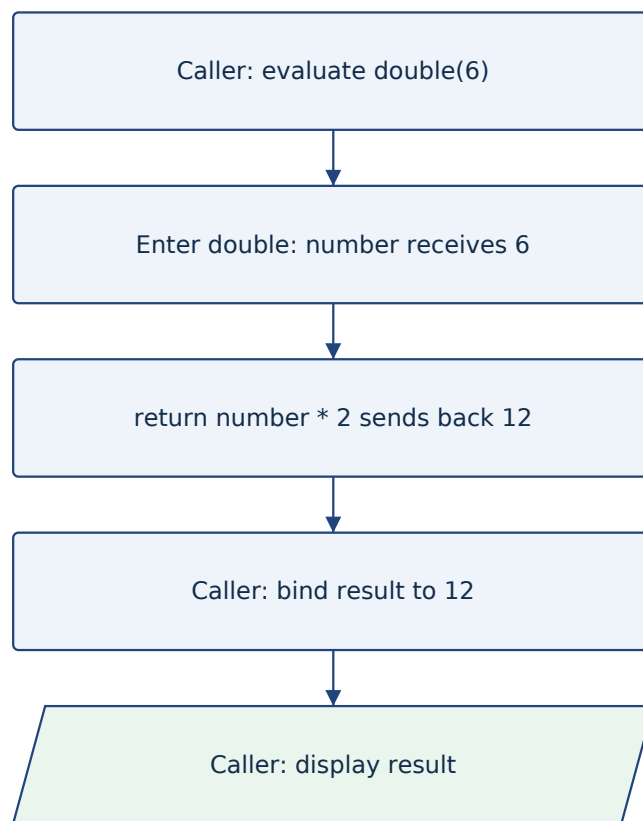


Figure 25.1. A function call transfers control to its body. `return` transfers the result and control back to the caller.

25.8 A first example

```
def double(number):  
    return number * 2  
  
result = double(6)  
print(result)
```

Output

```
12
```

The call supplies `6`, the parameter `number` receives it, `return` sends `12` back, and the caller prints the stored result.

CHAPTER 25 / WORKED WALKTHROUGH

Follow the function call and return

At a call, evaluate arguments, bind parameter names, execute the body, then return a value to the caller. The caller can store or combine that value. `print` shows text and returns `None`.

Assigning a new value to a local parameter does not rebind the caller name.

Trace the program

```
def double(x):  
    result = x * 2  
    return result  
  
n = 6  
answer = double(n)  
print(answer + 1)
```

State trace

Call	x -> 6, n -> 6
Body	result -> 12
Return	answer -> 12
Caller	13

A common wrong turn

A default list is created when `def` executes, not on every call. Use `None` as the default and create a fresh list inside when needed.

Check your understanding

Predict the result if `double` prints result but omits `return`. Identify the failed line in the caller.

CHAPTER 25 / WORKED WALKTHROUGH

Scope, mutation and safe defaults

A parameter is a local name, but it may point to a caller-owned mutable object. Rebinding the local name changes only the local arrow; `append` mutates the shared list object. A default expression is evaluated once at definition time, so a mutable default can carry state across calls.

Trace the program

```
def collect(item, items=None):  
    if items is None:  
        items = []  
    items.append(item)  
    return items  
print(collect(1), collect(2))
```

State trace

First call	Fresh list -> [1]
Second call	Another list -> [2]
Supplied list	Caller sees append on that list

A common wrong turn

Returning two values uses a tuple: `return low, high`. At the caller, `low, high = function(...)` unpacks it. Empty input needs an explicit contract.

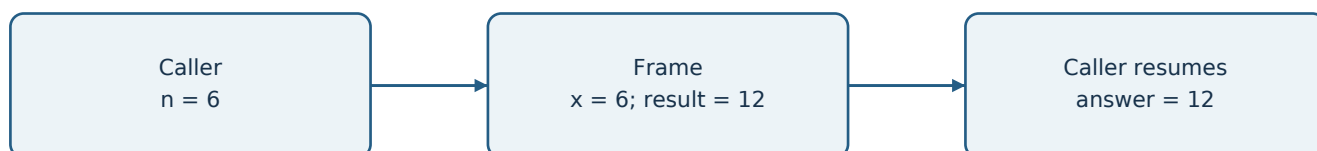
Check your understanding

Write a version that never mutates a supplied list, using `items.copy()` before `append`.

25. Concept workshop: Function contracts and frames

A function contract names its inputs, its returned result and any side effect. At a call, argument expressions are evaluated, parameter names bind in a new local frame, the body runs, and return sends a value back. Rebinding a parameter changes its local arrow; mutating a list reached through that parameter changes the shared object. A mutable default is constructed once when def executes, so use None and create a new list during each call when fresh state is intended. Returning two values packs a tuple that the caller can unpack. Keep print at the user interface boundary so calculations remain composable.

Execution model



Use the model before the code

25.1 Name the computation and its inputs, return the area, then let callers choose different dimensions.

25.2 Parity is a Boolean expression.

25.3 The default value is used only when the caller omits that argument.

25.4 Track total through the values and return total and count together as a tuple.

25.5 A print-only function returns None, so its caller cannot add one to the displayed number.

Boundary habit

Predict the result if double prints result but omits return. Identify the failed line in the caller.

Challenge the model

25.1 Test zero dimensions and two unequal rectangles.

25.3 Call with only the required name, then with the named override.

25.5 Test the return value independently from printed output.

25. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 25.1

Define an area function and reuse it with two inputs.

Reasoning and algorithm

Name the computation and its inputs, return the area, then let callers choose different dimensions. Definition alone does not run the body.

Program

```
def rectangle_area(length, width):  
    return length * width  
print(rectangle_area(7, 4))  
print(rectangle_area(3, 3))
```

Output

```
28  
9
```

Why this result follows

Each call gets its own parameter bindings. The formula is written once and produces 28 and 9 for the two calls.

Boundary and error check

Test zero dimensions and two unequal rectangles.

Solved problem 25.2

Return whether a number is even.

Reasoning and algorithm

Parity is a Boolean expression. Return that result directly rather than printing it or building an unnecessary if/else.

Program

```
def is_even(number):  
    return number % 2 == 0  
print(is_even(0))  
print(is_even(7))
```

Output

```
True  
False
```

Why this result follows

For 0, the remainder is 0 and the function returns True. For 7, the remainder is 1 and it returns False. The caller prints each result.

Boundary and error check

Test zero, a positive odd number and a negative even number.

Solved problem 25.3

Use a default parameter for a greeting.

Reasoning and algorithm

The default value is used only when the caller omits that argument. A keyword argument can replace it without changing parameter order.

Program

```
def greet(name, greeting="Hello"):
    return greeting + ", " + name
print(greet("Asha"))
print(greet("Dev", greeting="Welcome"))
```

Output

```
Hello, Asha
Welcome, Dev
```

Why this result follows

The first call uses the default greeting. The second overrides it by keyword.

Boundary and error check

Call with only the required name, then with the named override.

Solved problem 25.4

Return both a total and a count.

Reasoning and algorithm

Track total through the values and return total and count together as a tuple. The caller unpacks both values.

Program

```
def summarise(values):
    total = 0
    for value in values:
        total += value
    return total, len(values)
total, count = summarise([4, 6, 8])
print(total, count)
```

Output

18 3

Why this result follows

The function returns a two-item tuple, which the caller unpacks. Empty input would return total 0 and count 0.

Boundary and error check

For empty input expect (0,0); do not divide unless count is positive.

Solved problem 25.5

Repair a calculation function that prints instead of returning.

Reasoning and algorithm

A print-only function returns None, so its caller cannot add one to the displayed number. Return the calculated value and print at the boundary.

Program

```
def square(number):
    return number * number
answer = square(5)
print(answer + 1)
```

Output

26

Why this result follows

If square only printed its result, answer would be None and the addition would fail. Returning 25 allows the caller to continue calculating.

Boundary and error check

Test the return value independently from printed output.

Trace the return boundary

Step	Inside square	At caller
Call square(5)	number = 5	Wait for result
return 5 * 5	25	answer = 25
print(answer + 1)	Function finished	Print 26

Replacing return with print(25) would display 25, then assign None to answer; None + 1 raises TypeError.

25. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 25.1

Define `cube(number)` to return a cube. Call it with a positive integer, zero and a negative integer; predict each result first.

Exercise 25.2

Define `count_vowels(text)` to count every vowel occurrence, ignoring case. Keep input and print statements outside the function.

Exercise 25.3

Predict the caller's value after `x = 5` and a call to a function that assigns 99 to its parameter. Explain the difference if a list parameter is modified with `append`.

Exercise 25.4

Write `min_max(values)` returning the minimum and maximum of a nonempty numeric list. State and test the input assumption.

Exercise 25.5

Define a function that returns an average, returning `None` for empty data. Write caller code that displays an appropriate message in either case.

CHAPTER 26

26. Classes

A class describes a kind of object by grouping related data and behaviour. Start with a small object whose changing state you can follow.

26.1 From separate values to one object

A student might have a name and mark, while a rectangle has a length and width. A class groups related values, called attributes, with operations on them, called methods. The class describes the kind of object; each instance is an individual object created from it. Two Rectangle instances may share the same methods but hold different measurements. Use a class when keeping data and associated behaviour together makes the program clearer.

26.2 Class definition and construction

`class Rectangle:` starts a class definition. Indent method definitions inside it. Calling `Rectangle(7, 4)` creates an instance and runs its initialisation method. `__init__` is the special method used to initialise the instance; it should not return a result other than `None`. Within it, `self.length = length` stores the supplied value on this object. The double underscores in `__init__` are part of its exact name.

26.3 The role of self

`self` is the conventional name of a method's first parameter and refers to the instance receiving the call. In `rect.area()`, Python supplies `rect` as `self` automatically. Therefore `def area(self):` has one parameter even though `rect.area()` supplies no explicit argument. `self.length` accesses this object's length; plain `length` refers to a local or otherwise resolved name. Forgetting `self` can create a local variable instead of an instance attribute.

26.4 Attributes and methods

Read an attribute with dot notation, such as `rect.length`. Call a method with parentheses, such as `rect.area()`. A method can calculate a result from attributes or update the object's state. Return values work just as they do for ordinary functions. Validate updates when the object has rules, such as a counter that cannot go below zero. Keep input and output outside the class's core calculation methods when possible.

26.5 Independent instances and shared references

Attributes assigned on `self` inside `__init__` belong to each instance. If two objects each create `self.items = []`, they get separate lists. A list assigned directly in the class body can instead be shared by instances, which is often surprising for beginners. Also remember that `second = first` does not create another instance: it makes another reference to the same one. Construct a new object when you want independent state.

26.6 A useful design sequence

First list what each object needs to remember. Next write an initializer that establishes a valid starting state. Then add one method for each clear operation. Create two instances to test that their data is independent, and trace the attributes after a modifying method. Avoid turning every small arithmetic expression into a class: a function is often enough for a single calculation without persistent state.

26.7 Encapsulation and later concepts

Encapsulation means organising an object's state and the methods that operate on it behind a clear interface. Python does not make an attribute secret merely because it belongs to a class. A leading underscore conventionally marks an internal attribute. Inheritance defines a specialised class from another class, and polymorphism allows different objects to support similar operations. These are later topics; first learn construction, instance attributes, `self` and method calls reliably.

Visual walkthrough: Classes

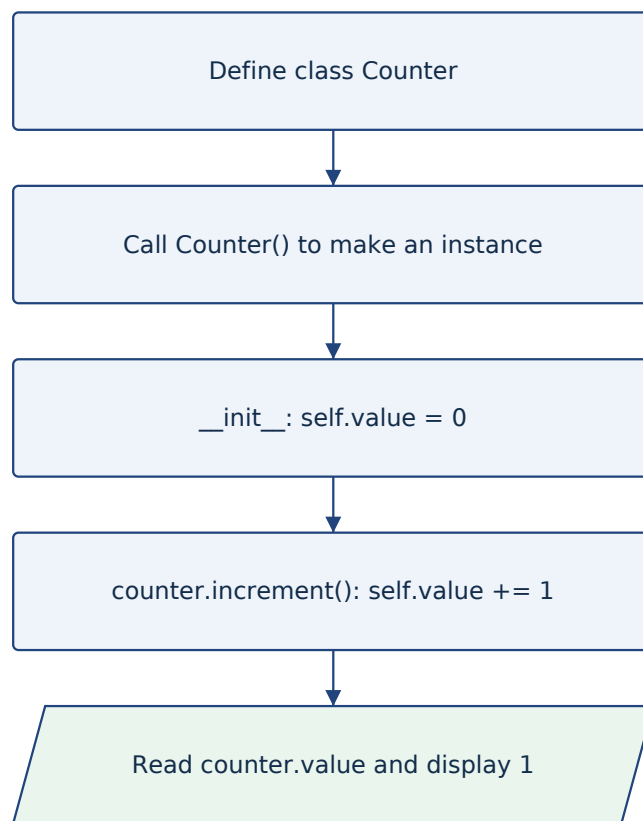


Figure 26.1. `self` refers to the receiving instance. Construction establishes state and a method changes that state.

26.8 A first example

```
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1

counter = Counter()
counter.increment()
print(counter.value)
```

Output

```
1
```

The class groups the stored value with an operation that updates it. This one instance moves from state 0 to state 1.

CHAPTER 26 / WORKED WALKTHROUGH

Separate classes from instances

A class describes how instances are built and what methods they have. `__init__` runs for each construction and stores instance attributes on `self`. Method calls supply the instance as `self`. Draw a separate box per instance; each should have its own mutable list when independent state is intended.

Trace the program

```
class Counter:
    def __init__(self):
        self.value = 0
    def add(self):
        self.value += 1
a = Counter()
b = Counter()
a.add()
print(a.value, b.value)
```

State trace

Construct a	a.value = 0
Construct b	b.value = 0
a.add()	a.value = 1, b.value = 0

A common wrong turn

A mutable class attribute such as `items = []` can be shared across instances. Put `self.items = []` in `__init__` for separate bags.

Check your understanding

Call `a.add()` twice more. Explain why `b.value` remains zero.

CHAPTER 26 / WORKED WALKTHROUGH

Trace object identity and shared data

A class attribute is stored on the class and found through instances unless shadowed. If that attribute is a list, every instance can reach the same list. Initialise mutable per-object data in `__init__`. A method call `bag.add(x)` is conceptually `Bag.add(bag, x)`.

Trace the program

```
class Bag:
    def __init__(self):
        self.items = []
    def add(self, item):
        self.items.append(item)
a, b = Bag(), Bag()
a.add("pen")
print(a.items, b.items)
```

State trace

a.__init__	a.items -> new list A
b.__init__	b.items -> new list B
a.add	list A = ["pen"]; list B = []

A common wrong turn

If `items = []` sat in the class body instead, both instances would find and mutate the same list until an instance attribute shadowed it.

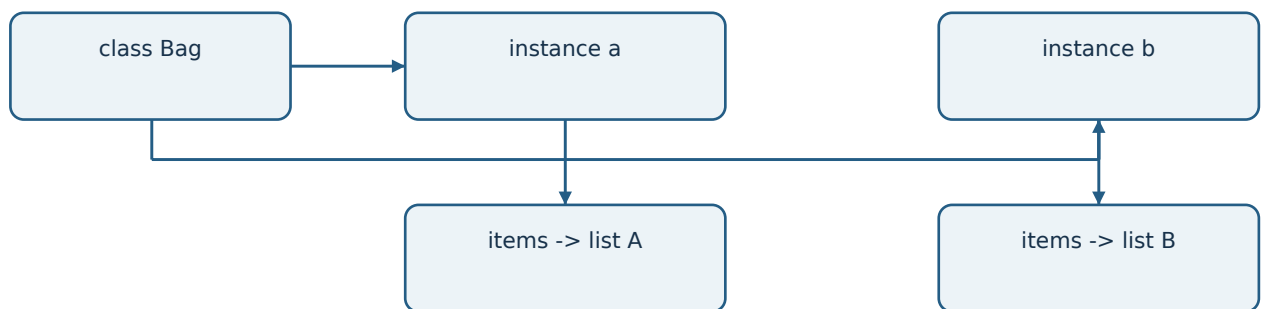
Check your understanding

Draw the two instance boxes and their list arrows; test `a.items` is `b.items`.

26. Concept workshop: Objects, class attributes and ownership

A class supplies construction and method behavior; each instance owns its own attributes assigned through `self` in `__init__`. A method call supplies the instance as `self`. Two distinct Counter objects can therefore advance independently, while two names assigned to one Counter refer to the same state. A mutable attribute placed on the class may be shared by every instance that looks it up. Put per-object lists on `self` in the initializer and test identity with `is`. State invariants for methods: for a nonnegative counter, `add` should change exactly one object's value by one.

Execution model



Use the model before the code

- 26.1 Store dimensions on `self` during construction; `area` reads that instance state.
- 26.2 Each Counter construction sets `self.value` to zero.
- 26.3 A method can read `self.mark` and return a Boolean pass rule.
- 26.4 A bag needs `self.items = []` in `__init__`, creating a fresh list per instance.
- 26.5 Assignment of one object to a second name creates an alias, not a new instance.

Boundary habit

Call `a.add()` twice more. Explain why `b.value` remains zero.

Challenge the model

- 26.1 Test a zero side and reject or document negative dimensions.
- 26.3 Test marks on either side of the threshold and the exact boundary.
- 26.5 Use `is` to check identity; create a separate instance for independent state.

26. Worked solutions

For each problem, read the method and predict a result before looking at the program. The final check tests an assumption or a boundary.

Solved problem 26.1

Create a rectangle object and calculate its area.

Reasoning and algorithm

Store dimensions on self during construction; area reads that instance state. Two rectangle objects may carry different dimensions.

Program

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width
rect = Rectangle(7, 4)
print(rect.area())
```

Output

28

Why this result follows

Construction stores 7 and 4 on rect. The method receives rect as self and returns the product of its attributes.

Boundary and error check

Test a zero side and reject or document negative dimensions.

Follow the method call

Step	Instance state	Returned value
Rectangle(7, 4)	length=7, width=4	rect reference
rect.area()	self is rect	7 * 4 = 28
print(...)	State unchanged	28 displayed

A second Rectangle(3, 3) would carry its own dimensions and return 9.

Solved problem 26.2

Keep two counters independent.

Reasoning and algorithm

Each Counter construction sets `self.value` to zero. A method call updates only the instance passed as `self`.

Program

```
class Counter:
    def __init__(self):
        self.value = 0
    def increment(self):
        self.value += 1
first = Counter()
second = Counter()
first.increment()
first.increment()
print(first.value, second.value)
```

Output

```
2 0
```

Why this result follows

Two constructor calls make two Counters: `first.value = 0` and `second.value = 0`. Both `increment` calls target `first`, giving (2, 0).

Boundary and error check

Increment one counter and verify the other remains unchanged.

Solved problem 26.3

Give a student object a pass-checking method.

Reasoning and algorithm

A method can read `self.mark` and return a Boolean pass rule. Construction fills the attribute before the method is called.

Program

```
class Student:
    def __init__(self, name, mark):
        self.name = name
        self.mark = mark
    def has_passed(self):
        return self.mark >= 40
student = Student("Mira", 40)
print(student.name, student.has_passed())
```

Output

```
Mira True
```

Why this result follows

The boundary is inclusive: exactly 40 passes. The example assumes the supplied mark is already within 0 through 100.

Boundary and error check

Test marks on either side of the threshold and the exact boundary.

Solved problem 26.4

Create separate item lists for two bags.

Reasoning and algorithm

A bag needs `self.items = []` in `__init__`, creating a fresh list per instance. A class-level shared list would couple separate bags.

Program

```
class Bag:
    def __init__(self):
        self.items = []
    def add(self, item):
        self.items.append(item)
red = Bag()
blue = Bag()
red.add("book")
print(red.items)
print(blue.items)
```

Output

```
['book']
[]
```

Why this result follows

A fresh list is created for each instance inside `__init__`. Adding to red does not change blue.

Boundary and error check

Add an item to the first bag and check the second is still empty.

Follow two independent bags

Step	red.items	blue.items
red = Bag()	[]	Not created
blue = Bag()	[]	[] (new list)
red.add("book")	["book"]	[]

The call `red.add` passes `red` as `self`. Because each constructor makes a new list, blue remains empty.

Solved problem 26.5

Demonstrate two names referring to one object.

Reasoning and algorithm

Assignment of one object to a second name creates an alias, not a new instance. Draw both arrows to the same object before mutating.

Program

```
class Lamp:
    def __init__(self):
        self.is_on = False
    def switch_on(self):
        self.is_on = True
first = Lamp()
alias = first
alias.switch_on()
print(first.is_on)
```

Output

True

Why this result follows

alias and first refer to the same Lamp instance. Calling a method through either name changes that one object.

Boundary and error check

Use is to check identity; create a separate instance for independent state.

Trace the object reference

Step	first	alias
Lamp()	Object A: False	Not assigned
alias = first	Object A: False	Same object A
alias.switch_on()	Object A: True	Same object A
print(first.is_on)	True	No new object

first is alias evaluates to True. A second Lamp() call would create an independent object.

26. Practice problems

For each exercise, write the input assumption, an example trace and at least one boundary test before running your program.

Exercise 26.1

Create a Circle class with a radius attribute and an area() method using math.pi. Instantiate two circles with different radii.

Exercise 26.2

Add a reset() method to Counter. Create two counters, change both and reset only one. Display their final values.

Exercise 26.3

Explain the difference between Student, Student("Asha", 80), student.mark and student.has_passed(). Identify a class, construction, attribute access and method call.

Exercise 26.4

Create a Book class with title and is_borrowed attributes and borrow() and return_book() methods. Decide how repeated borrowing should be handled.

Exercise 26.5

Repair the initializer so that the object retains its name. Explain the missing instance attribute.

```
class Student:
    def __init__(self, name):
        name = name
```

Reference and study notes

A simple problem-solving routine

Identify the input, processing and output. Write a few pseudocode steps in ordinary language. Test the plan by hand on a small input. Translate it to Python, run it and compare the output with your independently calculated answer. For a total, initialise at zero; for a product, initialise at one. For a maximum, use a real first value rather than assuming zero is smaller than every possible input.

Testing checklist

Case	Why it matters
Typical input	Shows the usual route through the code.
Boundary	Distinguishes $<$ from $\leq$ and excluded from included endpoints.
Empty input	Exposes undefined averages and missing first elements.
Single element	Checks a smallest nonempty collection.
Repeated values	Checks equality, ties, counting and deduplication.
Negative or zero values	Checks arithmetic and termination assumptions.
Invalid input	Checks whether your stated validation rule is enforced.

Further reading

The following official references support further study. They are reference manuals, so use them after learning the introductory explanations rather than trying to memorise them.

Python tutorial: <https://docs.python.org/3/tutorial/>

Built-in types: <https://docs.python.org/3/library/stdtypes.html>

Built-in functions: <https://docs.python.org/3/library/functions.html>

math: <https://docs.python.org/3/library/math.html>

random: <https://docs.python.org/3/library/random.html>

statistics: <https://docs.python.org/3/library/statistics.html>

Reader note

This edition also includes dedicated beginner chapters on printing patterns, functions and classes. These extend the earlier KIPS-aligned sequence. File handling, inheritance in depth and third-party packages are possible next steps. The set chapter develops the set type introduced in the KIPS data-type discussion.

PART II / KIPS PYTHON PRACTICE

KIPS Python: theory and programs

An original companion to KIPS Computer Science with Python 11, Chapters 5-13 (printed pages 94-299). This section adds 36 answered theory questions, 36 theory self-checks with keys, 27 worked programs with execution traces, and 27 independent program tasks with test guidance. It complements the 26 topic chapters in Part I.

How to work through a unit

First answer the four theory questions without looking. For each worked program, trace the values before reading the explanation, then run the code in Python 3. Attempt the three independent programs and use their stated tests. The self-check answers are deliberately brief; explain each result aloud before marking it correct.

Navigation by KIPS chapter

- 5. Programming with Python | KIPS 94-104
- 6. Python Basics | KIPS 105-131
- 7. Data Types and Operators | KIPS 132-166
- 8. Control Flow | KIPS 167-198
- 9. String Manipulation | KIPS 199-223
- 10. List Manipulation | KIPS 224-248
- 11. Tuples | KIPS 249-267
- 12. Dictionaries | KIPS 268-290
- 13. Introduction to Python Modules | KIPS 291-299

Coverage boundary

These are new questions aligned to the skills and formats of the KIPS Python unit, not copied questions or an answer key to that book. The full KIPS volume also covers computer systems, number systems, Boolean algebra, problem solving, and society/law/ethics outside this Python companion. Use Python 3 behaviour when a source example has obsolete syntax or an incorrect printed answer.

KIPS 5. Programming with Python

KIPS 94-104 / Part I route: Running Python; Tokens; Errors

The first chapter asks for explanations of the language and its working modes as well as short shell and script activities. Separate what the editor, interpreter, prompt and displayed result do.

Theory: reason through the answer

T5.1 What changes between the interactive shell and a saved script?

The shell runs a submitted statement and immediately shows the value of a bare expression. A script executes a sequence from a .py file; use print when its result must be displayed. In either mode, Python uses the same language rules.

T5.2 What are source code, bytecode and the Python virtual machine?

Source is the text you write. In CPython it is compiled to bytecode, which its execution engine runs. The phrase “interpreted language” is a teaching shorthand, not a claim that compilation never occurs.

T5.3 MCQ: Which file normally stores a Python script? A .py; B .shell; C .idle; D .exe

A. The .py extension identifies ordinary Python source. IDLE is an editor and shell, not a source-file extension.

T5.4 Assertion: Python can run on multiple operating systems. Reason: its source must be rewritten in each operating system. Decide both truth values and whether the reason explains the assertion.

The assertion is generally true; the reason is false. Portable source can often run unchanged when compatible Python and dependencies exist, though platform-specific calls still require care.

Theory self-check

C5.1 Name two uses of the interactive shell.

C5.2 What does the prompt >>> signify?

C5.3 True or false: A bare 2 + 3 in a script always prints 5. Explain.

C5.4 Fill in: A commonly used beginner editor and shell included with Python is ____.

Self-check key

C5.1: Trying a small expression and inspecting a value or API quickly.

C5.2: The shell is ready for an input statement; it is not source code to type in a .py file.

C5.3: False. Use print(2 + 3) to display it in a script.

C5.4: IDLE.

Worked programs: plan, execute, trace

P5.1 Shell result versus script output

Show why an expression alone and an explicit print have different visible behaviour in a script.

Method. The expression is evaluated, but its value is discarded; print sends text to standard output.

```
2 + 3
print(2 + 3)
print("Ready")
```

Why this result follows. Line 1: value 5, no display. Line 2: display 5. Line 3: display Ready.

P5.2 Predict an arithmetic transcript

Calculate a short, deterministic transcript with both string repetition and division.

Method. Evaluate each expression first, then pass the value to print. Keep a numeric value distinct from a quoted string.

```
print("go" * 3)
print(17 // 5)
print(17 / 5)
```

Why this result follows. "go" repeated three times -> gogogo; 17 // 5 -> 3; 17 / 5 -> 3.4.

P5.3 A small saved script

Display a name and the hours in three days using a named value.

Method. Assign a number, multiply it, then print labelled results. A comment documents the unit conversion.

```
name = "Asha"
days = 3
# One day contains 24 hours.
print("Name:", name)
print("Hours:", days * 24)
```

Why this result follows. days starts at 3 -> 3 * 24 = 72; output: Name: Asha, then Hours: 72.

Independent program practice

U5.1 Write a script that prints a greeting, then the value and type of 7 / 2. Explain why a bare 7 / 2 differs in a shell.

Check: Expected numeric value 3.5; type float; the shell displays a bare expression, a script does not.

U5.2 In both a shell and a .py file, try "ha" * 2 and print("ha" * 2). Record what is visible.

Check: Predict before running; distinguish expression display from explicit output.

U5.3 Repair a script that calls Print("Hi") and leaves a closing parenthesis out of a second call.

Check: Both lines should execute; explain case-sensitive names and matching parentheses.

KIPS 6. Python Basics

KIPS 105-131 / Part I route: Tokens; Variables; Data types; Type conversion; Input and output; Errors

KIPS mixes definitions, token and identifier classification, type questions, short code, and error repair. The theory below gives reasons a learner can reuse when the spelling changes.

Theory: reason through the answer

T6.1 How do a keyword, identifier, literal and delimiter differ?

A keyword is reserved syntax such as `if`. An identifier is a programmer-chosen name such as `total`. A literal writes a value such as `12` or `"hi"`. Delimiters such as commas, colons and brackets separate or group parts of syntax.

T6.2 Why can a variable name later refer to a different type?

Assignment binds a name to an object. The object has a type; the name is not permanently declared to be that type. For example `x = 4` followed by `x = "four"` rebinds `x`.

T6.3 MCQ: Which is invalid as an identifier? A `total_2`; B `_total`; C `2total`; D `Total`

C. An identifier cannot begin with a digit. Case matters, so `Total` and `total` name different bindings.

T6.4 Assertion: `input()` returns text. Reason: `int(input())` converts valid integer text to an int. Are both true, and does the reason explain the assertion?

Both are true, but the conversion is a subsequent operation, not the reason `input` returns a `str`. Invalid numeric text raises `ValueError` during conversion.

Theory self-check

C6.1 Explain the difference between `=` and `==`.

C6.2 True or false: `#` inside a quoted string begins a comment.

C6.3 Classify `False`, `score`, `3.5` and `:`.

C6.4 Why does `int("3.5")` fail, while `float("3.5")` works?

Self-check key

C6.1: `=` assigns a binding; `==` compares values and produces a `bool`.

C6.2: False. It is a character in the string.

C6.3: Keyword/Boolean literal, identifier, float literal, delimiter, respectively.

C6.4: `int` accepts integer syntax in a string; `float` accepts the decimal form. To truncate, use `int(float("3.5"))`.

Worked programs: plan, execute, trace

P6.1 Swap two names safely

Exchange two stored marks without losing either value.

Method. Read the right side of a parallel assignment before rebinding either name.

```
first = 72
second = 88
first, second = second, first
print(first, second)
```

Why this result follows. Before: (72, 88). Evaluate RHS: (88, 72). Rebind: (88, 72). Output: 88 72.

P6.2 Turn text marks into arithmetic

Compute the mean of three marks supplied as strings.

Method. Convert each text value before addition; divide the numeric total by the count.

```
responses = ["68", "75", "82"]
marks = [int(item) for item in responses]
total = sum(marks)
print(total, total / len(marks))
```

Why this result follows. Conversions -> [68, 75, 82]; total -> 225; count -> 3; mean -> 75.0.

P6.3 Repair a name and quote error

Produce a message containing an apostrophe and a numeric count.

Method. Quote the text correctly and convert the number to text, or let print separate arguments.

```
count = 3
message = "It's time"
print(message, "for", count, "tests")
```

Why this result follows. message is a str; count is int; print separates four arguments with spaces. Output: It's time for 3 tests.

Independent program practice

U6.1 Read a price and a quantity, then print a two-decimal total. Avoid adding raw input strings.

Check: Check 12.50 and 3 -> 37.50; quantity zero -> 0.00.

U6.2 Classify names and literals in a five-line example; repair a name starting with a digit and a reserved-word name.

Check: Retest after renaming; explain case sensitivity.

U6.3 Convert a total number of seconds to hours, minutes and seconds.

Check: Check 3671 -> 1 hour, 1 minute, 11 seconds; also test 0 and 59.

KIPS 7. Data Types and Operators

KIPS 132-166 / Part I route: Data types; Type conversion; Operators; Expressions; Errors

This chapter demands careful distinctions: value versus identity, mutable versus immutable, precedence, conversion, and exact Python 3 output.

Theory: reason through the answer

T7.1 What makes a list mutable while a tuple is immutable?

A list supports replacement of an element in the existing object; a tuple does not. A tuple may contain a mutable list, whose contents can still change without replacing the tuple element.

T7.2 Why can `-17 // 5` be surprising?

Floor division rounds the quotient down toward negative infinity: `-3.4` becomes `-4`. The remainder then satisfies `-17 == (-4)*5 + 3`.

T7.3 MCQ: Which checks whether two names refer to the same object? A `==`; B `is`; C `=`; D `in`

B. `==` checks value equality, `is` object identity, `=` assignment, and `in` membership.

T7.4 Assertion: `2 ** 3 ** 2` equals 512. Reason: exponentiation groups right-to-left here. Decide and explain.

Both are true, and the reason explains the result: evaluate `3 ** 2 = 9`, then `2 ** 9 = 512`.

Theory self-check

C7.1 State the results of `bool("False")` and `bool("")`.

C7.2 True or false: `7 / 2` and `7 // 2` have the same type and value.

C7.3 Explain one difference between ``a += [2]`` and ``a = a + [2]`` for a list.

C7.4 Predict `3 + 4 * 2` and `(3 + 4) * 2`.

Self-check key

C7.1: True and False. Nonempty strings are truthy regardless of their spelling.

C7.2: False: 3.5 float versus 3 int.

C7.3: The former normally mutates the list, affecting aliases; the latter creates a new list and rebinds `a`.

C7.4: 11 and 14; multiplication precedes addition unless grouped.

Worked programs: plan, execute, trace

P7.1 Audit an expression

Compute a mixed expression with visible intermediate values.

Method. Exponentiate, multiply, then add; do not infer grouping from left-to-right reading.

```
power = 2 ** 3 ** 2
value = 5 + 3 * 4
print(power, value)
```

Why this result follows. $3 ** 2 \rightarrow 9$; $2 ** 9 \rightarrow 512$. $3 * 4 \rightarrow 12$; $5 + 12 \rightarrow 17$. Output: 512 17.

P7.2 Identity and equality

Show why an alias and a copy behave differently.

Method. Make one second reference to the same list and one distinct equal list.

```
a = [4, 5]
b = a
c = a.copy()
print(a == c, a is b, a is c)
b.append(6)
print(a, c)
```

Why this result follows. Before append, values a and c compare equal; a/b share identity. Appending through b changes a, not c. Outputs: True True False; [4, 5, 6] [4, 5].

P7.3 Calculate a bill

Apply a discount before tax and retain decimal arithmetic.

Method. Compute discounted subtotal once and then apply tax to that subtotal.

```
price = 200.0
discount = 0.10
tax = 0.05
subtotal = price * (1 - discount)
total = subtotal * (1 + tax)
print(f"total: {total:.2f}")
```

Why this result follows. $200 * 0.90 \rightarrow 180.00$; $180 * 1.05 \rightarrow 189.00$; output: 189.00.

Independent program practice

U7.1 Predict $2 ** 3 ** 2$, $-3 ** 2$, $(-3) ** 2$, $-17 // 5$ and $-17 \% 5$, showing grouping.

Check: Expected 512, -9, 9, -4, 3; check the quotient/remainder identity.

U7.2 Repair three independent errors: adding a number to text, converting "hello" to float, and assigning to $1 + 2$.

Check: Classify TypeError, ValueError and SyntaxError before rewriting each statement.

U7.3 Write expressions for age within 10 through 15 inclusive, a city other than Delhi, and two equal marks.

Check: Test both endpoints; use $!=$ for value inequality and $==$ for equality.

KIPS 8. Control Flow

KIPS 167-198 / Part I route: Conditional statements; For loops; While loops; Nested loops; Loop control; Printing patterns

KIPS combines explanations with traces, code repair, number problems and patterns. Check the decision before changing state, and test each loop boundary.

Theory: reason through the answer

T8.1 When should a decision use if/elif/else rather than separate if statements?

Use a chain when alternatives are mutually exclusive: only the first true branch runs. Separate if statements each test independently and may execute several bodies.

T8.2 How do break, continue and pass differ?

break exits the nearest loop; continue skips to its next iteration; pass does nothing and allows normal execution to continue.

T8.3 MCQ: Which values appear in range(5, 0, -2)? A 5,3,1; B 5,3,1,0; C 5,4,3,2,1; D none

A. The stop 0 is excluded, and each step subtracts 2.

T8.4 Assertion: A loop with `while True` is necessarily infinite. Reason: break can leave a loop. Evaluate.

The assertion is false: a reached break can terminate it. The reason is true. Also consider whether that break is reachable.

Theory self-check

C8.1 What runs in a loop else clause?

C8.2 True or false: continue updates a while counter automatically.

C8.3 How many inner-body executions occur for 3 outer iterations and 4 inner iterations?

C8.4 Why test a pattern program at one row as well as at five rows?

Self-check key

C8.1: The else body runs after normal exhaustion, but not after break.

C8.2: False. If its update is skipped, a loop can repeat forever.

C8.3: 12, if each inner loop runs four times for every outer iteration.

C8.4: The first and last row conditions can coincide at one row; that reveals boundary mistakes.

Worked programs: plan, execute, trace

P8.1 Grade with a guard

Reject invalid marks before assigning a grade.

Method. Validation is the first branch; thresholds descend so exactly one grade is chosen.

```
mark = 72
if not 0 <= mark <= 100:
    result = "Invalid"
elif mark >= 80:
    result = "A"
elif mark >= 60:
    result = "B"
else:
    result = "C"
print(result)
```

Why this result follows. 72 passes validation; 72 >= 80 is false; 72 >= 60 is true. Output: B.

P8.2 Prime search

Print primes from 2 through 15.

Method. For each candidate, try possible divisors; a found factor breaks the inner loop. Its else runs only when no factor was found.

```
for n in range(2, 16):
    for d in range(2, n):
        if n % d == 0:
            break
    else:
        print(n, end=" ")
print()
```

Why this result follows. 2 has no trial divisors, so prints; 4 breaks at d=2; 5 finds none. Output: 2 3 5 7 11 13.

P8.3 Descending row counts

Print four rows containing 4, 3, 2 and 1 stars.

Method. The outer loop chooses a row length; multiplication constructs exactly that many stars.

```
for width in range(4, 0, -1):
    print("*" * width)
```

Why this result follows. width: 4 -> ****; 3 -> ***; 2 -> **; 1 -> *. The stop 0 is excluded.

Independent program practice

U8.1 Read positive numbers until -1, then report count and mean; handle no numbers.

Check: Trace 4, 6, -1 -> count 2, mean 5.0; -1 alone -> no mean.

U8.2 Write an Armstrong-number test for a nonnegative integer, using its digit count as the exponent.

Check: Check 0, 153, 370 and 154; avoid assuming every input has three digits.

U8.3 Print a centred pyramid of 1, 3, 5 and 7 stars and annotate spaces and stars per row.

Check: For row *r* from 1 to 4: leading spaces 4-*r*, stars 2*r*-1; test a one-row version.

KIPS 9. String Manipulation

KIPS 199-223 / Part I route: Strings; For loops; While loops

KIPS asks for indexing, methods, output tracing, invalid mutation repair and multi-step text transformation. Choose a small input and record the string after each step.

Theory: reason through the answer

T9.1 Why does `s[0] = "X"` fail for a string?

Strings are immutable. Build a new string such as `"X" + s[1:]` and rebind the name; the old object is not changed.

T9.2 How do `find` and `index` behave when a substring is missing?

`find` returns -1; `index` raises `ValueError`. Check membership or use `find` when absence is expected.

T9.3 MCQ: What is `"Python"[-2]`? A t; B o; C n; D error

B. Negative indices count back from the end: -1 is `n`, -2 is `o`.

T9.4 Assertion: `strip()` removes all spaces inside a sentence. Reason: `strip()` removes characters from the ends. Evaluate.

The assertion is false, and the reason is true. To normalise internal whitespace, split into words and join them.

Theory self-check

C9.1 What does `partition(":")` return when the separator exists?

C9.2 True or false: `s[100:200]` always raises `IndexError` for a short string.

C9.3 Predict `"banana".find("ana")` and `"banana".count("ana")`.

C9.4 Why do `lstrip` and `rstrip` differ?

Self-check key

C9.1: A three-part tuple: text before, the separator, and text after its first occurrence.

C9.2: False. A slice clips to the available extent; `s[100]` would raise `IndexError`.

C9.3: 1 and 1; `count` finds non-overlapping matches.

C9.4: They remove a chosen set of characters only from the left or right edge, respectively.

Worked programs: plan, execute, trace

P9.1 Count internal whitespace

Count whitespace between words while ignoring leading and trailing whitespace.

Method. Trim edges, then count whitespace characters in the remaining text. This counts every interior whitespace character.

```
text = "  red  blue green  "
inside = text.strip()
count = sum(ch.isspace() for ch in inside)
print(count)
```

Why this result follows. After strip: "red blue green"; two spaces then one space -> 3.

P9.2 Sum digits embedded in text

Ignore letters and sum every decimal digit in an input string.

Method. Traverse one character at a time; convert only when isdecimal is true.

```
text = "a1b24"
total = 0
for ch in text:
    if ch.isdecimal():
        total += int(ch)
print(total)
```

Why this result follows. Digits encountered: 1 -> total 1; 2 -> 3; 4 -> 7. Output: 7.

P9.3 Swap two punctuation marks

Exchange commas and dots without changing other characters.

Method. Read the original character; use if/elif so a replacement is not replaced again.

```
text = "a,b.c"
result = ""
for ch in text:
    if ch == ",":
        result += "."
    elif ch == ".":
        result += ","
    else:
        result += ch
print(result)
```

Why this result follows. a stays a; comma -> dot; b stays b; dot -> comma; c stays c. Output: a.b,c.

Independent program practice

U9.1 Ask for a full name and display its whole form, first character, first five characters, and characters at positions 4 to 6 (one-based).

Check: Handle empty and short names; distinguish s[3:6] from s[4:6].

U9.2 Capitalise the first and last letters of each word, preserving single spaces between output words.

Check: Test "a red fox" and an empty string; decide how one-character words behave.

U9.3 Create a small text transformation that changes vowels and S/s according to an explicit mapping.

Check: Trace each original character exactly once; test uppercase/lowercase and repeated letters.

KIPS 10. List Manipulation

KIPS 224-248 / Part I route: Lists; Nested loops; Errors

List questions test operations, aliasing, nested indexing and small algorithms. Show the intermediate list after each mutation rather than trusting a final printout.

Theory: reason through the answer

T10.1 How do append and extend differ?

append adds its argument as one item; extend iterates over its argument and adds each item. Thus `[1].append([2, 3])` produces `[1, [2, 3]]`, while extend produces `[1, 2, 3]`.

T10.2 How do sort and sorted differ?

`list.sort()` rearranges the existing list and returns `None`. `sorted(list_value)` returns a new list, leaving the input order intact.

T10.3 MCQ: If `a = [10, 20, 30, 40]`, what is `a[-2]`? A 10; B 20; C 30; D 40

C. -1 refers to the final element, so -2 refers to 30.

T10.4 Assertion: ``[[[]] * 3`` gives three independent inner lists. Reason: list repetition copies references. Evaluate.

The assertion is false; the reason is true. Repetition creates three references to the same inner list. Use a comprehension to create independent rows.

Theory self-check

C10.1 Predict `len([None] * 5)`.

C10.2 True or false: `a[:]` makes a deep copy of every nested object.

C10.3 What does `pop(1)` return, and what else does it do?

C10.4 Explain why ``result = a.reverse()`` is usually a mistake.

Self-check key

C10.1: 5. `None` is an element; the list is not empty.

C10.2: False. It copies the outer list; inner objects may still be shared.

C10.3: It returns the item at index 1 and removes it from the list.

C10.4: `reverse` mutates `a` and returns `None`; use `a.reverse()`; `result = a`, or `reversed/slicing` as appropriate.

Worked programs: plan, execute, trace

P10.1 Move repeats to the end

Keep first occurrences in their original order, then place later repeats after them.

Method. Track seen values; append to one of two lists, then concatenate. This handles hashable values.

```
items = [3, 1, 3, 2, 1]
seen = set()
first = []
repeats = []
for value in items:
    if value in seen:
        repeats.append(value)
    else:
        seen.add(value)
        first.append(value)
print(first + repeats)
```

Why this result follows. 3 -> first [3]; 1 -> [3,1]; repeated 3 -> repeats [3]; 2 -> first [3,1,2]; repeated 1 -> [3,1]. Output: [3, 1, 2, 3, 1].

P10.2 Filter a nested list

Select short names from a table of planet names.

Method. The outer loop visits rows; the inner loop visits names. Check the length boundary explicitly.

```
rows = [["Mars", "Jupiter"], ["Venus", "Neptune"]]
short = []
for row in rows:
    for name in row:
        if len(name) <= 6:
            short.append(name)
print(short)
```

Why this result follows. Mars 4 -> include; Jupiter 7 -> skip; Venus 5 -> include; Neptune 7 -> skip. Output: ['Mars', 'Venus'].

P10.3 Find bounds without built-ins

Find minimum and maximum in a nonempty numeric list.

Method. Initialise both from the first element, then compare each remaining value. This works for all-negative data.

```
values = [-4, -9, -2]
smallest = largest = values[0]
for value in values[1:]:
    if value < smallest:
        smallest = value
    if value > largest:
        largest = value
print(smallest, largest)
```

Why this result follows. Start (-4,-4); -9 -> (-9,-4); -2 -> (-9,-2). Output: -9 -2.

Independent program practice

U10.1 For a rectangular nested list, calculate each row sum and the total of column 2 (index 1).

Check: Test `[[2, 4], [1, 7]]` -> row totals `[6, 8]`, column total `11`; handle an empty table by policy.

U10.2 Merge two input lists into a new list, then count occurrences of a chosen item without changing either source.

Check: Test `[1,2]` and `[2,3]`, target `2` -> `[1,2,2,3]`, count `2`.

U10.3 Trace a list after a slice replacement, remove, append and sort; predict the return values.

Check: Use a copy at each stage; include a case where remove cannot find its target.

KIPS 11. Tuples

KIPS 249-267 / Part I route: Tuples; Lists; For loops

The questions combine immutable sequence semantics, single-element syntax, slicing, conversion and records stored inside nested tuples.

Theory: reason through the answer

T11.1 Why are (5) and (5,) different?

(5) is the integer 5 in grouping parentheses. The comma makes (5,) a one-element tuple.

T11.2 Can a tuple be modified by converting to a list?

The original tuple cannot be modified. Convert to a list, edit the list, and create a new tuple, then rebind a name. Other references to the old tuple remain unchanged.

T11.3 MCQ: What type does `sorted((3, 1, 2))` return? A tuple; B list; C set; D iterator

B. sorted returns a new list. Wrap it with `tuple(...)` when a tuple is needed.

T11.4 Assertion: `t.index(9)` raises `ValueError` when 9 is absent. Reason: index returns -1 for absence. Evaluate.

The assertion is true; the reason is false. The -1 sentinel is used by `str.find`, not `tuple.index`.

Theory self-check

C11.1 Explain tuple packing and unpacking with two values.

C11.2 True or false: `t[0] = 7` is valid when `t` is a tuple of integers.

C11.3 Predict `(1, 2, 3)[-2:]` and its type.

C11.4 Can a tuple containing a list be a dictionary key?

Self-check key

C11.1: `a, b = 2, 3` packs/evaluates two RHS values and binds `a=2, b=3`; `(2,3)` is a tuple.

C11.2: False: `TypeError`; create a new tuple.

C11.3: `(2, 3)`, a tuple.

C11.4: No: its list element is unhashable, so the tuple is unhashable.

Worked programs: plan, execute, trace

P11.1 Totals in nested records

Calculate two subject totals from a tuple of student records.

Method. Unpack each record and add its two marks; keep names attached to totals.

```
records = (("Asha", 35, 40), ("Ravi", 42, 38))
for name, first, second in records:
    print(name, first + second)
```

Why this result follows. Asha: $35+40 \rightarrow 75$; Ravi: $42+38 \rightarrow 80$. Output lines: Asha 75; Ravi 80.

P11.2 Remove by constructing a new tuple

Remove the item at a valid index without mutating the original tuple.

Method. Join the slice before the index to the slice after it.

```
original = (2, 3, 4, 5, 6)
index = 2
updated = original[:index] + original[index + 1:]
print(original, updated)
```

Why this result follows. Before index 2 $\rightarrow (2,3)$; after $\rightarrow (5,6)$; new tuple $\rightarrow (2,3,5,6)$. Original retains 4.

P11.3 Sum a tuple with while

Traverse by index and update an accumulator.

Method. Test `index < len(values)` before reading; increment exactly once.

```
values = (4, 7, 2)
index = 0
total = 0
while index < len(values):
    total += values[index]
    index += 1
print(total)
```

Why this result follows. `(index,total)`: $(0,0) \rightarrow (1,4) \rightarrow (2,11) \rightarrow (3,13)$; stop at 3. Output: 13.

Independent program practice

U11.1 Insert 30 at index 2 in `(10, 20, 40, 50)` by constructing a new tuple.

Check: Expected `(10, 20, 30, 40, 50)`; original remains unchanged.

U11.2 Read or define a nonempty tuple of costs; print minimum, maximum and sum without changing it.

Check: Test (12.5, 7.0, 12.5) -> 7.0, 12.5, 32.0; state your empty-input policy.

U11.3 Search a tuple for a number and report its first index if present, otherwise a clear message.

Check: Test repeats and absence; avoid calling index before checking membership.

KIPS 12. Dictionaries

KIPS 268-290 / Part I route: Dictionaries; Lists; For loops

KIPS asks for definitions, method choice, key errors, nested employee records and programs that transform or aggregate mappings.

Theory: reason through the answer

T12.1 What is a dictionary key, and why can a list not be one?

Keys identify entries and must be hashable. A list can change its value and is unhashable; a tuple of hashable items can be a key.

T12.2 How do `d[key]` and `d.get(key)` differ for a missing key?

Indexing raises `KeyError`. `get` returns `None` by default or a caller-supplied default. If `None` could be a stored value, use `key` in `d` to distinguish absence.

T12.3 MCQ: Which iterates over pairs? A `d.keys()`; B `d.values()`; C `d.items()`; D `d.clear()`

C. `items` supplies (key, value) pairs for unpacking in a loop.

T12.4 Assertion: Duplicate keys coexist as separate dictionary entries.

Reason: a later value for an equal key replaces the earlier one. Evaluate.

The assertion is false and the reason is true. A literal such as `{1: "a", 1: "b"}` has one key with value "b".

Theory self-check

C12.1 What does `popitem` remove in current Python?

C12.2 True or false: `key` in `d` searches values.

C12.3 Why can `dict.fromkeys(["a", "b"], [])` be dangerous?

C12.4 When should a frequency update use `get`?

Self-check key

C12.1: The most recently inserted remaining key-value pair; it also returns that pair.

C12.2: False. It checks keys; use `key` in `d.values()` for values.

C12.3: Both keys receive the same list object. Mutating it through either key affects both.

C12.4: When the key may not exist: `counts[ch] = counts.get(ch, 0) + 1`.

Worked programs: plan, execute, trace

P12.1 Update one nested employee

Find a named employee and change only that employee's salary.

Method. Iterate over inner records; test the name before assigning the salary.

```
staff = {
    "e1": {"name": "Ravi", "salary": 7000},
    "e2": {"name": "Sandeep", "salary": 6500},
}
for record in staff.values():
    if record["name"] == "Sandeep":
        record["salary"] = 8500
print(staff["e2"]["salary"])
```

Why this result follows. e1 name differs -> unchanged; e2 matches -> salary 6500 to 8500. Output: 8500.

P12.2 Rename a key

Rename city to location without discarding its value.

Method. pop returns the former value and removes the old key; assign that value to the new key.

```
person = {"name": "Keya", "city": "Pune"}
person["location"] = person.pop("city")
print(person)
```

Why this result follows. pop city -> "Pune"; city removed; location added. Output: {'name': 'Keya', 'location': 'Pune'}.

P12.3 Two largest distinct values

Report the two highest distinct numeric values, with a stated policy for insufficient data.

Method. Make a set of values, sort descending, and require at least two distinct values.

```
scores = {"A": 84, "B": 91, "C": 91, "D": 76}
distinct = sorted(set(scores.values()), reverse=True)
if len(distinct) >= 2:
    print(distinct[:2])
else:
    print("Need two distinct values")
```

Why this result follows. Values [84,91,91,76] -> distinct {84,91,76} -> descending [91,84,76]. Output: [91, 84].

Independent program practice

U12.1 Count letters in a string using a dictionary, ignoring spaces and case.

Check: Check "A a b" -> {"a": 2, "b": 1}; test empty text.

U12.2 Merge two stock dictionaries by adding quantities for common keys.

Check: Test {"pen":2} and {"pen":3,"pad":1} -> {"pen":5,"pad":1}; do not mutate inputs.

U12.3 Create a record, add a field, change one value, delete a field, and finally clear it; print after each step.

Check: Trace keys and values at each step; compare del, pop and clear.

KIPS 13. Introduction to Python Modules

KIPS 291-299 / Part I route: Modules; The math module; The random module; The statistics module

Module exercises focus on import syntax, function selection, random endpoint reasoning and short combined programs. State the possible range of a random result rather than one accidental draw.

Theory: reason through the answer

T13.1 How do import math and from math import sqrt change available names?

The first binds math, so call math.sqrt(9). The second binds sqrt directly, so call sqrt(9). Mixing the two call styles without both bindings gives NameError.

T13.2 What are the possible values of randrange(7, 18)?

Integers 7 through 17. The stop is excluded. randint(7, 18) would include 18.

T13.3 MCQ: Which module supplies median? A math; B random; C statistics; D builtins

C. Import statistics and call statistics.median(values), or import the name selectively.

T13.4 Assertion: random.randrange(4, 9) may return 9. Reason: the stop argument is excluded. Evaluate.

The assertion is false and the reason is true. The possible outputs are 4, 5, 6, 7 and 8.

Theory self-check

C13.1 What is the difference between math.floor(-2.1) and math.fabs(-2.1)?

C13.2 True or false: random.randint(2, 9) can return both 2 and 9.

C13.3 Why can naming your own file random.py break import random?

C13.4 Is random suitable for a real authentication OTP?

Self-check key

C13.1: floor gives -3 (int); fabs gives 2.1 (float). They perform different operations.

C13.2: True. Both endpoints are included.

C13.3: The local file can shadow the standard-library module.

C13.4: No. Use the secrets module for security-sensitive tokens; random is for classroom simulation.

Worked programs: plan, execute, trace

P13.1 Circle area with math.pi

Calculate a circle area using the module constant.

Method. Import math, validate a nonnegative radius in real use, and apply πr^2 .

```
import math
radius = 3
area = math.pi * radius ** 2
print(f"{area:.2f}")
```

Why this result follows. $3^2 \rightarrow 9$; $9 * \pi \rightarrow$ about 28.2743; two-decimal output: 28.27.

P13.2 Reason about random bounds

Generate two classroom outcomes and assert invariant bounds.

Method. Use a local Random instance so the example does not depend on global random state. The important answers are its possible values.

```
import random
rng = random.Random(7)
die = rng.randint(1, 6)
step = rng.randrange(20, 50, 10)
assert 1 <= die <= 6
assert step in (20, 30, 40)
print("bounds verified")
```

Why this result follows. randint may give 1..6; randrange may give 20,30,40, never 50. Assertions hold; output: bounds verified.

P13.3 Compare the centre of a sample

Use a statistics module function after importing it.

Method. A mean adds and divides; median sorts and selects the centre. An outlier changes them differently.

```
import statistics
values = [2, 4, 6, 20]
print(statistics.mean(values))
print(statistics.median(values))
```

Why this result follows. Sum 32 / 4 $\rightarrow$ mean 8; sorted middle pair 4 and 6 $\rightarrow$ median 5.0. Outputs: 8, then 5.0.

Independent program practice

U13.1 List all outputs of randrange(5, 20, 5) and compare with randint(5, 20).

Check: First gives 5,10,15; randint can give every integer from 5 through 20.

U13.2 Repair calls to an unimported sqrt, random.randomint, and randrange with float endpoints.

Check: Use a matching import style, randint, and integer range arguments; run each corrected line.

U13.3 Generate a six-digit classroom number and explain its inclusive lower and upper bounds.

Check: A suitable randint range is 100000 through 999999; real OTPs require secrets.

Mixed checkpoint: can you transfer the ideas?

Attempt these without opening the preceding answers. Give a reason, not just a final value. Then use the check notes to locate the chapter to revisit.

M1. Explain why a bare `4 + 5` is visible in a shell but not as output from a saved script.

Check: The interactive shell displays expression results; a script needs `print` for visible output.

M2. Predict the types after `x = "8"` and `y = int(x)`, then explain why `x + y` fails.

Check: `x` remains `str`, `y` is `int`; adding unlike types this way raises `TypeError`.

M3. Trace `-9 // 4`, `-9 % 4`, and `2 ** 3 ** 2`.

Check: `-3`, `3`, `512`. Check `-9 == (-3)*4 + 3` and exponentiation grouping.

M4. Write a loop that counts the digits in `"a1b20"` and a test for the empty string.

Check: Use character tests and a counter; the sample has three digits and empty text has zero.

M5. Explain what changes after `b = a`; `b.append(7)` when `a` is a list, and after `t += (7,)` when `t` is a tuple.

Check: The list alias sees the mutation; tuple concatenation constructs and binds a new tuple.

M6. From a dictionary of marks, report the maximum mark and the names tied at it; choose a module for the mean.

Check: Find `max(values)`, collect every matching key, and use `statistics.mean` for a nonempty sample.

If a check fails, trace the smallest counterexample first, rerun it in Python 3, and explain which rule changed your prediction.