FOUNDATION LESSON

WEEK 3 · LESSON 1

Why functions matter: define once, call when needed

A named operation: define once, call when needed.

THE MENTAL MODEL

CODE WALKTHROUGH W03-L01-S02

```
01  def show_start_message():
02      print("Starting daily checks")
```

WHAT TO TRACK

Definition: write the checklist.
Call: carry it out.
Body: indented instructions.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

MINIMAL EXAMPLE AND PREDICTION

CODE WALKTHROUGH W03-L01-S03

```
01  def show_start_message():
02      print("Starting daily checks")
03
04
05  print("Before call")
06  show_start_message()
07  print("After call")
```

WHAT TO TRACK

Trace execution from top to bottom.
The body runs at the call.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

RESULT AND LINE-BY-LINE REASONING

CODE WALKTHROUGH W03-L01-S04

```
01  def show_start_message():
02      print("Starting daily checks")
03
04
05  print("Before call")
06  show_start_message()
07  print("After call")
```

WHAT TO TRACK

A definition does not print.
The call executes the body.

VALIDATE THE RESULT

```
Before call
Starting daily checks
After call
```

BUSINESS APPLICATION

CODE WALKTHROUGH W03-L01-S05

```
01  def show_invoice_checks():
02      amounts = [120, 2500, 0]
03
04      for amount in amounts:
05          if amount <= 0:
06              print(amount, "invalid")
07          elif amount >= 2000:
08              print(amount, "review")
09          else:
10              print(amount, "standard")
11
12
13  print("Run 1")
14  show_invoice_checks()
15
16  print("Run 2")
17  show_invoice_checks()
```

WHAT TO TRACK

Amount ≤ 0: invalid.
Amount ≥ 2000: review.
Other positive: standard.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

PREDICT REPEATED CALLS

MENTAL MODEL

Predict repeated calls

Three amounts, two calls.
How many classification lines?

BUSINESS RESULT AND REASONING

CODE WALKTHROUGH W03-L01-S07

```
01  def show_invoice_checks():
02      amounts = [120, 2500, 0]
03
04      for amount in amounts:
05          if amount <= 0:
06              print(amount, "invalid")
07          elif amount >= 2000:
08              print(amount, "review")
09          else:
10              print(amount, "standard")
11
12
13  print("Run 1")
14  show_invoice_checks()
15
16  print("Run 2")
17  show_invoice_checks()
```

WHAT TO TRACK

Each call runs the same procedure.
Three records × two calls.

VALIDATE THE RESULT

```
Run 1
120 standard
2500 review
0 invalid
Run 2
120 standard
2500 review
0 invalid
```

DELIBERATE ERROR: CALLING TOO SOON

CODE WALKTHROUGH W03-L01-S08

```
01  show_end_message()
02
03
04  def show_end_message():
05      print("Checks complete")
```

WHAT TO TRACK

The call executes before the definition.
Predict the failure.

VALIDATE THE RESULT

```
NameError: name 'show_end_message' is not defined
```

REPAIR THE ORDER

CODE WALKTHROUGH W03-L01-S09

```
01  def show_end_message():
02      print("Checks complete")
03
04
05  show_end_message()
```

WHAT TO TRACK

Establish the function before calling it.

VALIDATE THE RESULT

```
Checks complete
```

LEARNER MODIFICATION

LEARNER PRACTICE

Learner modification

Use [999, 1000, 0].
Review threshold: 1000.
Keep one definition and call it once.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND ENDING

Recap and next step

Definition establishes an operation.
The indented body runs on a call.
One definition supports repeated calls.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 2

Parameters, arguments and safe defaults

Inputs, binding and defaults without hidden shared state.

EXPLAIN BINDING BEFORE SYNTAX

MENTAL MODEL

Explain binding before syntax

Parameter: a name in the definition.
Argument: a value supplied by a call.
Binding associates the two.

MINIMAL EXAMPLE AND PREDICTION

CODE WALKTHROUGH W03-L02-S03

```
01  def show_stock(item, quantity):
02      print(item, quantity)
03
04
05  show_stock("Pens", 12)
06  show_stock(quantity=4, item="Folders")
```

WHAT TO TRACK

Positional arguments bind in order.
Keyword arguments bind by name.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

RESULT AND BINDING TRACE

CODE WALKTHROUGH W03-L02-S04

```
01  def show_stock(item, quantity):
02      print(item, quantity)
03
04
05  show_stock("Pens", 12)
06  show_stock(quantity=4, item="Folders")
```

WHAT TO TRACK

Keyword order does not change the body.

VALIDATE THE RESULT

```
Pens 12
Folders 4
```

BUSINESS APPLICATION WITH A DEFAULT

CODE WALKTHROUGH W03-L02-S05

```
01  def show_delivery_total(order_total, delivery_fee=5):
02      print("Total:", order_total + delivery_fee)
03
04
05  show_delivery_total(100)
06  show_delivery_total(100, 0)
07  show_delivery_total(delivery_fee=8, order_total=100)
```

WHAT TO TRACK

Total = amount + fee.
Default fee: 5.
Explicit 0: free delivery.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

DEFAULT RESULT AND REASONING

CODE WALKTHROUGH W03-L02-S06

```
01  def show_delivery_total(order_total, delivery_fee=5):
02      print("Total:", order_total + delivery_fee)
03
04
05  show_delivery_total(100)
06  show_delivery_total(100, 0)
07  show_delivery_total(delivery_fee=8, order_total=100)
```

WHAT TO TRACK

Omitted, zero and eight are different inputs.

VALIDATE THE RESULT

```
Total: 105
Total: 100
Total: 108
```

DELIBERATE DESIGN MISTAKE: MUTABLE DEFAULT

CODE WALKTHROUGH W03-L02-S07

```
01  def record_review(invoice_id, reviewed=[]):
02      reviewed.append(invoice_id)
03      print(reviewed)
04
05
06  record_review("INV-01")
07  record_review("INV-02")
```

WHAT TO TRACK

Intention: each omitted list is fresh.
Does the default implement it?

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

THE RETAINED-STATE RESULT

CODE WALKTHROUGH W03-L02-S08

```
01  def record_review(invoice_id, reviewed=[]):
02      reviewed.append(invoice_id)
03      print(reviewed)
04
05
06  record_review("INV-01")
07  record_review("INV-02")
```

WHAT TO TRACK

The same default list retains mutations.

VALIDATE THE RESULT

```
['INV-01']
['INV-01', 'INV-02']
```

REPAIR WITH NONE

CODE WALKTHROUGH W03-L02-S09

```
01  def record_review(invoice_id, reviewed=None):
02      if reviewed is None:
03          reviewed = []
04
05      reviewed.append(invoice_id)
06      print(reviewed)
07
08
09  record_review("INV-01")
10  record_review("INV-02")
```

WHAT TO TRACK

None marks an omitted list.
Create a fresh list inside the call.

VALIDATE THE RESULT

```
['INV-01']
['INV-02']
```

GUIDED MODIFICATION

LEARNER PRACTICE

Guided modification

Add record_review("INV-03").
Predict its list after the None repair.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND ENDING

Recap and next step

Parameters name inputs; arguments supply them.
Defaults are evaluated at definition time.
A mutable default can retain state.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 3

Return useful results and understand local names

Capture results; distinguish local names from shared objects.

EXPLAIN THE RESULT CHANNEL

MENTAL MODEL

Explain the result channel

Print displays information.
Return supplies a reusable result.
No explicit return supplies None.

MINIMAL COMPARISON AND PREDICTION

CODE WALKTHROUGH W03-L03-S03

```
01  def display_profit(revenue, cost):
02      print(revenue - cost)
03
04
05  def calculate_profit(revenue, cost):
06      return revenue - cost
07
08
09  shown = display_profit(1200, 800)
10  saved = calculate_profit(1200, 800)
11
12  print("Display result:", shown)
13  print("Saved result:", saved)
14  print("After reserve:", saved - 100)
```

WHAT TO TRACK

Capture each call’s result.
Distinguish display from return.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

RESULT AND REASONING

CODE WALKTHROUGH W03-L03-S04

```
01  def display_profit(revenue, cost):
02      print(revenue - cost)
03
04
05  def calculate_profit(revenue, cost):
06      return revenue - cost
07
08
09  shown = display_profit(1200, 800)
10  saved = calculate_profit(1200, 800)
11
12  print("Display result:", shown)
13  print("Saved result:", saved)
14  print("After reserve:", saved - 100)
```

WHAT TO TRACK

The displayed 400 is not assigned to shown.
The returned 400 is assigned to

VALIDATE THE RESULT

```
400
Display result: None
Saved result: 400
After reserve: 300
```

CHAPTER BRIDGE: LOCAL WORKING NAMES

MENTAL MODEL

Chapter bridge: local working names

Local names belong to a call.
Names and objects are different.
Arguments can refer to shared objects.

BUSINESS APPLICATION: A RETURNED ACCUMULATOR

CODE WALKTHROUGH W03-L03-S06

```
01  def total_positive(amounts):
02      total = 0
03
04      for amount in amounts:
05          if amount > 0:
06              total += amount
07
08      return total
09
10
11  total = 999
12  accepted_total = total_positive([120, -20, 2500, 0])
13
14  print("Accepted total:", accepted_total)
15  print("Caller total:", total)
16  print("Empty batch:", total_positive([]))
```

WHAT TO TRACK

Include positive amounts only.
Empty batch total: 0.
Read without mutating.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

LOCAL-SCOPE RESULT AND LIFETIME

CODE WALKTHROUGH W03-L03-S07

```
01  def total_positive(amounts):
02      total = 0
03
04      for amount in amounts:
05          if amount > 0:
06              total += amount
07
08      return total
09
10
11  total = 999
12  accepted_total = total_positive([120, -20, 2500, 0])
13
14  print("Accepted total:", accepted_total)
15  print("Caller total:", total)
16  print("Empty batch:", total_positive([]))
```

WHAT TO TRACK

The local total and caller total differ.
Each call starts its own

VALIDATE THE RESULT

```
Accepted total: 2620
Caller total: 999
Empty batch: 0
```

MUTATION BOUNDARY AND PREDICTION

CODE WALKTHROUGH W03-L03-S08

```
01  def adjust_draft(amount, amounts):
02      amount = 0
03      amounts.append(0)
04      print("Inside:", amount, amounts)
05
06
07  amount = 100
08  amounts = [10, 20]
09
10  adjust_draft(amount, amounts)
11
12  print("Outside:", amount, amounts)
```

WHAT TO TRACK

Rebinding a local name differs from mutation.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

EXPLAIN THE DIFFERENT EFFECTS

CODE WALKTHROUGH W03-L03-S09

```
01  def adjust_draft(amount, amounts):
02      amount = 0
03      amounts.append(0)
04      print("Inside:", amount, amounts)
05
06
07  amount = 100
08  amounts = [10, 20]
09
10  adjust_draft(amount, amounts)
11
12  print("Outside:", amount, amounts)
```

WHAT TO TRACK

The caller’s integer binding stays 100.
The shared list gains a zero.

VALIDATE THE RESULT

```
Inside: 0 [10, 20, 0]
Outside: 100 [10, 20, 0]
```

MAKE A SEPARATE FLAT-LIST DRAFT

CODE WALKTHROUGH W03-L03-S10

```
01  def draft_with_marker(amounts):
02      draft = amounts.copy()
03      draft.append(0)
04      return draft
05
06
07  original = [10, 20]
08  draft = draft_with_marker(original)
09
10  print("Original:", original)
11  print("Draft:", draft)
```

WHAT TO TRACK

Copy the flat outer list before append.
Return the draft to the caller.

VALIDATE THE RESULT

```
Original: [10, 20]
Draft: [10, 20, 0]
```

GUIDED DEBUGGING

CODE WALKTHROUGH W03-L03-S11

```
01  def classify_amount(amount):
02      if amount <= 0:
03          return "invalid"
04
05      if amount >= 2000:
06          return "review"
07
08
09  print(classify_amount(2500))
10  print(classify_amount(100))
```

WHAT TO TRACK

What happens when a path reaches the end?
Add the missing ordinary-case

VALIDATE THE RESULT

```
review
None
```

RECAP AND ENDING

Recap and next step

Print displays; return supplies a result.
Local rebinding leaves caller bindings alone.
Shared mutations can remain visible.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 4

Document a function’s contract

Docstrings, basic type hints and validation

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

Think of the contract as a label on a measuring instrument: it says what goes in and what comes out. The label does not physically stop misuse. A runtime check is executable code that can reject an unsuitable value. A docstring is the first string in the function body. A type hint is an annotation describing an intended type.

A DOCUMENTED CALCULATION

CODE WALKTHROUGH W03-L04-S03

```
01  def revenue(quantity: int, price: int) -> int:
02      """Return quantity times price in whole teaching units."""
03      return quantity * price
04
05  print(revenue(3, 20))
```

WHAT TO TRACK

Parameters: quantity and price.
Result: integer revenue.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

A DOCUMENTED CALCULATION — RESULT

CODE WALKTHROUGH W03-L04-S04

```
01  def revenue(quantity: int, price: int) -> int:
02      """Return quantity times price in whole teaching units."""
03      return quantity * price
04
05  print(revenue(3, 20))
```

WHAT TO TRACK

Parameters: quantity and price.
Result: integer revenue.

VALIDATE THE RESULT

60

A HINT DOES NOT ENFORCE THE INPUT

CODE WALKTHROUGH W03-L04-S05

```
01  def revenue(quantity: int, price: int) -> int:
02      """Return quantity times price in whole teaching units."""
03      return quantity * price
04
05  print(revenue("3", 2))
```

WHAT TO TRACK

Observe what Python actually does with a string and an integer.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

A HINT DOES NOT ENFORCE THE INPUT — RESULT

CODE WALKTHROUGH W03-L04-S06

```
01  def revenue(quantity: int, price: int) -> int:
02      """Return quantity times price in whole teaching units."""
03      return quantity * price
04
05  print(revenue("3", 2))
```

WHAT TO TRACK

Observe what Python actually does with a string and an integer.

VALIDATE THE RESULT

33

VALIDATE A WHOLE-NUMBER QUANTITY

CODE WALKTHROUGH W03-L04-S07

```
01  def checked_revenue(quantity: int, price: int):
02      if type(quantity) is not int or type(price) is not int:
03          return None
04      if quantity < 0 or price < 0:
05          return None
06      return quantity * price
07
08  print(checked_revenue(3, 20))
09  print(checked_revenue("3", 20))
10  print(checked_revenue(0, 20))
```

WHAT TO TRACK

Policy: non-negative exact integers;
None marks invalid input.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

VALIDATE A WHOLE-NUMBER QUANTITY — RESULT

CODE WALKTHROUGH W03-L04-S08

```
01  def checked_revenue(quantity: int, price: int):
02      if type(quantity) is not int or type(price) is not int:
03          return None
04      if quantity < 0 or price < 0:
05          return None
06      return quantity * price
07
08  print(checked_revenue(3, 20))
09  print(checked_revenue("3", 20))
10  print(checked_revenue(0, 20))
```

WHAT TO TRACK

Policy: non-negative exact integers; None marks invalid input.

VALIDATE THE RESULT

```
60
None
0
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Modify checked_revenue so a negative quantity is invalid but zero quantity is still valid. Test quantity zero, minus one and the string two. Explain why a type check comes before a numeric comparison.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

A docstring explains the contract. Basic type hints describe intended types; they do not enforce them during ordinary execution. Runtime validation checks both type and value. In the next lesson, we make invalid inputs easier to diagnose using meaningful errors.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 5

Handle failures without hiding them

try/except and meaningful errors

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

An exception interrupts the ordinary flow of a calculation. Raising an exception reports that the operation cannot fulfil its contract. A try block marks an operation whose known failure we can handle. An except clause names the specific failure. Catching everything and continuing silently is like covering a warning light: the underlying problem still exists.

CONVERT ONE SUPPLIED VALUE

CODE WALKTHROUGH W03-L05-S03

```
01  text = "twelve"
02  try:
03      quantity = int(text)
04  except ValueError as error:
05      print("Quantity must be whole-number text")
06      print(type(error).__name__)
```

WHAT TO TRACK

Catch the anticipated conversion failure only.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

CONVERT ONE SUPPLIED VALUE — RESULT

CODE WALKTHROUGH W03-L05-S04

```
01  text = "twelve"
02  try:
03      quantity = int(text)
04  except ValueError as error:
05      print("Quantity must be whole-number text")
06      print(type(error).__name__)
```

WHAT TO TRACK

Catch the anticipated conversion failure only.

VALIDATE THE RESULT

```
Quantity must be
whole-number text
ValueError
```

RAISE ERRORS AT THE CONTRACT BOUNDARY

CODE WALKTHROUGH W03-L05-S05

```
01  def line_total(quantity, price):
02      if type(quantity) is not int or type(price) is not int:
03          raise TypeError("quantity and price must be integers")
04      if quantity < 0 or price < 0:
05          raise ValueError("quantity and price must be non-negati
↪   ve")
06      return quantity * price
07
08  print(line_total(3, 20))
09  print(line_total(-1, 20))
```

WHAT TO TRACK

Integers required; non-negative values required; zero allowed.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

RAISE ERRORS AT THE CONTRACT BOUNDARY — RESULT

CODE WALKTHROUGH W03-L05-S06

```
01  def line_total(quantity, price):
02      if type(quantity) is not int or type(price) is not int:
03          raise TypeError("quantity and price must be integers")
04      if quantity < 0 or price < 0:
05          raise ValueError("quantity and price must be non-negati
↪   ve")
06      return quantity * price
07
08  print(line_total(3, 20))
09  print(line_total(-1, 20))
```

WHAT TO TRACK

Integers required; non-negative values required; zero allowed.

VALIDATE THE RESULT

```
60
ValueError: quantity and
price must be non-negative
```

HANDLE A KNOWN RECORD ERROR

CODE WALKTHROUGH W03-L05-S07

```
01  def positive_quantity(value):
02      quantity = int(value)
03      if quantity < 0:
04          raise ValueError("quantity must not be negative")
05      return quantity
06
07  for text in ["3", "bad", "-1"]:
08      try:
09          print("Accepted:", positive_quantity(text))
10      except ValueError as error:
11          print("Rejected:", str(error))
```

WHAT TO TRACK

Conversion and domain errors remain visible; records processed separately.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

HANDLE A KNOWN RECORD ERROR — RESULT

CODE WALKTHROUGH W03-L05-S08

```
01  def positive_quantity(value):
02      quantity = int(value)
03      if quantity < 0:
04          raise ValueError("quantity must not be negative")
05      return quantity
06
07  for text in ["3", "bad", "-1"]:
08      try:
09          print("Accepted:", positive_quantity(text))
10      except ValueError as error:
11          print("Rejected:", str(error))
```

WHAT TO TRACK

Conversion and domain errors remain visible; records processed separately.

VALIDATE THE RESULT

```
Accepted: 3
Rejected: invalid literal
for int() with base 10:
'bad'
Rejected: quantity must not
be negative
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Write a function that converts supplied text to an integer and rejects a negative stock quantity. Process zero, minus two and the text unknown. Print a useful rejection message for ValueError. Do not use a bare except and do not replace invalid values with zero.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

Raise explains why an operation cannot honour its contract. Catch specific anticipated errors where you can make a useful decision. Keep the cause visible, and avoid silently swallowing failures. Next, we use development checks and boundary cases to verify reusable calculations.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 6

Check reusable KPI calculations

assert, edge cases and a small KPI library

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

Think of validation as the gate that checks incoming values, and a development assertion as a check on the result of your own reasoning. assert condition raises AssertionError if the condition is false during ordinary execution. Python can remove assertions under optimisation. Therefore, required input validation must use ordinary executable checks and explicit errors.

A PASSING DEVELOPMENT CHECK

CODE WALKTHROUGH W03-L06-S03

```
01  def profit(revenue, cost):
02      return revenue - cost
03
04  assert profit(1200, 800) == 400
05  assert profit(0, 0) == 0
06  print("Two checks passed")
```

WHAT TO TRACK

Normal case and zero boundary; assertions do not print on success.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

A PASSING DEVELOPMENT CHECK — RESULT

CODE WALKTHROUGH W03-L06-S04

```
01  def profit(revenue, cost):
02      return revenue - cost
03
04  assert profit(1200, 800) == 400
05  assert profit(0, 0) == 0
06  print("Two checks passed")
```

WHAT TO TRACK

Normal case and zero boundary; assertions do not print on success.

VALIDATE THE RESULT

```
Two checks passed
```

A KPI WITH AN EMPTY-INPUT POLICY

CODE WALKTHROUGH W03-L06-S05

```
01  def average_sale(amounts):
02      if not amounts:
03          raise ValueError("at least one sale is required")
04      for amount in amounts:
05          if amount < 0:
06              raise ValueError("sales must be non-negative")
07      return sum(amounts) / len(amounts)
08
09  print(average_sale([100, 200]))
10  print(average_sale([]))
```

WHAT TO TRACK

Policy: non-negative numeric sales; empty batch has no average.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

A KPI WITH AN EMPTY-INPUT POLICY — RESULT

CODE WALKTHROUGH W03-L06-S06

```
01  def average_sale(amounts):
02      if not amounts:
03          raise ValueError("at least one sale is required")
04      for amount in amounts:
05          if amount < 0:
06              raise ValueError("sales must be non-negative")
07      return sum(amounts) / len(amounts)
08
09  print(average_sale([100, 200]))
10  print(average_sale([]))
```

WHAT TO TRACK

Policy: non-negative numeric sales; empty batch has no average.

VALIDATE THE RESULT

```
150.0
ValueError: at least one
sale is required
```

A DELIBERATELY WRONG EXPECTATION

CODE WALKTHROUGH W03-L06-S07

```
01  def profit(revenue, cost):
02      return revenue - cost
03
04  assert profit(100, 40) == 70, "check the expected profit"
```

WHAT TO TRACK

Prediction: distinguish a failing test from proof the function is wrong.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

A DELIBERATELY WRONG EXPECTATION — RESULT

CODE WALKTHROUGH W03-L06-S08

```
01  def profit(revenue, cost):
02      return revenue - cost
03
04  assert profit(100, 40) == 70, "check the expected profit"
```

WHAT TO TRACK

Prediction: distinguish a failing test from proof the function is wrong.

VALIDATE THE RESULT

```
AssertionError: check the expected profit
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Create a total-positive-sales function. Return zero for an empty list and include only positive numeric amounts. Check a mixed batch, an empty batch and an all-zero batch with independently calculated expectations. Keep this code together as a small KPI collection; module imports come next.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

Check normal values, boundaries, empty input and invalid input. Assertions support development and may be disabled; required validation uses explicit checks. A failed test requires investigation of both logic and expectation. Next, we organise reusable functions into modules.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 7

Organise reusable code into modules

Imports and useful standard-library tools

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

Think of a module as a labelled toolbox. import math makes the toolbox available under the name math. math.ceil selects one tool from it. Importing does execute a module’s top-level statements when it is first loaded in a process, so keep demonstration prints out of a reusable module. Use module-qualified names to make the origin of a function clear.

USE A STANDARD-LIBRARY MODULE

CODE WALKTHROUGH W03-L07-S03

```
01  import math
02
03  print(math.ceil(4.2))
04  print(math.floor(4.2))
```

WHAT TO TRACK

Ceil rounds toward positive infinity; floor toward negative infinity.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

USE A STANDARD-LIBRARY MODULE — RESULT

CODE WALKTHROUGH W03-L07-S04

```
01  import math
02
03  print(math.ceil(4.2))
04  print(math.floor(4.2))
```

WHAT TO TRACK

Ceil rounds toward positive infinity; floor toward negative infinity.

VALIDATE THE RESULT

```
5
4
```

KEEP A LOCAL KPI MODULE SMALL

CODE WALKTHROUGH W03-L07-S05

```
01  import kpi_tools
02
03  print(kpi_tools.profit(1200, 800))
04  print(kpi_tools.profit(0, 0))
```

WHAT TO TRACK

Supplied dependency kpi_tools.py defines profit(revenue, cost).

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

KEEP A LOCAL KPI MODULE SMALL — RESULT

CODE WALKTHROUGH W03-L07-S06

```
01  import kpi_tools
02
03  print(kpi_tools.profit(1200, 800))
04  print(kpi_tools.profit(0, 0))
```

WHAT TO TRACK

Supplied dependency kpi_tools.py defines profit(revenue, cost).

VALIDATE THE RESULT

```
400
0
```

DIAGNOSE AN UNAVAILABLE MODULE

CODE WALKTHROUGH W03-L07-S07

```
01  import aisynapse_missing_demo_module
```

WHAT TO TRACK

Deliberate missing dependency; this name is fictional.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

DIAGNOSE AN UNAVAILABLE MODULE — RESULT

CODE WALKTHROUGH W03-L07-S08

```
01  import aisynapse_missing_demo_module
```

WHAT TO TRACK

Deliberate missing dependency; this name is fictional.

VALIDATE THE RESULT

```
ModuleNotFoundError: No module named 'aisynapse_missing_demo_module'
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Open the supplied kpi_tools.py file. Add a revenue(quantity, price) function returning the product. In a separate runner, import kpi_tools and print revenue for three items priced at twenty. Keep demonstration prints in the runner.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

Modules organise reusable definitions. Imports make their names available, and module-qualified calls show where a tool comes from. Keep local dependency files with the runner and avoid standard-library name collisions. Next, we use pathlib to work with portable file paths.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 8

Read and write portable text files

pathlib, UTF-8 and context managers

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

A Path object represents a location; creating it does not create the file. The slash operator joins path components portably. Opening a text file translates bytes using an encoding. with introduces a context manager: when the block finishes, including through an exception, the file is closed. Closing is part of resource handling, not the same action as saving a Python variable.

WRITE AND READ A SHORT REPORT

CODE WALKTHROUGH W03-L08-S03

```
01  from pathlib import Path
02
03  path = Path("report.txt")
04  with path.open("w", encoding="utf-8") as handle:
05      handle.write("Sales: 300\n")
06  with path.open("r", encoding="utf-8") as handle:
07      text = handle.read()
08  print(text, end="")
```

WHAT TO TRACK

Write mode replaces existing content; use the isolated exercise folder.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

WRITE AND READ A SHORT REPORT — RESULT

CODE WALKTHROUGH W03-L08-S04

```
01  from pathlib import Path
02
03  path = Path("report.txt")
04  with path.open("w", encoding="utf-8") as handle:
05      handle.write("Sales: 300\n")
06  with path.open("r", encoding="utf-8") as handle:
07      text = handle.read()
08  print(text, end="")
```

WHAT TO TRACK

Write mode replaces existing content; use the isolated exercise folder.

VALIDATE THE RESULT

```
Sales: 300
```

READ A SUPPLIED BUSINESS NOTE

CODE WALKTHROUGH W03-L08-S05

```
01  from pathlib import Path
02
03  path = Path("data") / "business_note.txt"
04  with path.open("r", encoding="utf-8") as handle:
05      lines = handle.read().splitlines()
06  print(lines[0])
07  print("Lines:", len(lines))
```

WHAT TO TRACK

Supplied UTF-8 note: café inventory; two lines.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

READ A SUPPLIED BUSINESS NOTE — RESULT

CODE WALKTHROUGH W03-L08-S06

```
01  from pathlib import Path
02
03  path = Path("data") / "business_note.txt"
04  with path.open("r", encoding="utf-8") as handle:
05      lines = handle.read().splitlines()
06  print(lines[0])
07  print("Lines:", len(lines))
```

WHAT TO TRACK

Supplied UTF-8 note: café inventory; two lines.

VALIDATE THE RESULT

```
Café inventory
Lines: 2
```

A MISSING PATH IS A REAL FAILURE

CODE WALKTHROUGH W03-L08-S07

```
01  from pathlib import Path
02
03  path = Path("data") / "missing_note.txt"
04  with path.open("r", encoding="utf-8") as handle:
05      print(handle.read())
```

WHAT TO TRACK

Deliberate missing file; read mode does not create it.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

A MISSING PATH IS A REAL FAILURE — RESULT

CODE WALKTHROUGH W03-L08-S08

```
01 from pathlib import Path
02
03 path = Path("data") / "missing_note.txt"
04 with path.open("r", encoding="utf-8") as handle:
05     print(handle.read())
```

WHAT TO TRACK

Deliberate missing file; read mode does not create it.

VALIDATE THE RESULT

```
FileNotFoundError: [Errno 2]
No such file or directory:
'data/missing_note.txt'
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Write a UTF-8 summary file in the exercise folder containing two lines: Accepted: 2 and Rejected: 1. Read it back with a context manager, print its text and check that splitlines contains two entries.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

A path describes a location. Explicit UTF-8 controls text encoding, and with closes opened files reliably. Relative paths support sharing when the working folder is documented. Next, we read structured records with CSV and JSON instead of manually splitting text.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 9

Read structured records and check their schema

CSV, JSON and validation

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

Parsing answers whether the text follows a format. Schema validation answers whether the parsed structure has the fields and types our task requires. CSV readers usually give strings for field values. JSON preserves several kinds of value, including numbers, strings, lists, objects and null, which becomes None in Python. A successful parse alone does not make a record usable.

CSV HANDLES A QUOTED COMMA

CODE WALKTHROUGH W03-L09-S03

```
01  import csv
02  from pathlib import Path
03
04  with Path("data/invoices.csv").open("r", encoding="utf-8", newl
↪   ine="") as handle:
05      rows = list(csv.DictReader(handle))
06  print(rows[0]["customer"])
07  print(type(rows[0]["amount"]).__name__)
```

WHAT TO TRACK

Required columns: invoice_id, customer, amount. CSV amount begins as text.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

CSV HANDLES A QUOTED COMMA — RESULT

CODE WALKTHROUGH W03-L09-S04

```
01  import csv
02  from pathlib import Path
03
04  with Path("data/invoices.csv").open("r", encoding="utf-8", newl
↪   ine="") as handle:
05      rows = list(csv.DictReader(handle))
06  print(rows[0]["customer"])
07  print(type(rows[0]["amount"]).__name__)
```

WHAT TO TRACK

Required columns: invoice_id, customer, amount. CSV amount begins as text.

VALIDATE THE RESULT

```
Acme, North
str
```

CONVERT AND VALIDATE A CSV IMPORT

CODE WALKTHROUGH W03-L09-S05

```
01  import csv
02  from pathlib import Path
03
04  with Path("data/invoices.csv").open("r", encoding="utf-8", newl
↪   ine="") as handle:
05      reader = csv.DictReader(handle)
06      required = {"invoice_id", "customer", "amount"}
07      if not required.issubset(set(reader.fieldnames or [])):
08          raise ValueError("missing invoice columns")
09      total = 0
10      for row in reader:
11          amount = int(row["amount"])
12          if amount <= 0:
13              raise ValueError("invoice amount must be positive")
14          total += amount
15  print("Total:", total)
```

WHAT TO TRACK

Two invoices: 120 and 250; positive integer amounts required.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

CONVERT AND VALIDATE A CSV IMPORT — RESULT

CODE WALKTHROUGH W03-L09-S06

```
01  import csv
02  from pathlib import Path
03
04  with Path("data/invoices.csv").open("r", encoding="utf-8", newl
↪   ine="") as handle:
05      reader = csv.DictReader(handle)
06      required = {"invoice_id", "customer", "amount"}
07      if not required.issubset(set(reader.fieldnames or [])):
08          raise ValueError("missing invoice columns")
09      total = 0
10      for row in reader:
11          amount = int(row["amount"])
12          if amount <= 0:
13              raise ValueError("invoice amount must be positive")
14          total += amount
15  print("Total:", total)
```

WHAT TO TRACK

Two invoices: 120 and 250; positive integer amounts required.

VALIDATE THE RESULT

```
Total: 370
```

VALID JSON, UNSUITABLE SCHEMA

CODE WALKTHROUGH W03-L09-S07

```
01  import json
02  from pathlib import Path
03
04  with Path("data/bad_schema.json").open("r", encoding="utf-8") a
↪   s handle:
05      record = json.load(handle)
06  if not isinstance(record, dict):
07      raise ValueError("invoice must be an object")
08  if "amount" not in record:
09      raise ValueError("missing amount")
10  print(record["amount"])
```

WHAT TO TRACK

JSON text is valid; its object lacks the required amount key.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

VALID JSON, UNSUITABLE SCHEMA — RESULT

CODE WALKTHROUGH W03-L09-S08

```
01  import json
02  from pathlib import Path
03
04  with Path("data/bad_schema.json").open("r", encoding="utf-8") a
↪   s handle:
05      record = json.load(handle)
06  if not isinstance(record, dict):
07      raise ValueError("invoice must be an object")
08  if "amount" not in record:
09      raise ValueError("missing amount")
10  print(record["amount"])
```

WHAT TO TRACK

JSON text is valid; its object lacks the required amount key.

VALIDATE THE RESULT

```
ValueError: missing amount
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Read the supplied invoices.json file. Require a list of objects, each with an invoice_id string and a positive integer amount. Return the sum. Test the normal file, an empty list and an object missing amount. Decide and document that an empty list has total zero.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

CSV handles quoted fields and normally supplies text. JSON supplies several structured types, but parsing does not prove a business schema. Check required keys, types and values before calculating. Next, the same distinction helps us understand API responses.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 10

Treat API responses as untrusted inputs

HTTP, query parameters, timeouts and errors

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

Think of sending a labelled request and receiving an envelope. The status describes the delivery outcome; the body contains the actual data. A query parameter narrows the request. A timeout limits waiting for a network operation, though it is not a promise that every whole workflow finishes by that instant. Required exercises here use local fixtures and make no live requests.

ENCODE QUERY PARAMETERS SAFELY

CODE WALKTHROUGH W03-L10-S03

```
01  from urllib.parse import urlencode
02
03  params = {"region": "North East", "limit": 2}
04  query = urlencode(params)
05  print(query)
```

WHAT TO TRACK

Encode parameter values rather than manually joining arbitrary text.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

ENCODE QUERY PARAMETERS SAFELY — RESULT

CODE WALKTHROUGH W03-L10-S04

```
01  from urllib.parse import urlencode
02
03  params = {"region": "North East", "limit": 2}
04  query = urlencode(params)
05  print(query)
```

WHAT TO TRACK

Encode parameter values rather than manually joining arbitrary text.

VALIDATE THE RESULT

```
region=North+East&limit=2
```

CHECK STATUS, PARSE AND VALIDATE

CODE WALKTHROUGH W03-L10-S05

```
01 import json
02 from pathlib import Path
03
04 with Path("data/mock_success.json").open("r", encoding="utf-8")
↪   as handle:
05     response = json.load(handle)
06 if response["status"] != 200:
07     raise ValueError("request did not succeed")
08 payload = json.loads(response["body"])
09 amount = payload.get("amount")
10 if type(amount) is not int or amount < 0:
11     raise ValueError("amount must be a non-negative integer")
12 print("Amount:", amount)
```

WHAT TO TRACK

Mock envelope: status 200 and a JSON body containing amount 370.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

CHECK STATUS, PARSE AND VALIDATE — RESULT

CODE WALKTHROUGH W03-L10-S06

```
import json
from pathlib import Path

with Path("data/mock_success.json").open("r", encoding="utf-8")
 as handle:
    response = json.load(handle)
if response["status"] != 200:
    raise ValueError("request did not succeed")
payload = json.loads(response["body"])
amount = payload.get("amount")
if type(amount) is not int or amount < 0:
    raise ValueError("amount must be a non-negative integer")
print("Amount:", amount)
```

WHAT TO TRACK

Mock envelope: status 200 and a JSON body containing amount 370.

VALIDATE THE RESULT

```
Amount: 370
```

HTTP SUCCESS DOES NOT PROVE THE SCHEMA

CODE WALKTHROUGH W03-L10-S07

```
01 import json
02 from pathlib import Path
03
04 with Path("data/mock_bad_schema.json").open("r", encoding="utf-
↪ 8") as handle:
05     response = json.load(handle)
06 if response["status"] != 200:
07     raise ValueError("request did not succeed")
08 payload = json.loads(response["body"])
09 if not isinstance(payload, dict) or "amount" not in payload:
10     raise ValueError("response is missing amount")
11 print(payload["amount"])
```

WHAT TO TRACK

Mock status is 200, body parses, required key is absent.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

HTTP SUCCESS DOES NOT PROVE THE SCHEMA — RESULT

CODE WALKTHROUGH W03-L10-S08

```
01 import json
02 from pathlib import Path
03
04 with Path("data/mock_bad_schema.json").open("r", encoding="utf-
↪ 8") as handle:
05     response = json.load(handle)
06 if response["status"] != 200:
07     raise ValueError("request did not succeed")
08 payload = json.loads(response["body"])
09 if not isinstance(payload, dict) or "amount" not in payload:
10     raise ValueError("response is missing amount")
11 print(payload["amount"])
```

WHAT TO TRACK

Mock status is 200, body parses, required key is absent.

VALIDATE THE RESULT

```
ValueError: response is
missing amount
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Write an offline response validator. Require status 200, a JSON object body and a non-negative integer amount. Test mock_success, mock_bad_schema and mock_http_error. Report the failure reason without a live request or credentials.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

Separate request status, body parsing and schema validation. Encode query parameters, bound live network waits and avoid endless retries. All required exercises remain offline. Next, we discuss secrets and the object vocabulary used by paths and response values.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 11

Protect secrets and understand objects

Environment configuration, attributes and methods

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

An object combines a value with operations and attributes associated with its type. An attribute is a named value accessed after a dot. A method is an operation accessed through an object and called with parentheses. An environment variable supplies configuration to a process. Reading it avoids embedding the value in source code, but environment variables are not automatically encrypted or safe to print.

REPORT CONFIGURATION WITHOUT REVEALING IT

CODE WALKTHROUGH W03-L11-S03

```
01  import os
02
03  key = os.environ.get("AISYNAPSE_DEMO_API_KEY")
04  if key:
05      print("Key configured")
06  else:
07      print("Offline mode: no key needed")
```

WHAT TO TRACK

Exercise runner supplies no key;
never print the secret value.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

REPORT CONFIGURATION WITHOUT REVEALING IT — RESULT

CODE WALKTHROUGH W03-L11-S04

```
01  import os
02
03  key = os.environ.get("AISYNAPSE_DEMO_API_KEY")
04  if key:
05      print("Key configured")
06  else:
07      print("Offline mode: no key needed")
```

WHAT TO TRACK

Exercise runner supplies no key; never print the secret value.

VALIDATE THE RESULT

```
Offline mode: no key needed
```

ATTRIBUTES AND METHODS ON A PATH

CODE WALKTHROUGH W03-L11-S05

```
01  from pathlib import Path
02
03  p = Path("data") / "invoices.csv"
04  print(p.name)
05  print(p.suffix)
06  print(p.exists())
```

WHAT TO TRACK

name and suffix are attributes;
exists() is a method call.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

ATTRIBUTES AND METHODS ON A PATH — RESULT

CODE WALKTHROUGH W03-L11-S06

```
01  from pathlib import Path
02
03  p = Path("data") / "invoices.csv"
04  print(p.name)
05  print(p.suffix)
06  print(p.exists())
```

WHAT TO TRACK

name and suffix are attributes;
exists() is a method call.

VALIDATE THE RESULT

```
invoices.csv
.csv
True
```

DO NOT CALL A DATA ATTRIBUTE

CODE WALKTHROUGH W03-L11-S07

```
01  from pathlib import Path
02
03  p = Path("data/invoices.csv")
04  print(p.name())
```

WHAT TO TRACK

Deliberate error: name contains a string, not a method.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

DO NOT CALL A DATA ATTRIBUTE — RESULT

CODE WALKTHROUGH W03-L11-S08

```
01  from pathlib import Path
02
03  p = Path("data/invoices.csv")
04  print(p.name())
```

WHAT TO TRACK

Deliberate error: name contains a string, not a method.

VALIDATE THE RESULT

```
TypeError: 'str' object is not callable
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Create a Path for the supplied invoices.json file. Display its name and suffix, then call exists to check it. Separately check whether the demo key is configured, printing only a status. Do not supply or reveal any key.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

Keep secrets outside shared source and never log their values. Attributes hold named values; methods are called operations. Existing Path and string objects already illustrate these ideas. Next, we define one small class and combine our offline ingestion skills.

UNDERSTAND APPLY VALIDATE EXPLAIN

FOUNDATION LESSON

WEEK 3 · LESSON 12

Build one small ingestion summary object

A small class and an offline file/API exercise

UNDERSTAND THE IDEA FIRST

MENTAL MODEL

Understand the idea first

A class describes an object type; an instance is one object created from that class. Calling BatchSummary creates an instance, and __init__ initialises its state. self is the conventional first parameter referring to that instance. self dot amounts is an instance attribute. A method receives the instance automatically when called through it; you do not pass self yourself in that call.

ONE CLASS, ONE ATTRIBUTE, ONE METHOD

CODE WALKTHROUGH W03-L12-S03

```
01  class BatchSummary:
02      def __init__(self, amounts):
03          self.amounts = amounts.copy()
04
05      def total(self):
06          return sum(self.amounts)
07
08  batch = BatchSummary([120, 250])
09  print(batch.amounts)
10  print(batch.total())
```

WHAT TO TRACK

Input contract here: validated flat list of positive integers.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

ONE CLASS, ONE ATTRIBUTE, ONE METHOD — RESULT

CODE WALKTHROUGH W03-L12-S04

```
01  class BatchSummary:
02      def __init__(self, amounts):
03          self.amounts = amounts.copy()
04
05      def total(self):
06          return sum(self.amounts)
07
08  batch = BatchSummary([120, 250])
09  print(batch.amounts)
10  print(batch.total())
```

WHAT TO TRACK

Input contract here: validated flat list of positive integers.

VALIDATE THE RESULT

```
[120, 250]
370
```

VALIDATE FILE RECORDS, THEN SUMMARISE

CODE WALKTHROUGH W03-L12-S05

```
01  import json
02  from pathlib import Path
03  from ingestion_tools import validated_amounts
04
05  class BatchSummary:
06      def __init__(self, amounts):
07          self.amounts = amounts.copy()
08
09      def total(self):
10          return sum(self.amounts)
11
12  with Path("data/invoices.json").open("r", encoding="utf-8") as
↪   handle:
13      records = json.load(handle)
14  amounts = validated_amounts(records)
15  batch = BatchSummary(amounts)
16  print("Accepted:", len(batch.amounts))
17  print("Total:", batch.total())
```

WHAT TO TRACK

Supplied dependency: validated_amounts in ingestion_tools.py.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

VALIDATE FILE RECORDS, THEN SUMMARISE — RESULT

CODE WALKTHROUGH W03-L12-S06

```
import json
from pathlib import Path
from ingestion_tools import validated_amounts

class BatchSummary:
    def __init__(self, amounts):
        self.amounts = amounts.copy()

    def total(self):
        return sum(self.amounts)

with Path("data/invoices.json").open("r", encoding="utf-8") as handle:
    records = json.load(handle)
amounts = validated_amounts(records)
batch = BatchSummary(amounts)
print("Accepted:", len(batch.amounts))
print("Total:", batch.total())
```

WHAT TO TRACK

Supplied dependency: validated_amounts in ingestion_tools.py.

VALIDATE THE RESULT

```
Accepted: 2
Total: 370
```

A MOCK API CANNOT BYPASS VALIDATION

CODE WALKTHROUGH W03-L12-S07

```
01  import json
02  from pathlib import Path
03  from ingestion_tools import validated_amounts
04
05  with Path("data/mock_invoice_bad.json").open("r", encoding="utf
↪   -8") as handle:
06      response = json.load(handle)
07  if response["status"] != 200:
08      raise ValueError("unsuccessful status")
09  records = json.loads(response["body"])
10  print(validated_amounts(records))
```

WHAT TO TRACK

Successful mock response contains an amount with the wrong type.

PREDICT THE RESULT

Predict the output.
Explain each operation.
Pause before continuing.

A MOCK API CANNOT BYPASS VALIDATION — RESULT

CODE WALKTHROUGH W03-L12-S08

```
01 import json
02 from pathlib import Path
03 from ingestion_tools import validated_amounts
04
05 with Path("data/mock_invoice_bad.json").open("r", encoding="utf
↪ -8") as handle:
06     response = json.load(handle)
07 if response["status"] != 200:
08     raise ValueError("unsuccessful status")
09 records = json.loads(response["body"])
10 print(validated_amounts(records))
```

WHAT TO TRACK

Successful mock response contains an amount with the wrong type.

VALIDATE THE RESULT

```
ValueError: amount must be a
positive integer
```

PAUSE AND APPLY

LEARNER PRACTICE

Pause and apply

Use the supplied successful invoice mock. Check status 200, parse the body, validate all records and construct BatchSummary. Print its total and compare it with the file total. Test the empty list policy: an empty valid batch totals zero. Report malformed records rather than silently dropping them.

PREDICT · MODIFY · RUN

01 Restate the supplied rules.

02 Predict the result.

03 Modify and run the code.

04 Compare actual output.

05 Explain any difference.

RECAP AND NEXT STEP

Recap and next step

You can now define reusable functions, validate inputs, handle known errors and organise code into modules. You can read text, CSV and JSON, check mock API responses and protect secrets. A small class can keep related validated state and one operation together. Complete the weekly quiz and project; formative practice remains separate. Next week introduces NumPy and pandas.

UNDERSTAND APPLY VALIDATE EXPLAIN