Lecture 13

Error Handling and Debugging

Understanding program failures

Error Handling and Debugging · Lecture 13

Three kinds of failure

- **Syntax errors** prevent Python from running the program
- **Runtime errors** occur while executing an operation
- **Logic errors** produce a result that violates the intended requirement
- Each category requires a different approach to diagnosis.

```
SYNTAX    missing : after if
RUNTIME   float("unknown")
LOGIC     wrong denominator
```

A program that runs without crashing is not necessarily correct.

Three different ways a program can fail

Choose the debugging action based on what actually happens.

SYNTAX ERROR	RUNTIME EXCEPTION	LOGIC ERROR
Cannot start	**Stops during execution**	**Runs but gives wrong result**
Fix grammar / indentation	Read traceback, handle expected case	Compare with a known test

Not all bugs raise exceptions: a plausible but wrong output is still a failure.

Syntax errors: read the location

- Python cannot parse a missing colon, parenthesis or invalid indentation
- The error often points close to the mistake; inspect nearby lines too
- Fix syntax errors before reasoning about the program result
- Do not mask syntax errors using `try/except`: code must parse first (we will see).

```
if amount > 0
    print(amount)

# SyntaxError: expected ":"
```

Always read the exception type and its explanation.

Exceptions during execution

- A **runtime exception** signals that an operation could not complete
- Unhandled exceptions stop normal execution and show a traceback
- The **exception has a type** and often a message describing the failure
- Some failures are expected user errors; some reveal programmer defects.

```
float("N/A")    # ValueError
values[99]      # IndexError
10 / 0          # ZeroDivisionError
```

Catching every exception indiscriminately hides useful information.

Anatomy of a traceback

- Start with the last line: exception type and message
- Look above it for the **file name** and **line** that failed
- If several functions are involved, traceback frames show the **call chain**
- Use nearby values and the task requirements to explain the cause.

```
Traceback (most recent call last):
  File "demo.py", line 4, in main
    amount = float(text)
ValueError: could not convert string to
  float: "NA"
```

Ask what input reached the failing line, not merely where it crashed.

Reading a traceback: where did the failure originate?

Read from the exception type at the bottom to the source location above it.

```
Traceback (most recent call last):
  File "main.py", line 12, in main
    result = convert(raw)
  File "helpers.py", line 4, in convert
    return float(raw)
ValueError: could not convert string to float: "abc"
```

CALL CHAIN
main → convert

SOURCE LINE
helpers.py, line 4

ERROR TYPE
ValueError

Tracebacks explain where the program failed; they do not always identify why your input was wrong.

Common exceptions at a glance

EXPLANATION

- `ValueError`: correct type but unsuitable value (e.g. `float("bad")`)
- `TypeError`: unsupported operation or wrong argument type
- `IndexError / KeyError`: missing sequence index or dictionary key
- `ZeroDivisionError`: zero denominator
- `FileNotFoundError`: requested file does not exist.

EXAMPLE / STRUCTURE

```
float("bad")   → ValueError
[1, 6][2]      → IndexError
{"x": 1}["y"] → KeyError
1 / 0          → ZeroDivisionError
```

Use the error type to narrow your investigation.

Validation versus handling

- Validation checks conditions that can be tested predictably *(I know the domain of the data, and I may expect this to be violated by the input, but I can also check it)*
- Exception handling manages an operation that may fail *(I can forcast some types of exception that may occur, but I cannot prevent them)*
- Both techniques are complementary: check domain values, handle conversion and I/O failures
- Do not use try/except to replace ordinary if/else logic.

```
if months <= 0:
    print("Months must be positive")
```

```
try:
    amount = float(text)
except ValueError:
    print("Numeric input required")
```

Choose a method matching the nature of the failure.

Managing expected exceptions

Error Handling and Debugging · Lecture 13

try/except/else/finally: basic syntax

The basic structure to catch exceptions is:

`try:`

Required. Indented statements that Python executes first; they may raise an exception.

`except ErrorType:`

Required. Handles a matching exception raised in `try`. If one occurs, the remaining `try` statements are skipped. You may pass multiple Error Types separated by commas (`except ET1, ET2,…:`). *Multiple except can be used to distingush different error types*

`else:`

Optional. Executes only when try finishes **without** an exception.

`finally:`

Optional. Executes when leaving the statement, whether an exception occurred or not; often used for cleanup (check if temporary files or structures must be deleted, close files, logs…).

```
try:
    risky_operation()
except ValueError:
    handle_invalid_value()
else:
    process_success()
finally:
    perform_cleanup()
```

try/except: basic control flow

- Python enters the `try` suite first
- When no exception occurs, the `except` suite(s) is skipped
- If a matching exception occurs, normal `try` execution stops and the handler runs
- Code after the try/except continues unless a new exception is raised.

```
try:
    number = int(input("Enter a number: "))
    result = 100 / number

except ValueError:
    print("Invalid input.")

except ZeroDivisionError:
    print("Cannot divide by zero.")

else:
    print("Result:", result)

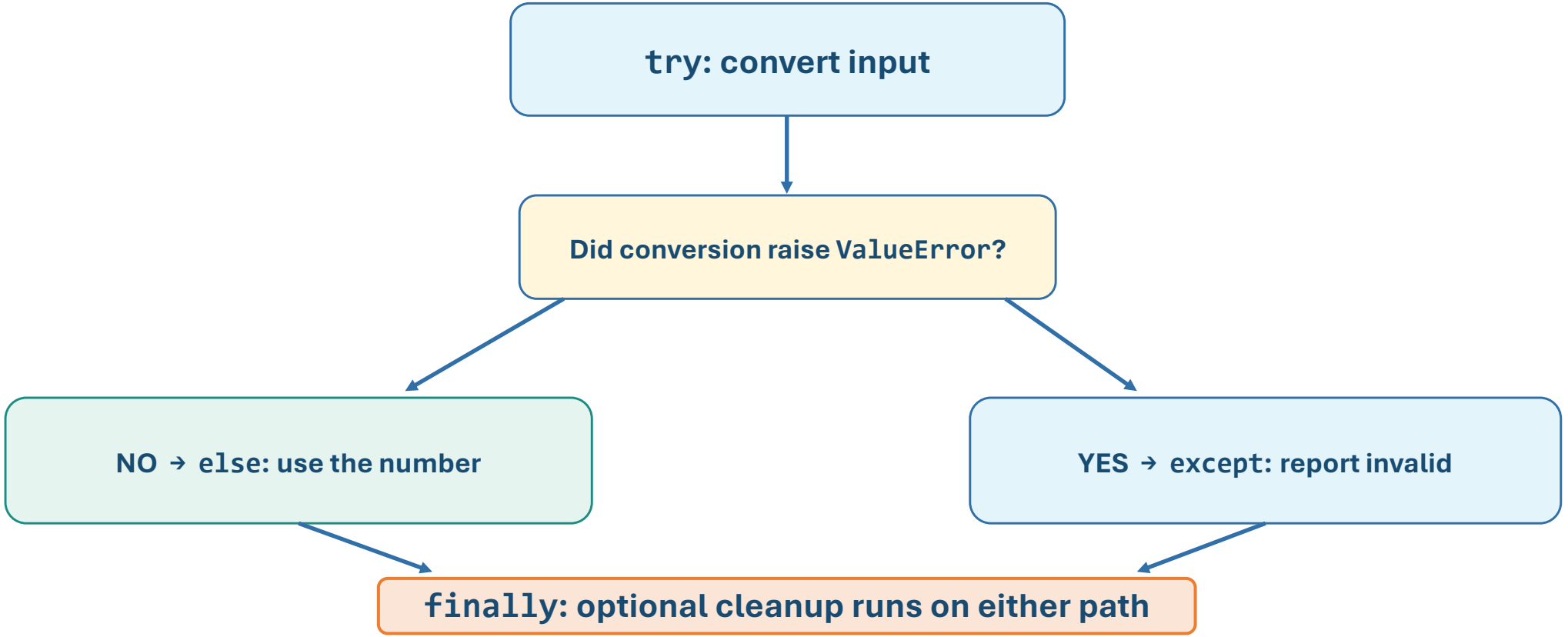
finally:
    print("End of try statement.")

# other operations may follow
print("The program continues!")
```

Keep the `try` suite small: only operations expected to fail.

Exception handling changes the control flow

An except branch runs only when a matching exception is raised.



Catch only expected errors, and keep the protected try block as narrow as practical.

Why catch a specific exception?

- `except ValueError` handles only numeric conversion failures
- An `IndexError` elsewhere should remain visible for debugging
- A bare `except` catches far too broadly, including unexpected failures
- Use a tuple of types only when the same policy applies to each.

```
try:
    amount = float(row["amount"])
except (KeyError, ValueError):
    print("Invalid amount field")
```

else and finally: distinct purposes

- `else` runs only if the try suite completed without raising an exception
- `finally` runs when leaving the construct, whether an exception occurred or not
- A `with` statement already manages normal file closing
- Choose `else` for success-only behaviour and `finally` for cleanup.

```
try:
    x = float(text)
except ValueError:
    print("Bad input")
else:
    print("Parsed", x)
finally:
    print("Attempt completed")
```

`else` is not equivalent to code after except when exceptions occur.

try/except: basic example

- Only the first matching except runs
- `else` runs only when try succeeds
- `finally`, always runs in these cases
- Then, if no other exception is raised in this statement, the program continues *assuming the problem is solved*

Input	Except	Else	Finally	After try
5	---	Y	Y	Y*
0	ZeroDivisionError	N	Y	Y*
abc	ValueError	N	Y	Y*

*if no exit() or other rise Error is called in the blocks

```
try:
    number = int(input("Enter a number: "))
    result = 100 / number

except ValueError:
    print("Invalid input.")

except ZeroDivisionError:
    print("Cannot divide by zero.")

else:
    print("Result:", result)

finally:
    print("End of try statement.")

# other operations may follow
print("The program continues!")
```

`raise`: reject a violated contract

- A function may raise a standard exception (or a custom one) to signal invalid arguments, using the keyword `raise`
- Statement structure: `raise ErrorType("optional message")`
- *The caller decides* whether to catch it and how to inform the user
- Raising is better than silently returning a misleading number
- Custom exception classes are not part of this lecture.

```python
def average(values):
    if not values:
        raise ValueError("Empty list")

    return sum(values) / len(values)
```

Document when a function can fail (e.g. in docstrings).

How exceptions propagate between functions

If a function raises an exception and does not handle it, execution stops inside that function. Python looks for a matching handler in the calling function, then in its caller, and so on.

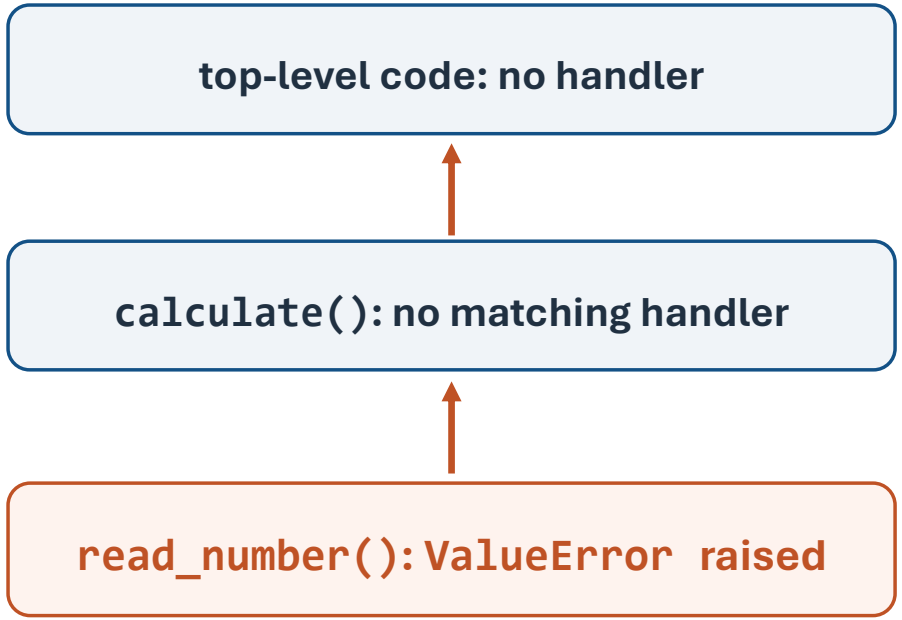
EXAMPLE – NO EXCEPTION HANDLER

```
def read_number(text):
    return int(text)

def calculate():
    value = read_number("abc")
    print("Value:", value)

calculate()
print("Program continues")
```

EXCEPTION PROPAGATION



The exception remains unhandled → traceback; the final print is NOT executed.

Handling a propagated exception in the caller

The caller can catch an exception raised by a function it calls. Python jumps to its matching except block; it does not resume the interrupted instruction.

EXAMPLE – calculate() HANDLES THE ERROR

```
def read_number(text):
    return int(text)

def calculate():
    try:
        value = read_number("abc")
        print("Value:", value)

    except ValueError:
        print("Invalid number")

    print("Function continues")

calculate()
print("Program continues")
```

WHAT HAPPENS?

`read_number()`: raises `ValueError`

↓

`calculate()`: except `ValueError` catches it

OUTPUT

```
Invalid number
Function continues
Program continues
```

The skipped `print("Value:", value)` does not run.

Working with missing files

- Opening in `'r'` mode can fail if the file does not exist
- Catch `FileNotFoundError` close to the operation or at the interface
- Provide a useful path-specific message
- Do not confuse missing data with an empty dataset.

```
try:
    with open(path, encoding="utf-8") as f:
        content = f.read()

except FileNotFoundError:
    print(f"Cannot find: {path}")
```

The file may be missing even if its name looks correct.

Malformed CSV rows: choose a policy

- One incorrect row does not always mean the whole file is unusable
- Specify whether to skip bad rows, stop processing or mark them invalid
- Count rejected records so information is not silently lost
- Avoid catching unrelated exceptions around the whole program.

```
for row in rows:
    try:
        amount = float(row["amount"])

    except (KeyError, ValueError):
        skipped += 1
        continue

    values.append(amount)
```

A parsing policy is part of the task specification.

Running example: processing trip data

A CSV records trips. The program must compute average speed for each row, then build a report containing only valid results

INPUT FILE | trips.csv

```
Trip,Distance_km,Hours
A,300,3
B,280,invalid
C,120,0
D,150,2
E,100,-2
```

REQUIRED CALCULATION

Speed (km/h) = Distance_km / Hours

Hours must be a number greater than zero

Row B cannot be converted to a number

Row C would require division by zero

Question: should one invalid record invalidate the whole report, or can we safely skip it?

Where does the exception come from?

Conversion and domain validation are different checks: Python detects some failures, while our program must enforce the business rule

```
# 1. Invalid conversion
hours = float("invalid")  # ValueError

# 2. Division by zero
hours = float("0")
speed = 120 / hours         # ZeroDivisionError

# 3. Invalid domain value
hours = float("-2")
if hours < 0:
    raise ValueError("Hours cannot be negative")
```

Conversion: Python raises the exception

Validation: we raise the exception

WHAT HAPPENS NEXT?

When an exception is raised, the current normal execution path stops and Python looks for a matching except handler

A try/except decides what to do about the failure, it does not automatically repair the invalid data

First step: read the exception message

The syntax `except <ErrorType> as e` captures the exception object in a variable named e, so we can explain what went wrong

ONE CSV ROW

```
row = {
    "Trip": "B",
    "Distance_km": "280",
    "Hours": "invalid"
}
```

INTERCEPT AND DISPLAY

```
try:
    hours = float(row["Hours"])
except ValueError as e:
    print(f"Trip {row["Trip"]}: {e}")
```

DISPLAYED MESSAGE

```
Trip B: could not convert string to float: 'invalid'
```

`print(e)` shows the error message, **not the full traceback**; and no usable hours value has been recovered

Safe recovery: skip the invalid CSV record

If the report allows incomplete data, the program can **ignore one bad row** and continue processing the remaining records

PROCESSING LOOP

```
import csv

with open("trips.csv", newline="") as f:
    for row in csv.DictReader(f):
        try:
            distance = float(row["Distance_km"])
            hours = float(row["Hours"])

            if hours < 0:
                raise ValueError("Hours cannot be negative")

            speed = distance / hours

        except ValueError as e:
            print(f"Skipping {row['Trip']}: {e}")
            continue

        except ZeroDivisionError as e:
            print(f"Skipping {row['Trip']}: {e}")
            continue

        print(f"{row['Trip']}: {speed:.2f} km/h")
```

WHY THIS IS SAFE

`continue` skips the remainder of the current for iteration

No division takes place with missing or non-positive hours

The next CSV row is still processed

This policy is appropriate only if skipping bad records is acceptable for the purpose of the report

Result of the skip policy

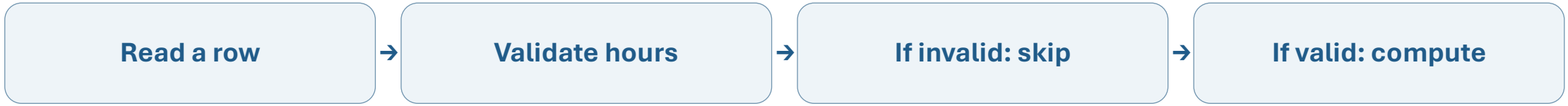
The report contains only successful calculations; invalid rows are explicitly reported instead of silently producing incorrect results

CONSOLE WARNINGS

```
Skipping B: could not convert string to
float: 'invalid'
Skipping C: float division by zero
Skipping E: Hours cannot be negative
```

VALID REPORT RECORDS

```
A: 100.00 km/h
D: 75.00 km/h
```



The same try/except runs for each CSV row. One failure does not discard the already processed rows

Another policy: ask the user to skip or quit

For an interactive program, the decision can be made by the user when an invalid CSV record is found

DECISION AFTER A FAILED VALIDATION

```
except (ValueError, ZeroDivisionError) as e:
    print(f"Invalid row {row['Trip']}: {e}")
    choice = input("[s]kip or [q]uit? ").strip().lower()

    if choice == "s":
        continue

    elif choice == "q":
        return

    else:
        print("Unknown option: stopping")
        return
```

USER INTERACTION

```
Invalid row B: ...
[s]kip or [q]uit? s

# Continue with row C
```

Here `return` stops the function that contains the loop; it does not automatically terminate every caller

A complete solution could repeat the question for invalid choices; this simplified version stops safely

Stopping deliberately: return vs `sys.exit()`

Stopping the current function is different from asking Python to terminate the entire program

`return` | stop the current function

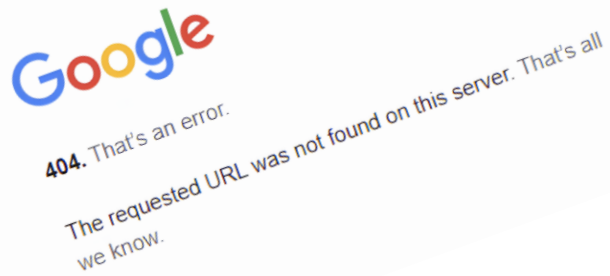
```
def process_trips():
    ...
    if choice == "q":
        return
        # Stops this function only
```

`sys.exit(1)` | end the program

```
import sys

if choice == "q":
    sys.exit(1)
    # Raises SystemExit internally
```





WHAT IS AN EXIT STATUS?

- A program returns a small status code to the operating system or shell when it finishes
- 0 conventionally means success; a non-zero value (such as 1) signals failure
- Different non-zero exit values may represent a different kind of errors

Let the caller decide: raise and propagation

A reusable function can detect a bad CSV field without deciding whether the caller should skip the row or stop the report

HELPER FUNCTION: VALIDATE OR RAISE

```python
def read_hours(row):
    hours = float(row["Hours"])

    if hours < 0:
        raise ValueError("Hours cannot be negative")

    return hours
```

CALLER: INSIDE THE CSV LOOP

```python
try:
    hours = read_hours(row)
    speed = float(row["Distance_km"]) / hours

except (ValueError, ZeroDivisionError) as e:
    print(f"Skipping: {e}")
    continue
```

read_hours detects failure → **Exception goes to caller** → **Caller catches ValueError** → **Caller skips the row**

Without a matching handler, the exception keeps propagating to the next caller

Re-raising: keep the original exception

Inside an except block, `raise` without arguments **re-raises the exception being handled**, rather than creating an unrelated error

HELPER ADDS CONTEXT, THEN RE-RAISES

```
def read_hours(row):
    try:
        return float(row["Hours"])
    except ValueError as e:
        print(f"Bad hours in trip {row['Trip']}: {e}")
        raise
```

THE CALLER CAN STILL DECIDE

```
try:
    hours = read_hours(row)
except ValueError as e:
    print(f"Skipping: {e}")
    continue
```

Re-raising preserves the original exception and its traceback information. It does not force the caller to display that traceback

Exception message vs traceback: what is shown?

An exception object retains diagnostic details, but the output depends on what the handler explicitly asks Python to display

SHORT MESSAGE

```
except ValueError as e:
    print(e)
```

Shows only the exception message

Example: *Hours must be > 0*

FULL TRACEBACK

```
import traceback

try:
    hours = float("invalid")

except ValueError:
    traceback.print_exc()
```

Shows the **error type**, **message** and **call locations**, including file names and line numbers

`traceback` is a Standard Library module; `print_exc()` prints the traceback of the exception currently being handled

What happens after the caller catches a re-raised error?

The traceback is preserved by `raise`, but `print(e)` alone still shows only the message, not the chain of calls

SAME EXCEPTION, TWO DISPLAY CHOICES

```python
try:
    read_hours(row)  # may re-raise

except ValueError as e:
    print(f"Invalid row: {e}")
    # No traceback printed
    # OR, for diagnosis:
    traceback.print_exc()
```

WHAT EACH CHOICE SHOWS

`print(e)`

Only the explanatory error message

`traceback.print_exc()`

File names, line numbers, call chain, exception type and message

If the error is never caught, Python normally prints a traceback automatically and the program fails

Choosing a response for invalid CSV data

A handler is correct only if the resulting program behavior matches the requirements of the application

IF A BAD ROW CAN BE DISCARDED

Catch the error, report it and use continue.

Keep previously accepted rows and process the remaining CSV.

IF THE USER MUST DECIDE

Explain the problem and ask whether to skip or quit.

Use return to leave the function safely when quitting.

IF THE REPORT CANNOT CONTINUE

Stop with return or `sys.exit(1)`, or `raise` to delegate the decision.

IF YOU NEED DIAGNOSTICS

Print the message for users; print a traceback when debugging failures.

Disclaimer: the previous running example is a bit artificial for didactic purposes; a good initial input validation will solve much of the problem

Choosing a response for invalid CSV data

Diagnostic: Print the message for users VS print a traceback when debugging failures

Disable traceback for normal usage --> Short simple message for the user
Enable traceback for debugging --> full diagnostic details for the developer

```python
import traceback

DEBUG = False

try:
    result = 10 / 0

except ZeroDivisionError as e:
    print(f"Error: {e}")

    if DEBUG:
        traceback.print_exc()
```

Use a debug flag:
DEBUG = False, show only: *Error: division by zero*
DEBUG = True, show full traceback

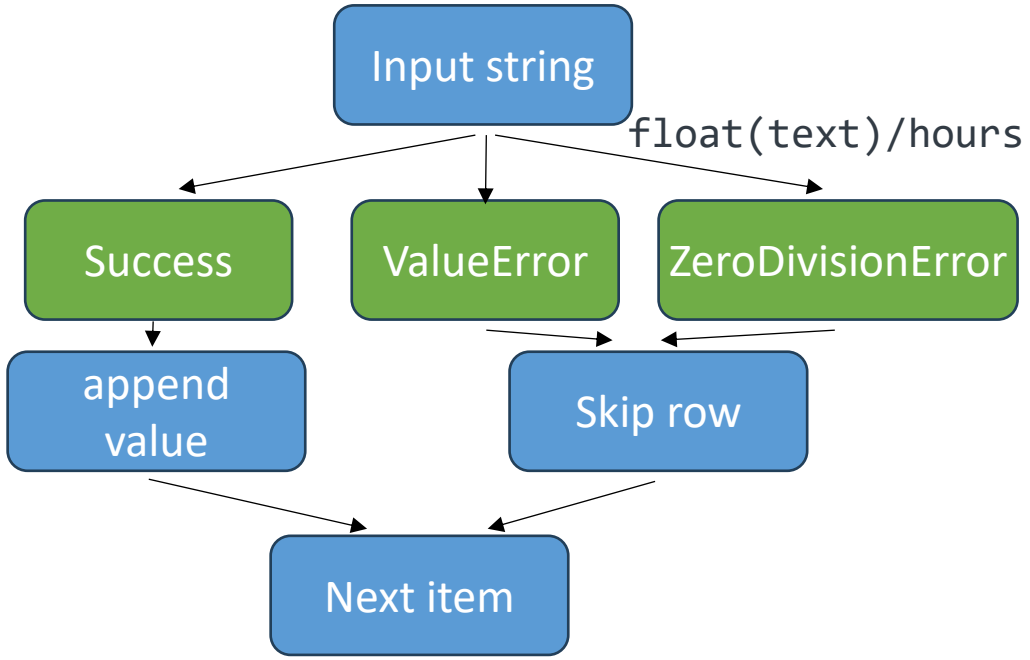
```python
parser.add_argument("--debug", action="store_true")

...

if args.debug:
    traceback.print_exc()
```

Exception handling flowchart

- Read/parse one value
- If conversion succeeds: append the numeric result
- If a known conversion failure occurs: record it and continue
- After processing: report both valid results and skipped records.



This is controlled recovery, not concealment of unexpected bugs.

Common error types

Error / Exception	Meaning	Example
SyntaxError	Invalid Python syntax	`if x > 0 print(x)`
IndentationError	Missing or incorrect indentation	`if True:` followed by an unindented `print()`
NameError	A variable or name is not defined	`print(total)` when total is undefined
TypeError	An operation receives an incompatible type	`"10" + 5`
ValueError	Correct type of argument, but invalid value	`int("hello")`
ZeroDivisionError	Division or modulo by zero	`10 / 0`
IndexError	List index outside the valid range	`[10, 20][5]`
KeyError	Dictionary key does not exist	`{"a": 1}["b"]`
FileNotFoundError	A requested file cannot be found	`open("missing.csv")`
ModuleNotFoundError	A requested module cannot be found	`import nonexistent_module`
AssertionError	An assert condition evaluates to False	`assert 2 + 2 == 5`

Debugging and testing

Error Handling and Debugging · Lecture 13

Debugging is an investigation

Step	What to do	Why
1. Reproduce	Find a simple input that consistently produces the problem	To observe the same failure repeatedly
2. Compare	Identify what the program should do and what it actually does	To understand the incorrect behavior
3. Investigate	Examine the traceback, relevant variables and intermediate results	To locate the instruction and understand the cause
4. Fix	Change the suspected cause and run the original test again	To verify whether the correction works
5. Retest	Test normal inputs and boundary cases; keep tests for future checks	To identify remaining problems and prevent regressions

Avoid random edits made without a testable hypothesis.

Logic error: wrong denominator

- A **logic error** may generate a plausible-looking number
- A three-element list [10,20,30] has sum 60 and mean 20
- Dividing by `len(values)-1` yields 30 instead
- Compare with a manually computed reference before using bigger datasets.

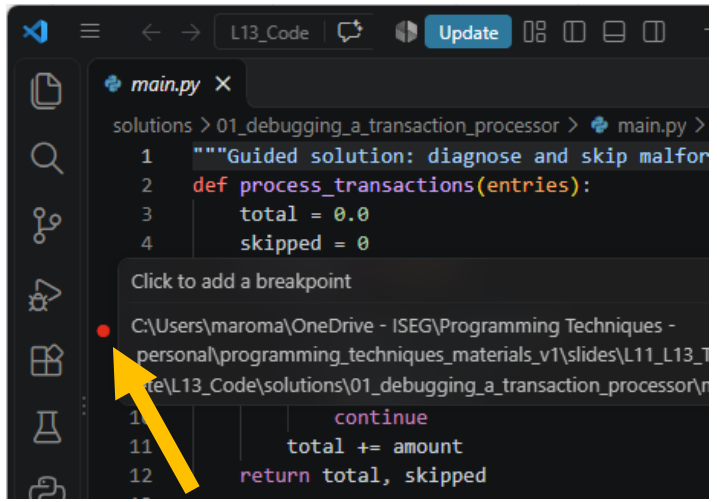
```
def average(values):
    return sum(values) / (len(values) - 1)

# [10, 20, 30] → wrong 30
# Correct denominator: len(values)
```

Inputs with one item also expose division-by-zero here.

VS Code: start the debugger

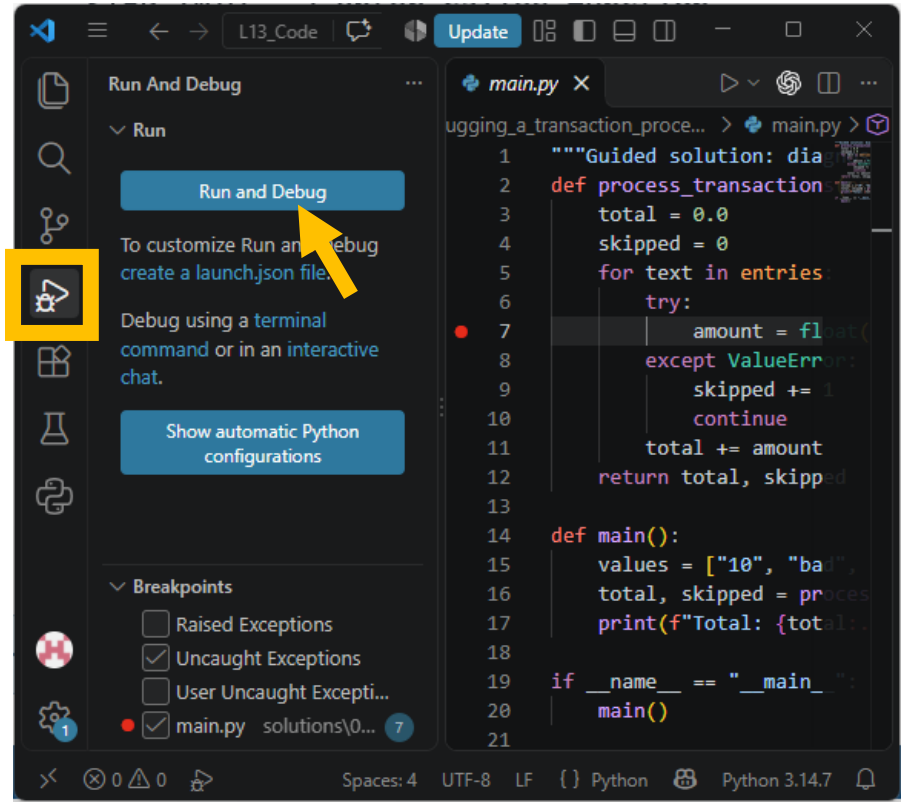
- Click beside a line number to set a **breakpoint**
- Start Debugging (F5) to pause program execution at that position
- Inspect Variables and Watch values before executing the next line
- Resume or step rather than adding many print statements.



The debugger shows the current state, not merely the final output.

VS Code: start the debugger

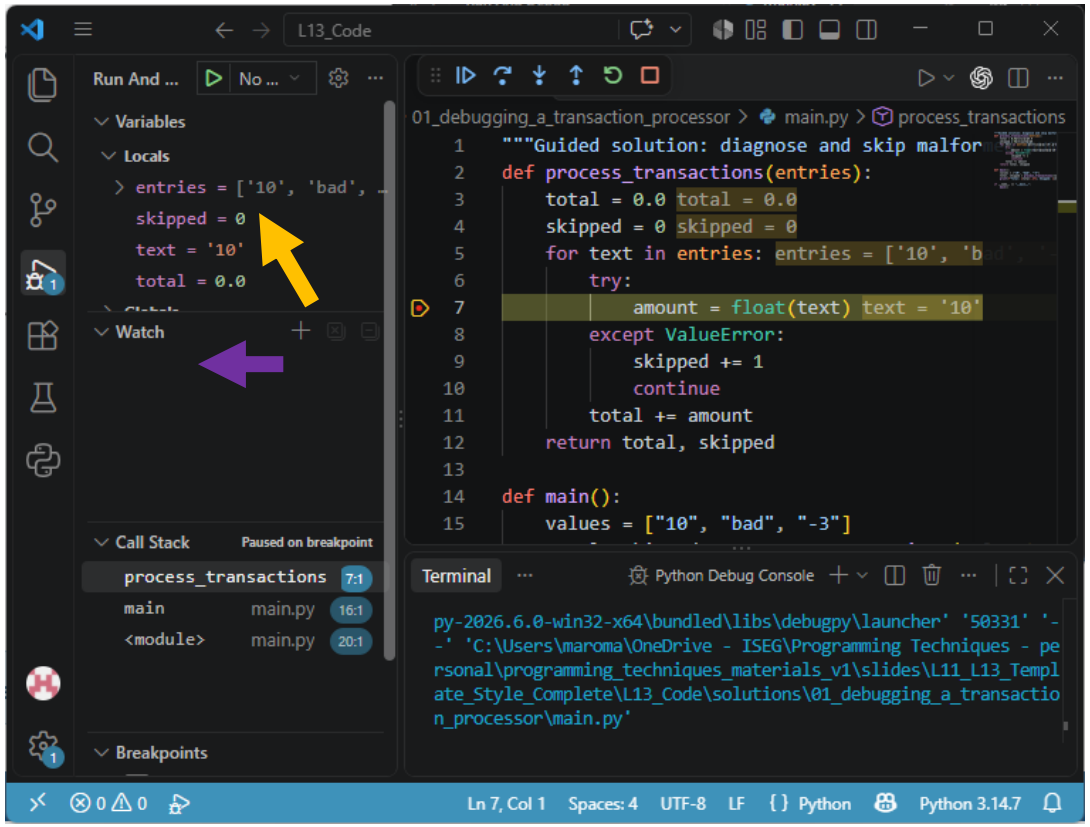
- Click beside a line number to set a **breakpoint**
- **Start Debugging** (F5) to pause program execution at that position
- Inspect Variables and Watch values before executing the next line
- Resume or step rather than adding many print statements.



The debugger shows the current state, not merely the final output.

VS Code: start the debugger

- Click beside a line number to set a **breakpoint**
- **Start Debugging** (F5) to pause program execution at that position
- Inspect **Variables** and **Watch** values before executing the next line (*watch* is a filtered selection of values you can observe separately)
- Resume or step rather than adding many print statements.

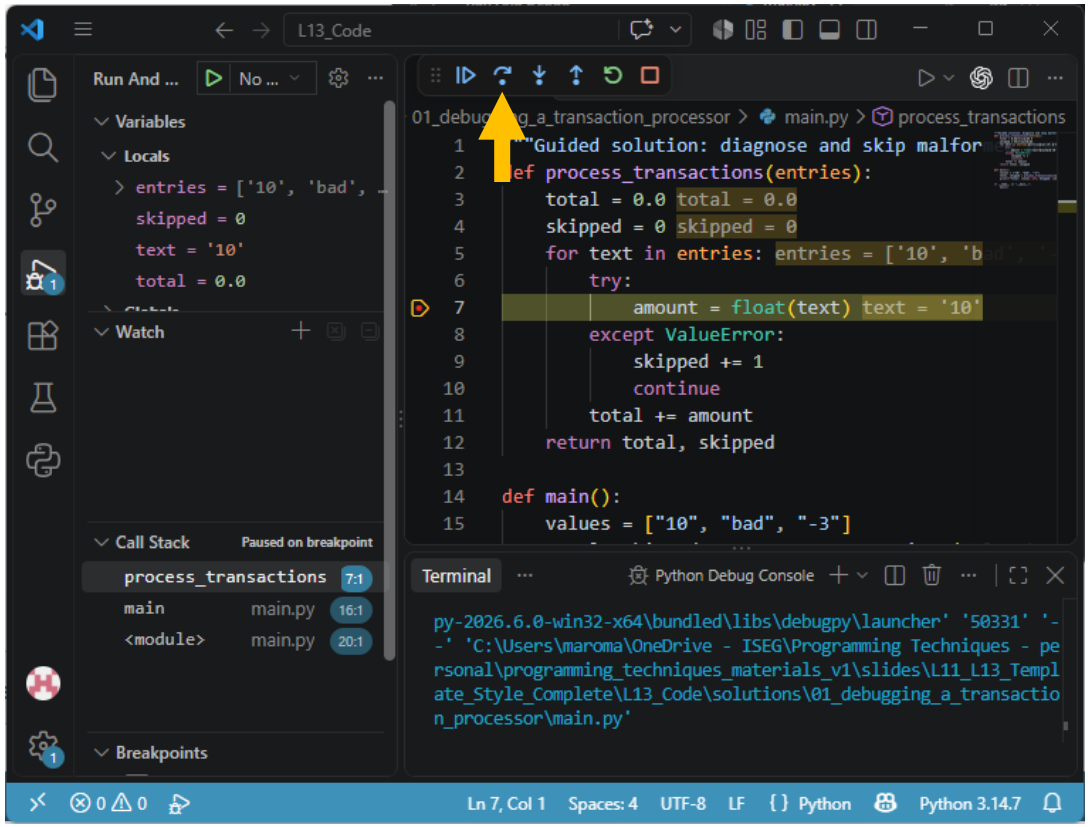


The debugger shows the current state, not merely the final output.

Step Over and Step Into

- **Step Over** executes a function call without entering its implementation
- **Step Into** pauses inside the called function
- **Step Out** returns to its caller
- **Continue** until next breakpoint
- Observe how local variables and the call stack change.

```
main()
value = compute_total(rows)
          ↑ Step Into enters compute_total
          ↑ Step Over executes call
```

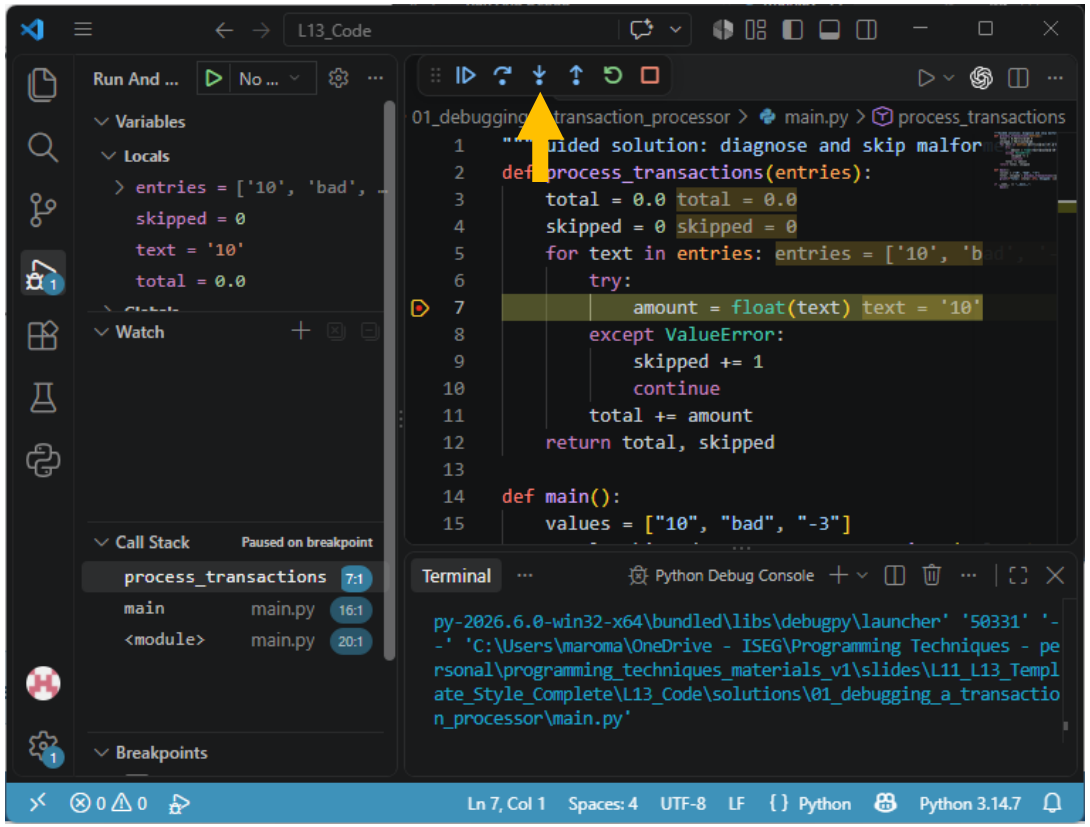


Use Step Into when the suspected defect is inside the function.

Step Over and Step Into

- **Step Over** executes a function call without entering its implementation
- **Step Into** pauses inside the called function
- **Step Out** returns to its caller
- **Continue** until next breakpoint
- Observe how local variables and the call stack change.

```
main()
value = compute_total(rows)
          ↑ Step Into enters compute_total
          ↑ Step Over executes call
```

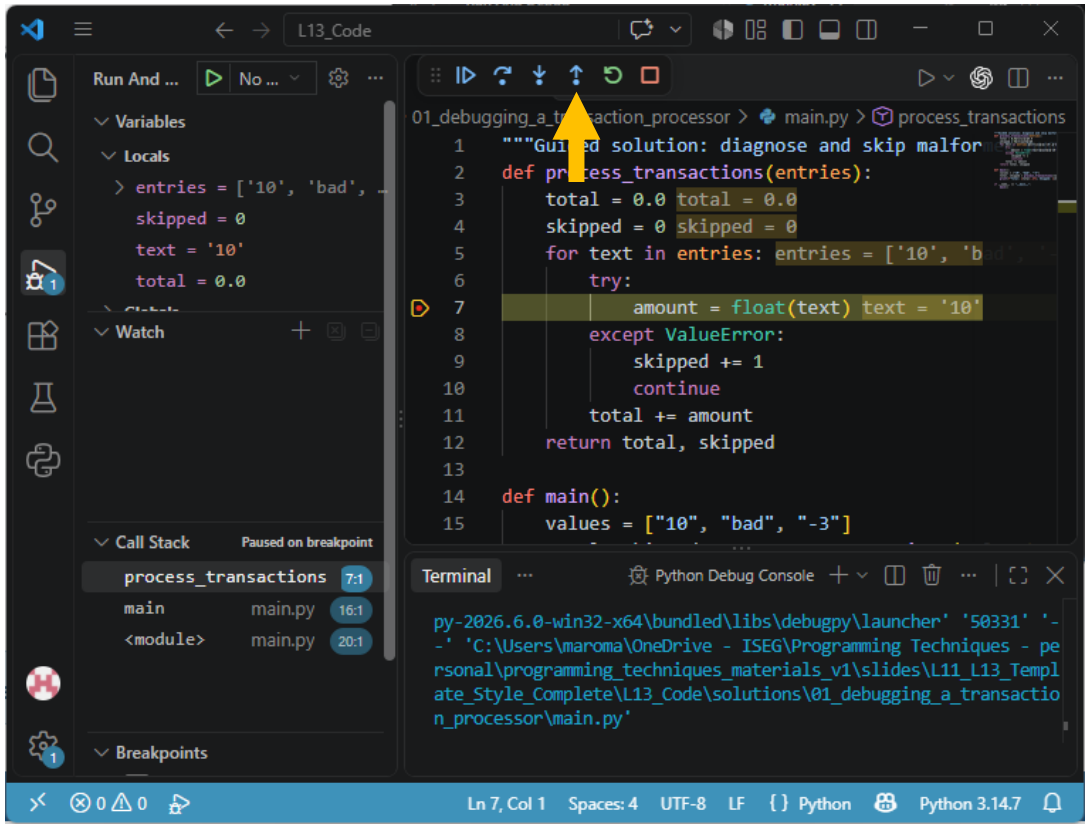


Use Step Into when the suspected defect is inside the function.

Step Over and Step Into

- **Step Over** executes a function call without entering its implementation
- **Step Into** pauses inside the called function
- **Step Out** returns to its caller
- **Continue** until next breakpoint
- Observe how local variables and the call stack change.

```
main()
value = compute_total(rows)
          ↑ Step Into enters compute_total
          ↑ Step Over executes call
```

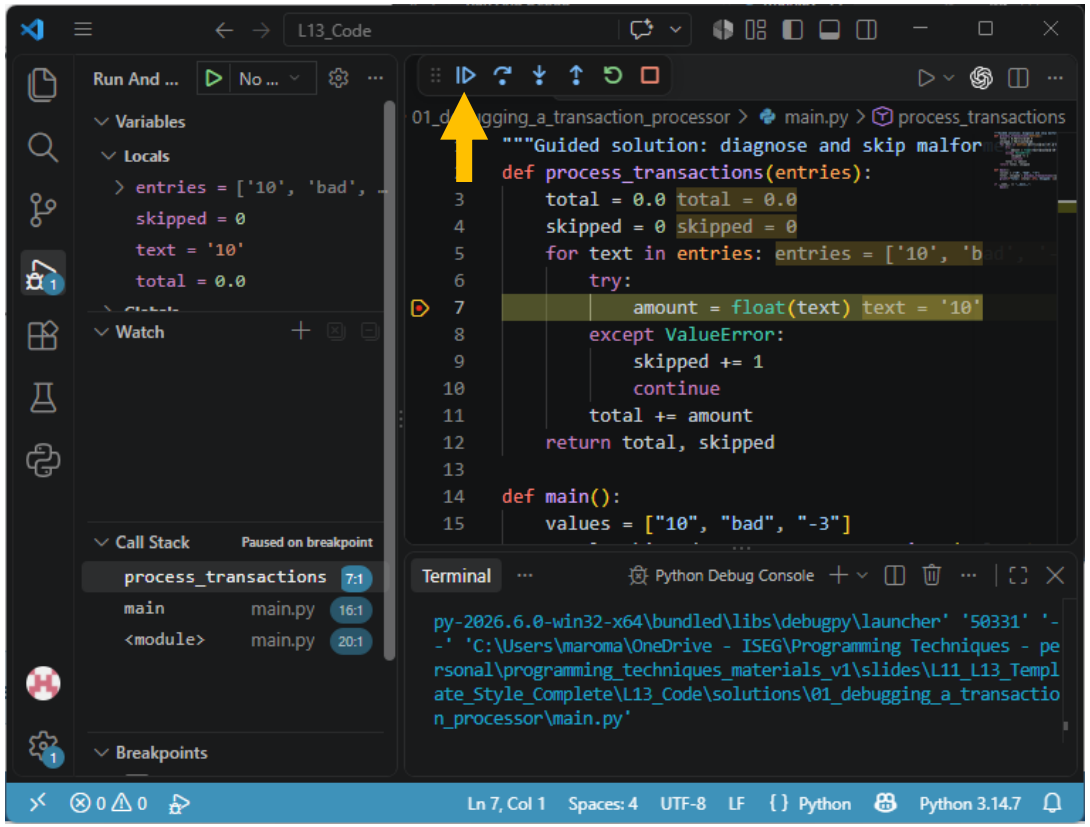


Use Step Into when the suspected defect is inside the function.

Step Over and Step Into

- **Step Over** executes a function call without entering its implementation
- **Step Into** pauses inside the called function
- **Step Out** returns to its caller
- **Continue** until next breakpoint
- Observe how local variables and the call stack change.

```
main()
value = compute_total(rows)
          ↑ Step Into enters compute_total
          ↑ Step Over executes call
```

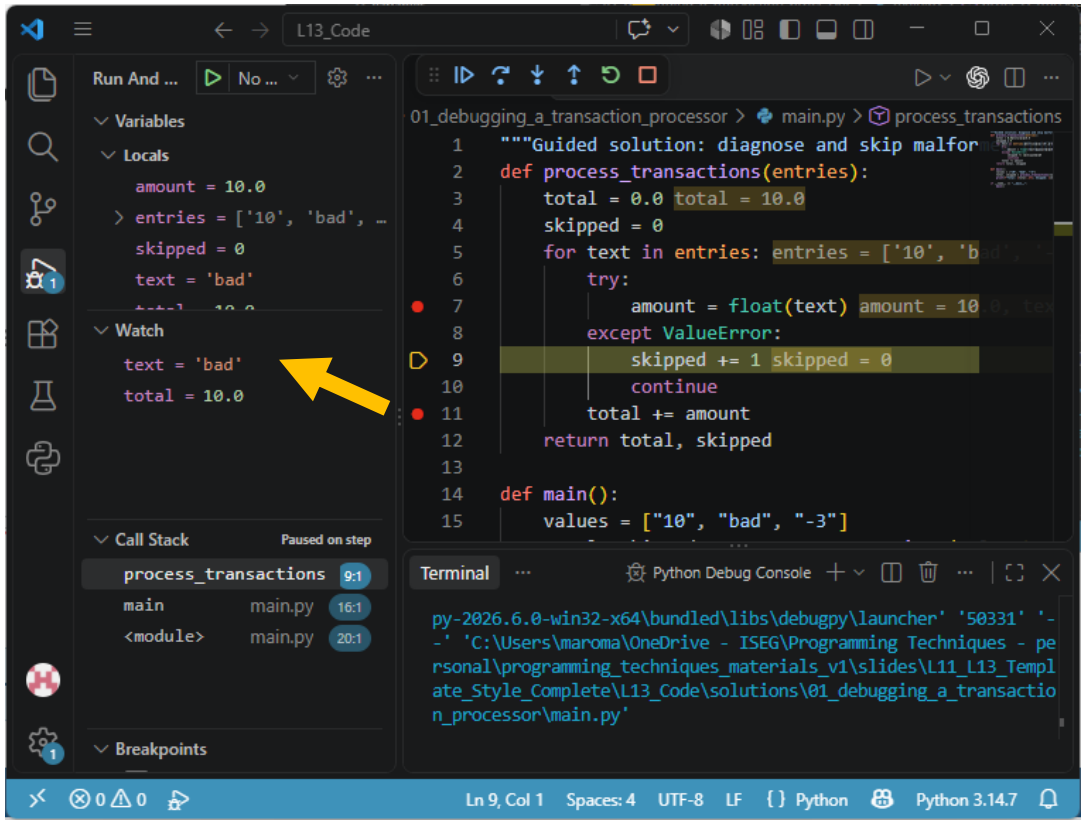


Use Step Into when the suspected defect is inside the function.

Step Over and Step Into

- **Step Over** executes a function call without entering its implementation
- **Step Into** pauses inside the called function
- **Step Out** returns to its caller
- **Continue** until next breakpoint
- Observe how local variables and the call stack change.

```
main()
value = compute_total(rows)
          ↑ Step Into enters compute_total
          ↑ Step Over executes call
```



Use Step Into when the suspected defect is inside the function.

Construct a test table

- For every scenario, record input and expected result first
- Run the code and record what actually happens
- Cover normal, boundary and invalid inputs
- A failing test demonstrates a discrepancy, not necessarily its root cause

CASE	EXPECTED
`[2, -1, 3]`	`total=4`
`[5]`	`mean=5`
`[]`	`ValueError`

Expected outputs should come from the specification, not from code.

Boundary cases are informative

- Test empty collections and lists of one element
- Test zero, negative values and values close to thresholds
- For files test a missing path and a wrong header
- For parsing test blanks, spaces and non-numeric strings.

```
"12"    → 12.0
" -2 "  → -2.0
"NA"    → ValueError
""      → ValueError
```

A handful of chosen examples is more useful than many redundant ones.

`assert`: checking internal assumptions

ADVANCED EXTENSION · simple demonstrations only

- assert condition raises `AssertionError` if the condition is false
- It is useful for development checks and simple test examples
- It must not be used to validate untrusted user input
- Assertions may be disabled when Python runs with optimizations.

```python
def calculate_speed(distance, hours):
    return distance / hours


# Tests performed during development
assert calculate_speed(100, 2) == 50, "incorrect speed calculation"
assert calculate_speed(150, 3) == 50, "incorrect speed calculation"
assert calculate_speed(0, 2) == 0, "incorrect zero speed calculation"

print("All tests passed!")
```