Module 1 of 20

Python and the interpreter

What CPython is and how it runs a file, the REPL, scripts and -m, the anatomy of a program, the console I/O patterns every exercise uses, and how to read a syntax error and a traceback.

Python is the language most people can read before they have written a line of it, and this module explains why that is a design rather than an accident: an interpreter that compiles to bytecode and executes it, an object model in which every value is an object and every name a label, indentation that is the block structure rather than decoration, and a small set of conventions — the main guard, imports at the top, one function per job — that make a file look the same whoever wrote it. It fixes the vocabulary the rest of the track relies on: statement and expression, block, docstring, module, exit code, traceback.

The five lessons move from the language to the machine and back. What Python is settles dynamic and strong typing, "everything is an object", the version timeline and what this track's CPython 3.11 runtime does and does not have. Running Python covers the REPL, scripts, `python -m`, `__name__`, exit codes, and precisely how the study judge runs a program. Anatomy of a program names the parts. Console I/O sets out the five reading patterns that cover every exercise input format and the `print` and f-string idioms that produce the expected output exactly. Errors and tracebacks teaches the bottom-up reading of a traceback, the eight exceptions of the first month, and the debugging habit of `print(repr(x))` on standard error.

Every exercise is a whole program that reads standard input and prints standard output — a greeting that counts letters, a feature checker that compares the running interpreter's version to a table, a classifier of interpreter invocations, a script built around a `main()` guard, an indentation checker and a line classifier, a summary of a count-then-values input, a line numberer that reverses words, an exception namer that evaluates expressions, and a traceback parser. The checkpoint adds a receipt with fixed-decimal totals, a calculator that reports exception names instead of crashing, and a word-count utility that reads to the end of input.

CONTENTS

1	What Python is, and why it looks the way it does	12 min
2	Running Python — the REPL, scripts, modules and the judge	12 min

3	Anatomy of a Python program	13 min
4	Console input and output — the patterns every exercise uses	14 min
5	Errors and tracebacks — reading what the interpreter tells you	13 min
✓	Checkpoint — Python and the interpreter	22 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

What Python is, and why it looks the way it does

Python is the language you can read before you can write, and that is not an accident of taste but the design goal it was built around. Guido van Rossum started it in 1989 as a scripting language for a distributed operating system, released it in 1991, and kept one principle above all others: code is read far more often than it is written, so the language should make the common case obvious. Thirty-five years later Python runs data pipelines, web backends, machine-learning research, build systems and a large share of every technical interview. This lesson settles what Python actually is — an interpreter, an object model and a set of conventions — so that everything later in the track has somewhere to hang.

An interpreted, compiled, dynamic language

"Interpreted" is half the truth. When you run `python program.py`, CPython — the reference implementation, written in C, and the one this track runs on — first **compiles** your source to *bytecode*, a compact instruction set for a stack machine, and then **interprets** that bytecode in a loop. The compile step is why a syntax error anywhere in the file stops the program before the first line runs, and why a `.pyc` file appears in `__pycache__` when a module is imported: the bytecode is cached so the compile step is skipped next time.

```
import dis

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

dis.dis(add)
```

```
  4           0 RESUME                   0

  5           2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
```

Those five instructions are what the interpreter executes. There is no type in `BINARY_OP`: at run time it looks at the two objects on the stack and asks the left one whether it knows how to

add the right one. That is **dynamic typing** — types belong to objects, not to names — and it is the source of both Python's flexibility and its most common bugs. Python is also **strongly** typed: `"1" + 1` raises `TypeError` rather than guessing; nothing is silently coerced except numbers among themselves.

Everything is an object

An integer, a string, a function, a class, a module, `None` — every value in Python is an object with an identity, a type and a value. Names are labels bound to objects; assignment attaches a label and never copies anything. The consequences are the subject of Module 2, but the picture to hold from the first day is this:

```
xs = [1, 2, 3]
ys = xs          # a second name for the same list
ys.append(4)
print(xs)        # [1, 2, 3, 4]
```

Two names, one object. Learners who imagine variables as boxes holding values are surprised here; learners who imagine names as sticky notes on objects are not.

Because everything is an object, everything can be inspected and passed around: a function is an argument like any other, a class can be created at run time, and the operators you type (`+`, `[]`, `len()`) are calls to methods with double-underscore names — `__add__`, `__getitem__`, `__len__` — that your own classes can define. This is the *data model*, and Module 8 and Module 16 are about it.

Indentation is syntax

Python has no braces. A block is the set of lines indented under the line that ends in a colon, and that indentation is not a style choice but the grammar:

```
def sign(n):
    if n < 0:
        return "negative"
    elif n == 0:
        return "zero"
    return "positive"
```

Four spaces per level is the universal convention (PEP 8), tabs and spaces cannot be mixed in one block, and an `IndentationError` is a compile-time error like any other. The design forces the code to look like its structure — the indentation you would have added for readability in another language is the only indentation there is.

The versions that matter

Python 2 and Python 3 were incompatible for a decade; Python 2 ended in 2020 and nothing in this track concerns it. Within Python 3, a new minor version ships every October and each is supported for five years. The features you will meet by version:

Version	What it added
3.6	f-strings, underscores in numeric literals
3.7	dataclasses, dicts guaranteed to keep insertion order
3.8	the walrus operator `:=`, positional-only parameters `/`
3.9	`list[int]` generics without `typing`, dict union `|`, `str.removeprefix`
3.10	structural pattern matching (`match`), `X | Y` union types, `zip(strict=True)`
3.11	exception groups and `except*`, `tomllib`, `typing.Self`, `asyncio.TaskGroup`, a 10–60 % faster interpreter
3.12	the `type` statement, PEP 695 generics `def f[T](x: T)`, `itertools.batched`
3.13	an experimental free-threaded build without the GIL, a new REPL

This track runs on CPython 3.11. Everything through 3.11 is fair game in an exercise; 3.12 and later features are described so you recognise them, and marked as reading only.

CPython and the others

CPython is the implementation everyone means by "Python" and the one that defines the language in practice. Others exist for particular reasons: PyPy, a just-in-time compiler that runs pure-Python loops several times faster; MicroPython for microcontrollers; and, historically, Jython and IronPython on the JVM and .NET. Two CPython facts shape how the language is used. Reference counting frees most objects the moment the last name to them goes away,

so a file closed by a `with` block or a list dropped at the end of a function is released immediately, not at some later collection. And the **global interpreter lock** (GIL) lets only one thread execute Python bytecode at a time, which is why CPU-bound work is spread over *processes* and threads are for waiting on I/O (Module 17).

Where Python is used, and where it is not

Python dominates data science and machine learning (NumPy, pandas, PyTorch — libraries whose hot loops are C and CUDA with a Python steering wheel), scripting and automation, web backends (Django, FastAPI, Flask), and teaching. It is the most common interview language for the same reason it is the most common teaching language: the code that expresses an algorithm is short and close to pseudocode. It is not the language for a game engine, a kernel or a hot inner loop — pure Python is roughly 20–100 times slower than C on such code — and its answer to that is not to compete but to call out: the fast parts are written in C, Rust or Cython, and Python composes them.

Pitfalls

- Believing "interpreted" means "no compile step". A syntax error on line 400 stops the whole file before line 1 runs.
- Thinking of a name as a typed box. `x = 5` then `x = "five"` is legal; the name has no type, the objects do.
- Mixing tabs and spaces, or copying code from a page that changed the indentation. The interpreter refuses both.
- Reading Python 2 answers. `print "x"`, `xrange`, `raw_input` and integer division with `/` are all gone.

Key takeaways

- CPython compiles source to bytecode and interprets it; a syntax error anywhere stops everything.
- Types belong to objects, not names; Python is dynamically and strongly typed.
- Everything is an object, names are labels, and assignment never copies.
- Indentation is the block structure — four spaces, never mixed with tabs.

This track runs on CPython 3.11; 3.12+ features are taught as reading only.

EXERCISE 1

Hello, whoever you are

Read one line holding a name (it may have spaces around it) and print a greeting followed by how many **letters** the name contains — characters for which `str.isalpha()` is true, so digits, spaces and hyphens do not count. Use `letter` when the count is exactly 1 and `letters` otherwise.

Input: one line. **Output:** two lines: `Hello, <name>!` with the surrounding whitespace removed, then `Your name has <n> letters.`

```
Grace Hopper
```

prints

```
Hello, Grace Hopper!
Your name has 11 letters.
```

INPUT 1

```
Grace Hopper
```

OUTPUT 1

```
Hello, Grace Hopper!
Your name has 11 letters.
```

INPUT 2

```
Ada
```

OUTPUT 2

```
Hello, Ada!
Your name has 3 letters.
```

REFERENCE SOLUTION

```python
name = input().strip()
letters = sum(1 for c in name if c.isalpha())
print(f"Hello, {name}!")
noun = "letter" if letters == 1 else "letters"
print(f"Your name has {letters} {noun}.")
```

EXERCISE 2

Does this interpreter have it?

The interpreter you are running on has a version, and every language feature arrived in a particular one. The starter holds a table of features and the `(major, minor)` version that introduced each.

Read `n` feature names and, for each, compare the table entry with the running interpreter's `sys.version_info` and report whether the feature is available.

Input: `n`, then `n` lines each holding a feature name. **Output:** one line per feature: `<name>: yes (<major>.<minor>)` when the interpreter's version is at least the feature's, `<name>: no (needs <major>.<minor>)` when it is older, and `<name>: unknown` for a name not in the table.

```
3
match
type-statement
walrus
```

prints (on CPython 3.11)

```
match: yes (3.10)
type-statement: no (needs 3.12)
walrus: yes (3.8)
```

INPUT 1

```
3
match
type-statement
walrus
```

OUTPUT 1

```
match: yes (3.10)
type-statement: no (needs 3.12)
walrus: yes (3.8)
```

INPUT 2

```
2
except-star
free-threading
```

OUTPUT 2

```
except-star: yes (3.11)
free-threading: no (needs 3.13)
```

REFERENCE SOLUTION

```python
import sys

FEATURES = {
    "f-strings": (3, 6),
    "dataclasses": (3, 7),
    "walrus": (3, 8),
    "match": (3, 10),
    "except-star": (3, 11),
    "type-statement": (3, 12),
    "batched": (3, 12),
    "free-threading": (3, 13),
}

running = (sys.version_info.major, sys.version_info.minor)

n = int(input())
for _ in range(n):
    name = input().strip()
    if name not in FEATURES:
        print(f"{name}: unknown")
        continue
    major, minor = FEATURES[name]
    if running >= (major, minor):
        print(f"{name}: yes ({major}.{minor})")
    else:
        print(f"{name}: no (needs {major}.{minor})")
```

QUIZ

1. What does CPython do with `program.py` before the first statement runs?

A. Nothing — it reads and executes one line at a time, so a syntax error on line 40 only matters when line 40 is reached

B. It compiles the whole file to bytecode; a syntax error anywhere stops the program before line 1 runs

C. It compiles the file to native machine code

D. It type-checks every variable and refuses to run if a name changes type

2. What does this print?

```
x = 5
x = "five"
print(type(x).__name__)
```

A. `int`

B. `str`

C. It raises `TypeError` because `x` is already an `int`

D. `object`

3. What does `"1" + 1` do?

A. Produces `"11"`

B. Produces `2`

C. Raises `TypeError`

D. Produces `"2"`

4. What does this print?

```
a = [1]
b = a
b.append(2)
print(a)
```

A. `[1]`

B. `[1, 2]`

C. `[2]`

D. It raises an error because `a` was not modified through its own name

5. Which statement about indentation in Python is correct?

A. Indentation is a style convention; braces define blocks

B. Indentation defines blocks, and mixing tabs and spaces within one block is an error

C. Any indentation works as long as each block uses exactly four spaces

D. Indentation matters only inside functions

ANSWER KEY · WHAT PYTHON IS, AND WHY IT LOOKS THE WAY IT DOES

1. B Compilation to bytecode is a whole-file step, which is why a syntax error at the end of a file prevents the beginning from running. There is no native code and no static type check: types belong to objects and are examined only when an operation runs.

2. B A name has no type; it is rebound to whatever object is assigned. After the second assignment `x` labels a `str`. Nothing about the first binding constrains the second.

3. C Python is strongly typed: a `str` and an `int` are not silently converted to each other. `str.__add__` refuses an `int` and the result is `TypeError: can only concatenate str (not "int") to str`. Convert explicitly with `int("1") + 1` or `"1" + str(1)`.

4. B Assignment binds a second name to the same list object; it copies nothing. Appending through `b` changes the one list that `a` also labels. To get an independent copy you would write `b = a.copy()` or `b = list(a)`.

5. B There are no braces: the indented lines after a colon are the block. The interpreter accepts any consistent width per block but refuses a tab/space mix with `TabError`; four spaces is the PEP 8 convention, not a rule the grammar enforces.

LESSON 2 · 12 MIN

Running Python — the REPL, scripts, modules and the judge

There are four ways to hand code to the interpreter, and knowing which one you are in explains a whole class of "it works here but not there" confusions: the REPL, a script file, a module run with `-m`, and a one-liner with `-c`. This lesson walks through each, fixes the `if __name__ == "__main__":` idiom that every serious script uses, explains exit codes, and describes exactly how the study judge runs the programs you write in this track.

The REPL

Type `python` (or `python3` on systems where `python` is still Python 2) with no arguments and you get the read–eval–print loop:

```
$ python
Python 3.11.13 (main) [GCC 13.3.0] on linux
>>> 2 ** 10
1024
>>> name = "Ada"
>>> f"Hello, {name}"
'Hello, Ada'
>>> _
'Hello, Ada'
```

The REPL evaluates each line and prints the value of an expression automatically, using `repr` — which is why the string comes back with quotes. `_` holds the last printed value. Statements (`name = "Ada"`) print nothing. The REPL is where you check what a method does or what an expression is before committing it to a file; `help(str.split)` and `dir(str)` answer most questions faster than a search. Leave with `exit()` or Ctrl-D.

Scripts

A script is a file run top to bottom:

```
$ python greet.py
```

```
# greet.py
name = "Ada"
print(f"Hello, {name}")
```

Nothing is printed unless you `print` it — the REPL's automatic echo is a REPL feature. The interpreter compiles the whole file first (a syntax error on any line stops everything), then executes statements in order. A `def` or `class` statement *defines* a function or class when it is reached; the body runs only when called. This is why a script that calls a function before its `def` fails with `NameError`: at the time of the call, the name has not been bound yet.

Modules and `-m`

Every Python file is a *module*; importing it runs its top-level code once and binds its names. `python -m package.module` runs a module *as a script* by its import name rather than its file path, which is how the standard library's tools are invoked:

```
$ python -m json.tool data.json      # pretty-print JSON
$ python -m http.server 8000         # serve the current directory
$ python -m venv .venv               # create a virtual environment
$ python -m pip install requests     # pip for *this* interpreter
```

The `-m` form finds the module on the import path and sets things up as if it were the main script, which matters for packages with relative imports (Module 12). `python -c "print(2 ** 64)"` runs a one-liner; `python -i script.py` runs the script and then drops into the REPL with its names still defined — the quickest way to poke at a program's state.

`__name__` and the main guard

When a file runs as a script, the interpreter sets the module's `__name__` to the string `"__main__"`. When the same file is imported, `__name__` is the module's name (`"greet"`). The idiom that keeps a file usable both ways:

```python
def total(nums):
    return sum(nums)

def main():
    nums = list(map(int, input().split()))
    print(total(nums))

if __name__ == "__main__":
    main()
```

Run as a script, `main()` executes. Imported by a test file that wants `total`, it does not — no input is read, nothing is printed. Two habits come with the idiom: put the program's work in a `main()` function rather than at top level, so that the names it creates are local and the file has one obvious entry point; and keep the guard at the bottom, after every definition it needs.

Exit codes

A process ends with an integer status: `0` means success, anything else means failure, and shells, CI systems and the study judge read it. A Python program that reaches the end of the file exits `0`. `sys.exit(2)` exits with `2`; `sys.exit("message")` prints the message to stderr and exits `1`; an uncaught exception prints a traceback to stderr and exits `1`.

```python
import sys

if len(sys.argv) < 2:
    print("usage: tool.py <file>", file=sys.stderr)
    sys.exit(2)
```

Errors go to standard error, not standard output, so that a program whose output is piped somewhere does not corrupt it with messages.

How the judge runs your program

Every exercise in this track is a whole program. The judge writes your source to `Main.py`, runs it once per test case with that case's input on **standard input**, captures **standard output**, and compares it with the expected output after ignoring trailing spaces on each line and trailing blank lines. Three consequences:

Standard error is ignored for the comparison — but a non-zero exit code is a *runtime error* verdict, so an uncaught exception fails the case even if the right output was printed first, and `sys.exit(1)` fails it too.

Read with `input()` or `sys.stdin`, never with a prompt. `input("Enter n: ")` writes `Enter n:` to standard output, which becomes part of your answer and fails the comparison. Bare `input()` is the rule.

There is a time limit of about a second per case. Every exercise here is designed to run in milliseconds; a program that takes seconds has the wrong algorithm.

The interpreter is CPython 3.11 on Linux. Features newer than 3.11 do not exist there, and the lesson text marks them.

Useful flags

Flag	Effect
`-i`	Enter the REPL after the script finishes (or fails)
`-c "code"`	Run the string as a program
`-m mod`	Run a module as a script
`-X dev`	Development mode: extra warnings, `ResourceWarning` for unclosed files
`-O`	Strip `assert` statements (Module 10 explains why that matters)
`-u`	Unbuffered stdout, so prints appear immediately when piped
`-W error`	Turn warnings into exceptions
`--version`	Print the version and exit

Pitfalls

Expecting a script to echo expression values like the REPL does. Only `print` prints in a script.

Calling a function above its `def`. Definitions are executed in order; the call must come after.

Putting work at module top level in a file you also import. The main guard exists for this.

Using `input("prompt")` in a judged program. The prompt is output.

`python` pointing at a different interpreter than `pip`. `python -m pip` always matches.

Key takeaways

The REPL echoes expression values; scripts print only what you `print`.

`python -m module` runs a module as a script by its import name; `-c` runs a string; `-i` keeps the REPL open afterwards.

`__name__ == "__main__"` is true only when the file runs as the script; keep the work in `main()` under that guard.

Exit `0` is success; an uncaught exception or `sys.exit(non-zero)` is failure — and a runtime error on the judge.

Judged programs read from stdin with bare `input()` / `sys.stdin`, print to stdout, and finish within a second.

EXERCISE 1

Classify the invocation

Read `n` command lines, each starting with `python` or `python3`, and say how the interpreter would run: with no arguments it opens the **REPL**; `-c` runs a **command**; `-m <name>` runs a **module**; `-i <file>` runs a **script** and then opens the REPL; a bare file name runs a **script**. Anything after the module or file name (extra arguments) is ignored.

Input: `n`, then `n` command lines. **Output:** one line per command: `REPL`, `command`, `module <name>`, `script <file> then REPL` or `script <file>`.

```
4
python app.py
python -m json.tool
python
python -c "print(1)"
```

prints

```
script app.py
module json.tool
REPL
command
```

INPUT 1

```
4
python app.py
python -m json.tool
python
python -c "print(1)"
```

OUTPUT 1

```
script app.py
module json.tool
REPL
command
```

REFERENCE SOLUTION

```python
n = int(input())
for _ in range(n):
    parts = input().split()
    args = parts[1:]
    if not args:
        print("REPL")
    elif args[0] == "-c":
        print("command")
    elif args[0] == "-m":
        print(f"module {args[1]}")
    elif args[0] == "-i":
        print(f"script {args[1]} then REPL")
    else:
        print(f"script {args[0]}")
```

EXERCISE 2

A script with a main guard

Write a program in the shape every serious script has: a function `total(nums)` that returns the sum of a list of integers, a `main()` that does all the reading and printing, and the `if __name__ == "__main__":` guard as the only top-level statement that runs code. `main()` reads lines until the end of input; each line holds integers separated by spaces. Print the sum of each line, then a final line `total: <sum of everything>`.

Input: zero or more lines of integers. **Output:** one sum per input line, then `total: <n>`.

```
1 2 3
10 20
```

prints

```
6
30
total: 36
```

INPUT 1

```
1 2 3
10 20
```

OUTPUT 1

```
6
30
total: 36
```

INPUT 2

```
-5
```

OUTPUT 2

```
-5
total: -5
```

REFERENCE SOLUTION

```python
import sys


def total(nums):
    return sum(nums)


def main():
    grand = 0
    for line in sys.stdin:
        nums = [int(tok) for tok in line.split()]
        line_sum = total(nums)
        grand += line_sum
        print(line_sum)
    print(f"total: {grand}")


if __name__ == "__main__":
    main()
```

QUIZ

1. What does `python -m json.tool data.json` do?

A. Imports `json.tool` into the REPL and waits for input

B. Runs the `json.tool` module as a script, found by its import name rather than a file path

C. Compiles `json/tool.py` to bytecode without running it

D. Runs `data.json` as a Python program using the `json.tool` interpreter

2. When is the expression `__name__ == "__main__"` true?

A. Whenever a function called `main` is defined in the file

B. When the file is being run directly as the program, rather than imported

C. Always — every module's `__name__` is `"__main__"`

D. Only when the file is named `main.py`

3. This file is run with `python calc.py`. What is printed?

```
print(2 + 2)
3 + 3
```

A. `4` then `6`

B. `4` only

C. `6` only

D. Nothing — expression statements are errors in a script

4. A judged program prints the correct answer and then calls `sys.exit(3)`. What verdict does it get?

A. Accepted — the output matched

B. Wrong answer — `3` is appended to the output

C. Runtime error — a non-zero exit status is a failure regardless of what was printed

D. Time limit exceeded

5. What happens on the judge when a program reads with `n = int(input("Enter n: "))`?

A. Nothing special — prompts go to stderr, which is ignored

B. The prompt `Enter n:` is written to standard output and becomes part of the answer, which then does not match

C. It raises `EOFError` because the judge cannot answer prompts

D. The prompt is shown to the learner in the editor and the program waits

ANSWER KEY · RUNNING PYTHON — THE REPL, SCRIPTS, MODULES AND THE JUDGE

1. B `-m` locates a module on the import path and runs it as the main program, which is how the standard library's command-line tools (`json.tool`, `http.server`, `venv`, `pip`) are invoked. Nothing is compiled without running, and JSON is data, not a program.

2. B The interpreter sets the running script's `__name__` to `"__main__"`; an imported module's `__name__` is its module name. The file's name and whether it defines `main()` are irrelevant. That is what lets one file be both a script and an importable module.

3. B Only the REPL echoes expression values. In a script `3 + 3` is evaluated and discarded, so the only output is the `print`. It is not an error: an expression statement is legal, just useless here.

4. C The judge reads the exit status as well as stdout. Anything other than `0` — `sys.exit(3)`, an uncaught exception — is reported as a runtime error even when the right text was printed first. `sys.exit` does not print the number.

5. B `input(prompt)` writes the prompt to stdout, not stderr, before reading. The judge compares everything on stdout, so the answer is prefixed with the prompt text and fails as a wrong answer. Bare `input()` is the rule.

LESSON 3 · 13 MIN

Anatomy of a Python program

A Python file is a sequence of statements, some of which introduce indented blocks, some of which are expressions with a value, and a few of which — `import`, `def`, `class` — bind names that the rest of the file uses. This lesson names the parts so that the vocabulary is fixed before the track uses it: statement and expression, block and suite, comment and docstring, name and keyword, and the shape a well-formed program has from its first import to its last line.

Statements and expressions

An **expression** is anything that produces a value: `2 + 3`, `len(xs)`, `name.upper()`, `x if x > 0 else -x`, a function call, a literal. A **statement** is an instruction: assignment (`x = 5`), `if`, `for`, `while`, `def`, `class`, `import`, `return`, `pass`, `del`. Every expression can stand alone as a statement (its value is thrown away, which is what a bare `xs.append(4)` line does), but a statement cannot appear where an expression is expected — you cannot write `y = if x: 1 else: 2`, and you cannot put an assignment inside a function call's arguments. The one exception is the walrus operator, `(n := len(xs))`, an *assignment expression* added in 3.8 precisely for the cases where a value must be both tested and kept (Module 2).

```
total = 0                # statement: assignment
for x in [1, 2, 3]:      # statement introducing a block
    total += x           # augmented assignment
print(total)             # expression statement: a call whose value is discar
```

Blocks and indentation

A line ending in a colon — `if`, `elif`, `else`, `for`, `while`, `def`, `class`, `try`, `except`, `finally`, `with`, `match`, `case` — opens a **block** (the grammar calls it a *suite*). The block is every following line indented further than the opening line, and it ends when the indentation returns. Four spaces per level; the interpreter accepts any consistent width, and refuses a mix of tabs and spaces within a block with `TabError`.

```
def describe(n):
    if n % 2 == 0:
        kind = "even"
    else:
        kind = "odd"
    return f"{n} is {kind}"     # back at one level: part of the function, aft
```

A block cannot be empty. When the body is intentionally nothing — a placeholder function, a class with no members yet — write `pass`, or a docstring, or `...` (the `Ellipsis` literal, which some codebases use for stubs).

Line structure

A statement normally ends at the end of the line. It continues onto the next line automatically inside open brackets — `(`, `[`, `{` — which is the preferred way to break a long call or literal:

```
result = compute(
    first_value,
    second_value,
    scale=2.5,
)
names = [
    "ada", "grace",
    "linus",
]
```

The trailing comma before a closing bracket is legal and conventional: adding a line later changes one line in the diff instead of two. A backslash at the end of a line also continues it, but is fragile (a space after it is a syntax error) and rarely needed. Two statements may share a line separated by `;`; style guides forbid it.

Comments and docstrings

`#` starts a comment to the end of the line; the interpreter ignores it. A **docstring** is a string literal that is the first statement of a module, function or class; it is not ignored — it is stored as `__doc__` and shown by `help()`:

```
def area(radius):
    """Return the area of a circle with the given radius."""
    return 3.141592653589793 * radius ** 2
```

Comments explain *why* — the reason for a non-obvious choice, the invariant a loop maintains — and docstrings explain *what* a function does for its caller. A comment that restates the code (`# increment i`) is noise. Module 15 returns to the conventions (PEP 257) that make docstrings useful to tools.

Names and keywords

A name is letters, digits and underscores, not starting with a digit, case-sensitive. Conventions carry meaning: `snake_case` for variables and functions, `PascalCase` for classes, `UPPER_CASE` for constants (which are ordinary names — Python has no `const`), a leading underscore `_internal` for "not part of the public interface", a trailing one `class_` to dodge a keyword. The 35 keywords — `if`, `else`, `for`, `while`, `def`, `class`, `return`, `import`, `from`, `as`, `in`, `is`, `not`, `and`, `or`, `None`, `True`, `False`, `lambda`, `yield`, `with`, `try`, `except`, `finally`, `raise`, `pass`, `break`, `continue`, `global`, `nonlocal`, `del`, `assert`, `async`, `await`, `elif` — cannot be names. `match` and `case` are *soft* keywords: they act as keywords only at the start of a match statement, so a variable called `match` still works. Built-in function names like `list`, `str`, `sum`, `id`, `input` are not keywords, which is why shadowing them is possible and a classic bug: `list = [1, 2]` and the `list()` constructor is gone for the rest of the scope.

The shape of a program

A readable program has a fixed order:

```python
"""Compute the average of the numbers on standard input."""
import sys                        # 1. imports: standard library, then third-pa

PRECISION = 2                     # 2. module-level constants

def mean(values):                 # 3. functions and classes
    return sum(values) / len(values)

def main():                       # 4. the entry point
    nums = [int(tok) for tok in sys.stdin.read().split()]
    print(f"{mean(nums):.{PRECISION}f}")

if __name__ == "__main__":        # 5. the guard, last
    main()
```

Imports at the top, so a missing dependency fails immediately and every reader knows what the file depends on; one function per job; the work under `main()`; the guard at the end. `import sys` binds the name `sys` to the module object; `from math import sqrt` binds just `sqrt`. `from module import *` binds an unknown set of names and is avoided outside the REPL.

Pitfalls

An empty block. `if x:` followed by a dedented line is a syntax error; use `pass`.

Shadowing a built-in: `sum = 0` then `sum(xs)` fails with `TypeError: 'int' object is not callable`.

A stray `;` or a backslash followed by a space.

Putting a docstring anywhere but first. A string in the middle of a function is a no-op expression statement.

Assuming `UPPER_CASE` is protected. It is a promise to readers, not to the interpreter.

Key takeaways

Expressions have values; statements do things; an expression can be a statement but not the reverse (except the walrus).

A colon opens a block; the block is the indented lines; `pass` fills an intentionally empty one.

Statements continue inside open brackets; prefer that to backslashes.

`#` comments are ignored; a first-statement string is a docstring, kept in `__doc__`.

Order a file: docstring, imports, constants, definitions, `main()`, the guard.

EXERCISE 1

Indentation check

Write a checker for the one rule the interpreter enforces before anything else. Read lines of source code until the end of input and check each line's leading whitespace: a line whose leading whitespace contains a tab is bad, and so is a line whose leading spaces are not a multiple of four. Blank lines (only whitespace) are ignored. Report the **first** bad line, or `ok` if there is none.

Input: zero or more lines of code. **Output:** `ok`, or `line <n>: tab`, or `line <n>: indent <k>` where `k` is the number of leading spaces (lines are numbered from 1).

```
def f():
   return 1
```

prints

```
line 2: indent 3
```

INPUT 1

```
def f():
    return 1
```

OUTPUT 1

```
ok
```

INPUT 2

```
def f():
   return 1
```

OUTPUT 2

```
line 2: indent 3
```

REFERENCE SOLUTION

```python
import sys

for number, line in enumerate(sys.stdin, start=1):
    line = line.rstrip("\n")
    if not line.strip():
        continue
    leading = line[: len(line) - len(line.lstrip())]
    if "\t" in leading:
        print(f"line {number}: tab")
        break
    if len(leading) % 4 != 0:
        print(f"line {number}: indent {len(leading)}")
        break
else:
    print("ok")
```

EXERCISE 2

Count the kinds of line

Read source lines until the end of input and classify each one: **blank** if it contains only whitespace, **comment** if its first non-whitespace character is `#`, **code** otherwise (a line of code with a trailing comment is code). Print the three counts.

Input: zero or more lines. **Output:** three lines: `code <n>`, `comment <n>`, `blank <n>`.

```
# header
x = 1

y = 2  # trailing
```

prints

```
code 2
comment 1
blank 1
```

INPUT 1

```
# header
x = 1

y = 2  # trailing
```

OUTPUT 1

```
code 2
comment 1
blank 1
```

INPUT 2

```
def f():
    return 1
```

OUTPUT 2

```
code 2
comment 0
blank 0
```

REFERENCE SOLUTION

```python
import sys

code = comment = blank = 0
for line in sys.stdin:
    stripped = line.strip()
    if not stripped:
        blank += 1
    elif stripped.startswith("#"):
        comment += 1
    else:
        code += 1
print(f"code {code}")
print(f"comment {comment}")
print(f"blank {blank}")
```

QUIZ

1. Which of these is an expression?

A. `x = 1`

B. `pass`

C. `x if c else y`

D. `import os`

2. What happens when this file runs?

```
x = 3
if x > 0:
print("positive")
```

A. Prints `positive`

B. Prints nothing

C. `IndentationError: expected an indented block`

D. `NameError`

3. What does this print?

```
list = [1, 2]
print(list((3, 4)))
```

A. `[3, 4]`

B. `[1, 2, 3, 4]`

C. `TypeError: 'list' object is not callable`

D. `[1, 2]`

4. Where must a docstring appear to be stored in `__doc__`?

A. Anywhere in the function body

B. As the first statement of the module, function or class

C. On the line immediately above the `def`

D. Inside a comment starting with `#:`

5. Which way of continuing a long statement onto the next line is preferred?

A. A backslash at the end of the line

B. A semicolon at the end of the line

C. Breaking inside open brackets — parentheses, square or curly

D. Indenting the continuation by eight spaces with no other marker

ANSWER KEY · ANATOMY OF A PYTHON PROGRAM

1. C The conditional expression produces a value and can appear inside a call or on the right of an assignment. Assignment, `pass` and `import` are statements: they do something and have no value.

2. C A colon opens a block, and the block must be indented. A line at the same level as the `if` is not its body, so the compiler reports the missing block before anything runs. `pass` fills a block that is meant to be empty.

3. C `list` is a built-in name, not a keyword, so the assignment shadows it with a list object. Calling `list(...)` then tries to call that list, which is not callable. Never name a variable after a built-in you still need.

4. B Only a string literal that is the *first* statement is a docstring. A string elsewhere in the body is an expression statement that evaluates to itself and is discarded; comments are ignored by the compiler entirely.

5. C A statement continues automatically until its brackets close, which is robust and what PEP 8 recommends. A backslash works but breaks if followed by a space; a semicolon separates statements rather than continuing one; indentation alone is a syntax error.

LESSON 4 · 14 MIN

Console input and output — the patterns every exercise uses

Every program in this track reads standard input and writes standard output, so the reading and printing idioms are worth fixing once, precisely. `input()` gives you one line as a string; `sys.stdin` gives you the whole stream; `print` writes objects separated by spaces with a newline at the end, and both of those defaults can be changed. The lesson ends with the five reading patterns that cover every input format the exercises use — copy them, and the I/O in an exercise stops being where the time goes.

`input()`

`input()` reads one line from standard input, strips the trailing newline, and returns it as a `str`. It always returns a string: the digits `42` come back as `"42"`, and arithmetic on that raises `TypeError` until you convert.

```
line = input()           # "3 4"
n = int(input())         # "7" → 7
x = float(input())       # "2.5" → 2.5
a, b = input().split()   # "3 4" → "3", "4" — still strings
```

`int(" 7\n")` tolerates surrounding whitespace, so `int(input())` is safe even if the line has stray spaces; `int("3.5")` and `int("")` raise `ValueError`. When the stream is exhausted, `input()` raises `EOFError` — the signal that there is no more input, and a way to loop "until the end":

```
lines = []
try:
    while True:
        lines.append(input())
except EOFError:
    pass
```

Never pass a prompt: `input("n = ")` writes `n =` to standard output first, and on the judge that text is part of your answer.

Several values on one line

`str.split()` with no argument splits on any run of whitespace and drops leading and trailing whitespace, which is exactly what "numbers separated by spaces" needs. `map(int, …)` converts each token; `list(…)` materialises the result.

```
nums = list(map(int, input().split()))        # "3 -1 4" → [3, -1, 4]
a, b, c = map(int, input().split())           # exactly three tokens
words = input().split()                       # ["to", "be", "or"]
first, *rest = input().split()                # one and the others
```

`split(",")` splits on an explicit separator and keeps empty fields (`"a,,b".split(",")` is `["a", "", "b"]`); `strip()` removes surrounding whitespace from a token. Module 5 covers parsing in depth.

`sys.stdin`

For bulk input, `sys.stdin` is a file object: iterate it for lines (each keeps its `\n` , so `.rstrip("\n")` or `.split()` them), `read()` for everything at once, `readline()` for one line (an empty string at end of stream, where `input()` would raise).

```
import sys

for line in sys.stdin:                    # until EOF
    print(line.rstrip("\n").upper())

data = sys.stdin.read().split()           # every whitespace-separated token in
```

`sys.stdin.read().split()` is the fastest way to read a large input: one system call, one split, then walk the tokens with an index or an iterator. It is also the most forgiving, because it does not care how the numbers are arranged into lines.

`print`

`print(*objects, sep=" ", end="\n", file=sys.stdout, flush=False)` converts each object with `str()` , joins them with `sep` , and appends `end` .

```
print("a", 1, 2.5)              # a 1 2.5
print("a", "b", sep="")         # ab
print("x", end="")              # no newline
print(*[1, 2, 3])               # 1 2 3   — unpack the list into arguments
print(*[1, 2, 3], sep=", ")     # 1, 2, 3
print("error", file=sys.stderr)
```

`print(nums)` for a list prints its `repr` — `[1, 2, 3]`, brackets and all — which is almost never the format an exercise asks for. To print elements separated by spaces use `print(*nums)`; to build the line yourself use `" ".join(map(str, nums))`, which is also how you print a list of strings as lines: `print("\n".join(lines))`.

Output is buffered when it goes to a pipe or file rather than a terminal. That is invisible for a judged program, which is flushed when it exits; it matters only when a program prints and then crashes (the crash's traceback on stderr may appear *before* buffered stdout on a terminal) or prints progress you want to see immediately — `print(..., flush=True)` or `python -u` forces it out.

Formatting numbers

The two formats every exercise needs are a fixed number of decimals and padding to a width, both spelled inside an f-string:

```
avg = 2 / 3
print(f"{avg:.2f}")          # 0.67
print(f"{42:5d}|")           # "   42|"
print(f"{'ab':<5}|")         # "ab   |"
print(f"{1234567:,}")        # 1,234,567
```

Module 5 lesson 3 has the whole format-spec mini-language; `.2f` is the one you must not print a float without, because `print(0.1 + 0.2)` gives `0.30000000000000004` and the expected output will not.

The five reading patterns

1. One number, then that many lines.

```
n = int(input())
rows = [input() for _ in range(n)]
```

2. A count and a line of values.

```
n = int(input())
nums = list(map(int, input().split()))      # n values (trust the count or che
```

3. Lines until end of input.

```
import sys
lines = [line.rstrip("\n") for line in sys.stdin]
```

4. Tokens regardless of line breaks.

```
import sys
tokens = sys.stdin.read().split()
it = iter(tokens)
n = int(next(it))
values = [int(next(it)) for _ in range(n)]
```

5. Records with a fixed shape per line.

```
n = int(input())
items = []
for _ in range(n):
    name, qty, price = input().split()
    items.append((name, int(qty), float(price)))
```

Pattern 3 is the one for "process every line"; pattern 4 is the one for large inputs and for inputs whose line structure does not matter. Reading everything first and computing second keeps the logic separate from the I/O, which is the shape the reference solutions in this track keep.

Pitfalls

Forgetting that `input()` returns a string. `input() + 1` is a `TypeError`; `input() * 2` doubles the text.

`print(list)` when the expected output is space-separated values.

Reading one line when the values are spread over several, or vice versa: read the input format twice before writing the first line.

A prompt in `input()`.

Trailing whitespace is forgiven by the judge; a missing newline between lines is not. `print` adds it; `sys.stdout.write` does not.

Key takeaways

`input()` returns one line as a string without its newline; `EOFError` marks the end.

`int(input())`, `list(map(int, input().split()))` and `for line in sys.stdin` cover almost every format; `sys.stdin.read().split()` for bulk tokens.

`print` joins with `sep` and ends with `end`; `print(*items)` spreads a list; `"\n".join(lines)` prints many lines at once.

Format floats with `:.2f`; never print a raw float.

Never prompt, never print a container's `repr` unless asked.

EXERCISE 1

A count, then the values

Read the most common input shape — a count on the first line, that many integers on the second — and print a summary. The mean is printed with exactly two decimals.

Input: `n` (at least 1), then a line of `n` integers. **Output:** four lines: `sum <s>`, `min <a>`, `max <b>`, `mean <m>` with `m` to two decimals.

```
5
3 -1 4 1 5
```

prints

```
sum 12
min -1
max 5
mean 2.40
```

INPUT 1

```
5
3 -1 4 1 5
```

OUTPUT 1

```
sum 12
min -1
max 5
mean 2.40
```

REFERENCE SOLUTION

```python
n = int(input())
nums = list(map(int, input().split()))
total = sum(nums)
print(f"sum {total}")
print(f"min {min(nums)}")
print(f"max {max(nums)}")
print(f"mean {total / n:.2f}")
```

EXERCISE 2

Numbered, with the words reversed

Read lines until the end of input with `sys.stdin`. Print each line numbered from 1 with its **words in reverse order** (words are whitespace-separated; extra spaces are not preserved), then a final line with the count.

Input: zero or more lines. **Output:** `<i>: <words reversed>` per line, then `lines: <n>`.

```
to be or
not to be
```

prints

```
1: or be to
2: be to not
lines: 2
```

INPUT 1

```
to be or
not to be
```

OUTPUT 1

```
1: or be to
2: be to not
lines: 2
```

REFERENCE SOLUTION

```python
import sys

count = 0
for line in sys.stdin:
    count += 1
    words = line.split()
    print(f"{count}: {' '.join(reversed(words))}")
print(f"lines: {count}")
```

QUIZ

1. The input line is `4`. What does this print?

```python
n = input()
print(n * 2)
```

A. `8`

B. `44`

C. `TypeError`

D. `4 4`

2. What does `print(*[1, 2, 3], sep="-")` print?

A. `[1, 2, 3]`

B. `1-2-3`

C. `1 2 3`

D. `-1-2-3-`

3. What is `"a,,b".split(",")` ?

A. `['a', 'b']`

B. `['a', '', 'b']`

C. `['a', ',', 'b']`

D. `['a,,b']`

4. What does `print(0.1 + 0.2)` print?

A. `0.3`

B. `0.30`

C. `0.30000000000000004`

D. `0.3000000000000000166`

5. How do `input()` and `sys.stdin.readline()` signal the end of input?

A. Both return `None`

B. `input()` raises `EOFError`; `readline()` returns an empty string `""`

C. Both raise `EOFError`

D. `input()` returns `""`; `readline()` raises `StopIteration`

ANSWER KEY · CONSOLE INPUT AND OUTPUT — THE PATTERNS EVERY EXERCISE USES

1. B `input()` returns the string `"4"`, and multiplying a string by an integer repeats it. Converting with `int(input())` first would give `8`. There is no error: `str * int` is defined.

2. B The star spreads the list into three separate arguments, and `sep` is placed between arguments. Without the star, `print([1, 2, 3])` prints the list's `repr` with brackets; the separator is never added before the first or after the last item.

3. B Splitting on an explicit separator keeps empty fields, so the two commas produce an empty string between them. Only the no-argument form `split()` collapses runs of whitespace and drops empties.

4. C Neither 0.1 nor 0.2 is exactly representable in binary floating point, and their sum's shortest round-tripping representation is `0.30000000000000004`. Format with `f"{x:.2f}"` when a fixed number of decimals is expected.

5. B `input()` raises `EOFError` when there is no more data, which is why a read-until-EOF loop wraps it in `try`. The file-object method returns an empty string — an actual empty line is `"\n"`, so the two are distinguishable.

LESSON 5 · 13 MIN

Errors and tracebacks — reading what the interpreter tells you

A Python error message is the most useful text the interpreter ever prints, and most beginners read only its last word. A traceback tells you *what* went wrong, *where*, and *how the program got there*; a syntax error tells you the line and the column. This lesson teaches how to read both from the bottom up, catalogues the eight exceptions you will meet in the first month, distinguishes them from the verdicts the judge gives, and sets out the two-line debugging habit — `print(repr(x))` and `f"{x=}"` — that solves most bugs before a debugger is needed.

Two kinds of error

A **syntax error** is found by the compiler before anything runs. The report names the file, the line, and points at the column:

```
  File "Main.py", line 3
    print("total:" total)
                   ^^^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

`IndentationError` (a block not indented, or dedented to a level that matches nothing) and `TabError` (tabs and spaces mixed) are syntax errors with better names. Since 3.10 the messages carry suggestions — the forgotten comma, an unclosed bracket named with the line it was opened on — and the caret range shows the exact tokens involved. When the reported line looks fine, look at the line above it: an unclosed `(` or `[` on line 2 is reported on line 3, where the parser finally gave up.

An **exception** is raised while the program runs. Execution stops at that point unless something catches it (Module 10), a traceback is printed to standard error, and the process exits with status 1.

Reading a traceback

```
Traceback (most recent call last):
  File "Main.py", line 12, in <module>
    main()
  File "Main.py", line 9, in main
    print(average(scores))
  File "Main.py", line 4, in average
    return sum(values) / len(values)
ZeroDivisionError: division by zero
```

Read it **bottom up**. The last line is the exception's type and message — the *what*. The frame just above it is *where* it happened: line 4, in `average`. The frames above that are the call chain that got there: `main` called `average` from line 9, and the module's top level called `main` from line 12. "Most recent call last" means the innermost frame is at the bottom. The bug is rarely on the line that raised; it is in whichever caller passed an empty list — here, `main` — and the chain is how you find it. From 3.11 a caret line under the source marks the exact sub-expression (`sum(values) / len(values)` with the division underlined), which settles which of several calls on one line failed.

The eight you will meet first

Exception	Typical cause
`NameError: name 'x' is not defined`	Typo, or used before assignment, or defined in another scope
`TypeError: can only concatenate str (not "int") to str`	Mixing types: `"a" + 1`, `len(5)`, calling a non-function, wrong argument count
`ValueError: invalid literal for int() with base 10: 'abc'`	Right type, bad value: `int("abc")`, `[1, 2].index(3)`, unpacking the wrong count
`IndexError: list index out of range`	`xs[len(xs)]`, an empty list's `xs[0]`
`KeyError: 'name'`	A missing dictionary key
`AttributeError: 'NoneType' object has no attribute 'append'`	A method on the wrong kind of object — very often on `None` returned by a function that did not `return`
`ZeroDivisionError: division by zero`	`x / 0`, `x % 0`, an average of nothing
`RecursionError: maximum recursion depth exceeded`	A recursion without a base case, or deeper than 1 000 frames

`AttributeError` on `NoneType` deserves a second look because its cause is one step back: `xs = xs.sort()` (`sort` sorts in place and returns `None`), a function with a missing `return`, or `re.match` finding nothing. `TypeError` versus `ValueError` is a distinction worth learning to name — the *type* was wrong versus the *value* was wrong — because your own code will raise them (Module 10).

What the judge reports

The judge's verdicts map onto these directly. A syntax error or an uncaught exception is a **runtime error**; the traceback is shown to you, so read its last line first. Output that does not match is a **wrong answer**, shown as expected versus actual — and the usual reasons are a formatting difference (`0.30000000000000004` for `0.30`, a `repr` of a list, a prompt string), not a wrong algorithm. Exceeding the time limit is **time limit exceeded**; a Python program that does this on small input is almost always in an infinite loop (a `while` whose condition never changes, or `input()` in a loop that never sees `EOFError` because a `try` swallowed it).

Debugging without a debugger

Most bugs are a value that is not what you believed. Print it — with `repr`, so that a string with a trailing newline or a number that is actually a string shows its true shape:

```
print(repr(line))        # '42\n'   — the newline you forgot to strip
print(f"{n=}, {total=}") # n=3, total=0   — name and value, since 3.8
```

Put such prints on standard error (`print(..., file=sys.stderr)`) while working on a judged program so that they do not become part of the answer, and remove them before submitting. `type(x)` answers "what is this actually?"; `dir(x)` lists what it can do. When a print is not enough, `breakpoint()` drops into `pdb`, the standard debugger — `n` steps to the next line, `p x` prints a variable, `c` continues, `q` quits — and Module 18 covers it properly. `assert cond, "message"` documents an assumption and fails loudly the moment it is false.

Warnings

A warning is a message that is not an error: `DeprecationWarning` for a feature scheduled for removal, `ResourceWarning` for a file never closed (shown under `python -X dev`), `SyntaxWarning` for `"is" with a literal`. They go to stderr and the program continues. `python -W error` promotes them to exceptions, which is how a test suite catches them early.

Pitfalls

- Reading only the last line. The frame above it is where, and the one above that is usually why.
- Fixing the reported syntax-error line when the real mistake is an unclosed bracket on the previous one.
- Leaving debug prints on stdout in a judged program.
- Catching an exception just to make the traceback go away. A `try` that hides a `ValueError` turns a clear failure into a silent wrong answer.
- Assuming `xs.sort()` returns the sorted list. It returns `None`; `sorted(xs)` returns a new list.

Key takeaways

Syntax errors are found before the program runs and name the line and column; exceptions happen at run time and stop the program with a traceback.

Read a traceback bottom up: the type and message, then the frame where it happened, then the callers.

`NameError`, `TypeError`, `ValueError`, `IndexError`, `KeyError`, `AttributeError`, `ZeroDivisionError`, `RecursionError` cover most early bugs; `AttributeError` on `NoneType` points at a missing `return` or an in-place method.

On the judge, an uncaught exception is a runtime error, a format difference is a wrong answer, an infinite loop is a time limit.

`print(repr(x))` and `f"{x=}"` on stderr find most bugs; `breakpoint()` is there for the rest.

EXERCISE 1

Name that exception

Learn the exception names by triggering them. Read `n` Python expressions, one per line, and evaluate each with `eval`. If evaluation raises, print the exception class's name (`type(e).__name__`); otherwise print the `repr` of the value.

Input: `n`, then `n` expressions. **Output:** one line per expression: `<expression> ⇒ <name or repr>`.

```
3
1/0
2 + 3
"a" * 3
```

prints

```
1/0 ⇒ ZeroDivisionError
2 + 3 ⇒ 5
"a" * 3 ⇒ 'aaa'
```

INPUT 1

```
5
1/0
int("x")
[1, 2][5]
{}["k"]
len(5)
```

OUTPUT 1

```
1/0 ⇒ ZeroDivisionError
int("x") ⇒ ValueError
[1, 2][5] ⇒ IndexError
{}["k"] ⇒ KeyError
len(5) ⇒ TypeError
```

INPUT 2

```
3
2 + 3
undefined_name
None.x
```

OUTPUT 2

```
2 + 3 ⇒ 5
undefined_name ⇒ NameError
None.x ⇒ AttributeError
```

REFERENCE SOLUTION

```python
n = int(input())
for _ in range(n):
    expr = input()
    try:
        value = eval(expr)
    except Exception as e:
        print(f"{expr} ⇒ {type(e).__name__}")
    else:
        print(f"{expr} ⇒ {value!r}")
```

EXERCISE 2

Read the traceback

A traceback arrives on standard input. Extract the three facts a reader needs: the exception type and message from the last line, and the innermost frame — the **last** line that starts with `File "` — with its line number and function name. The last line has the form `Type: message` (split on the first `:` only; the message may itself contain a colon) or just `Type` with no message.

Input: the traceback, one or more lines. **Output:** three lines: `type: <Type>`, `message: <message>` (or `message: (none)`), `where: line <n> in <function>`.

```
Traceback (most recent call last):
  File "Main.py", line 9, in main
    print(average(scores))
  File "Main.py", line 4, in average
    return sum(values) / len(values)
ZeroDivisionError: division by zero
```

prints

```
type: ZeroDivisionError
message: division by zero
where: line 4 in average
```

INPUT 1

```
Traceback (most recent call last):
  File "Main.py", line 12, in <module>
    main()
  File "Main.py", line 9, in main
    print(average(scores))
  File "Main.py", line 4, in average
    return sum(values) / len(values)
ZeroDivisionError: division by zero
```

OUTPUT 1

```
type: ZeroDivisionError
message: division by zero
where: line 4 in average
```

INPUT 2

```
Traceback (most recent call last):
  File "Main.py", line 2, in <module>
    print(data["name"])
KeyError: 'name'
```

OUTPUT 2

```
type: KeyError
message: 'name'
where: line 2 in <module>
```

REFERENCE SOLUTION

```python
import sys

lines = [line.rstrip("\n") for line in sys.stdin]
last = lines[-1]
if ": " in last:
    kind, message = last.split(": ", 1)
else:
    kind, message = last, "(none)"

frame = [line for line in lines if line.startswith('  File "')][-1]
# '  File "Main.py", line 4, in average'
_, rest = frame.split(", line ", 1)
number, function = rest.split(", in ", 1)

print(f"type: {kind}")
print(f"message: {message}")
print(f"where: line {number} in {function}")
```

QUIZ

1. In a traceback, which line tells you *where* the exception was raised?

A. The first `File` line, because the traceback is printed outermost-first

B. The `File` line directly above the final `Type: message` line — the innermost frame

C. The line that says `Traceback (most recent call last)`

D. The final line — it includes the line number

2. What does this do?

```
xs = [3, 1, 2]
xs = xs.sort()
print(xs[0])
```

A. Prints `1`

B. Prints `3`

C. `TypeError: 'NoneType' object is not subscriptable`

D. `AttributeError`

3. Which exception does `int("3.5")` raise?

A. `TypeError`

B. `ValueError`

C. `SyntaxError`

D. None — it returns `3`

4. The compiler reports `SyntaxError` on line 5, but line 5 looks correct. What is the most likely cause?

A. A bracket opened on an earlier line was never closed, and the parser gave up on line 5

B. Line 5 contains a keyword used as a variable name

C. Line 5 is indented with spaces instead of tabs

D. The file has Windows line endings

5. What does this raise?

```
d = {}
print(d["k"])
```

A. `IndexError`

B. `KeyError: 'k'`

C. `AttributeError`

D. Nothing — it prints `None`

ANSWER KEY · ERRORS AND TRACEBACKS — READING WHAT THE INTERPRETER TELLS YOU

1. B "Most recent call last" means the innermost frame is at the bottom, just above the exception line. The final line names the type and message but not the location; the first `File` line is the outermost caller.

2. C `list.sort()` sorts in place and returns `None`, so the assignment rebinds `xs` to `None` and indexing it fails. Either call `xs.sort()` without assigning, or use `xs = sorted(xs)`, which returns a new list.

3. B The argument has the right type (a string) but a value `int` cannot parse as an integer, which is `ValueError` by definition. `TypeError` is for the wrong kind of object, such as `int([1])`. `int(3.5)` on a float truncates to `3`, but that is a different call.

4. A An unclosed `(` or `[` makes the parser keep reading the next line as part of the same statement, and the error surfaces where it finally cannot continue. Modern messages usually say which line the bracket was opened on; look one line up regardless.

5. B Subscripting a dictionary with a missing key raises `KeyError`; `IndexError` is for sequences. To get `None` (or a default) without an exception, use `d.get("k")`.

CHECKPOINT · 22 MIN

Checkpoint — Python and the interpreter

This checkpoint covers the whole module: what CPython is and how it runs a file, the four ways to run Python and the `__name__` guard, the parts of a program (statements, expressions, blocks, comments, docstrings, names), the console I/O patterns every exercise uses, and how to read a syntax error and a traceback.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What does CPython do with your source before the first line runs, and what stops it?
- When is `__name__` equal to `"__main__"`, and why does a script check?
- What does `input()` return, what does it strip, and what does it raise at the end of input?
- How do you print the elements of a list separated by spaces, and why is `print(nums)` wrong?
- Which exit code means success, and what does an uncaught exception do to it?
- In a traceback, which line says *what* went wrong and which says *where*?
- What is the difference between `TypeError` and `ValueError`?

The three programs are a receipt built from records read line by line and printed with fixed decimals, a calculator that reports the exception name instead of crashing, and a word-count utility that reads standard input to the end.

EXERCISE 1

Receipt

Read `n` purchase records and print a receipt. Each record is a name, an integer quantity and a unit price. Print one line per item and a total, all prices with exactly two decimals.

Input: `n`, then `n` lines `name qty price`. **Output:** `<name> x<qty> @ <price> = <qty * price>` per item, then `total = <sum>`.

```
2
apple 3 0.5
bread 1 2.25
```

prints

```
apple x3 @ 0.50 = 1.50
bread x1 @ 2.25 = 2.25
total = 3.75
```

INPUT 1

```
2
apple 3 0.5
bread 1 2.25
```

OUTPUT 1

```
apple x3 @ 0.50 = 1.50
bread x1 @ 2.25 = 2.25
total = 3.75
```

REFERENCE SOLUTION

```python
n = int(input())
total = 0.0
for _ in range(n):
    name, qty, price = input().split()
    qty = int(qty)
    price = float(price)
    line_total = qty * price
    total += line_total
    print(f"{name} x{qty} @ {price:.2f} = {line_total:.2f}")
print(f"total = {total:.2f}")
```

EXERCISE 2

A calculator that never crashes

Read lines until the end of input, each meant to be `a op b` with integer operands and one of the operators `+ - * / // % **`. Print the result of each line — an integer for every operator except `/`, whose result is printed with two decimals. When a line cannot be computed, print the name of the exception instead of crashing: dividing by zero is `ZeroDivisionError`; a non-integer operand, an unknown operator, or a line without exactly three tokens is `ValueError`.

Input: zero or more lines. **Output:** one line per input line.

```
7 + 2
7 / 2
7 // 0
7 ^ 2
```

prints

```
9
3.50
ZeroDivisionError
ValueError
```

INPUT 1

```
7 + 2
7 / 2
7 // 0
7 ^ 2
```

OUTPUT 1

```
9
3.50
ZeroDivisionError
ValueError
```

REFERENCE SOLUTION

```python
import sys


def compute(line):
    a, op, b = line.split()
    a, b = int(a), int(b)
    if op == "+":
        return a + b
    if op == "-":
        return a - b
    if op == "*":
        return a * b
    if op == "/":
        return a / b
    if op == "//":
        return a // b
    if op == "%":
        return a % b
    if op == "**":
        return a ** b
    raise ValueError(f"unknown operator {op}")


for line in sys.stdin:
    try:
        result = compute(line)
    except (ValueError, ZeroDivisionError) as e:
        print(type(e).__name__)
    else:
        if isinstance(result, float):
            print(f"{result:.2f}")
        else:
            print(result)
```

EXERCISE 3

wc

Write the classic word-count utility. Read **all** of standard input and print the number of lines, words and characters. Lines are what `str.splitlines()` yields, words are whitespace-separated tokens, and characters are every character including newlines.

Input: any text, possibly empty. **Output:** three lines: `lines <n>`, `words <n>`, `chars <n>`.

```
hello world
foo
```

prints

```
lines 2
words 3
chars 16
```

INPUT 1

```
hello world
foo
```

OUTPUT 1

```
lines 2
words 3
chars 16
```

REFERENCE SOLUTION

```python
import sys

data = sys.stdin.read()
print(f"lines {len(data.splitlines())}")
print(f"words {len(data.split())}")
print(f"chars {len(data)}")
```

CHECKPOINT QUESTIONS

1. Which statement about how CPython runs a program is correct?

A. It compiles the source to bytecode and then interprets that bytecode

B. It interprets the source text directly, one character at a time

C. It compiles the source to machine code once and caches the binary

D. It runs the program twice: once to check types and once for real

2. What does `print(type(3 / 1).__name__)` print?

A. `int`

B. `float`

C. `Fraction`

D. `decimal`

3. What is printed when this script runs?

```
def f():
    return 5

f()
print(f())
```

A. `5` twice

B. `5` once

C. Nothing

D. `None`

4. What does `f"{7 / 2:.1f}"` evaluate to?

A. `'3.5'`

B. `'3'`

C. `'3.50'`

D. `'4'`

5. What does `print("a", "b", sep="")` print?

A. `a b`

B. `ab`

C. `a,b`

D. `a\nb`

6. In `for line in sys.stdin:`, what is true of each `line`?

A. It has the trailing newline already removed

B. It keeps its trailing newline, so `rstrip("\n")` or `split()` is needed

C. It is a list of the line's words

D. It is `None` on the last iteration

7. The input line is `1 2 3`. What happens?

```
a, b = input().split()
```

A. `a` is `"1"` and `b` is `"2 3"`

B. `a` is `"1"` and `b` is `"2"`; the `3` is dropped

C. `ValueError: too many values to unpack (expected 2)`

D. `TypeError`

8. What does a function return when it has no `return` statement?

A. `0`

B. `False`

C. `None`

D. An empty string

9. The input line is `hi ` (with one trailing space). What does `print(len(input()))` print?

A. `2`

B. `3`

C. `4`

D. `1`

10. What error does mixing a tab and spaces in the indentation of one block produce?

A. `SyntaxError: unexpected indent`

B. `TabError: inconsistent use of tabs and spaces in indentation`

C. No error — tabs are expanded to 8 spaces

D. `IndentationError: unindent does not match`

11. Which interpreter flag removes `assert` statements from the program?

A. `-i`

B. `-m`

C. `-O`

D. `-u`

12. A program raises an exception that nothing catches. What exit status does the process end with?

A. `0`

B. `1`

C. `-1`

D. The exception's line number

ANSWER KEY · CHECKPOINT — PYTHON AND THE INTERPRETER

1. A Bytecode compilation followed by interpretation in the evaluation loop is the CPython model; the `.pyc` files in `__pycache__` are cached bytecode, not machine code. There is no separate type-checking pass.

2. B In Python 3 the `/` operator always produces a `float`, even when the division is exact. Floor division `//` on two integers produces an `int`.

3. B The bare call `f()` evaluates to 5 and discards it — only the REPL would echo it. The `print` call prints the second result once.

4. A `7 / 2` is `3.5`, and `.1f` formats with exactly one decimal place. `.2f` would give `3.50`; `//` would be needed to get `3`.

5. B `sep` is the string placed between arguments; the empty string joins them directly. The default separator is one space.

6. B Iterating a file object yields raw lines including the `\n`, unlike `input()`, which strips it. The loop simply ends at EOF; there is no `None` sentinel.

7. C Unpacking requires the exact number of values unless a starred target absorbs the rest (`a, *b = ...`). Three tokens into two names is a `ValueError`.

8. C Falling off the end of a function returns `None`. This is the source of many `AttributeError: 'NoneType' object has no attribute ...` messages: a helper that forgot to `return` its result.

9. B `input()` strips only the trailing newline, not spaces, so the string is `"hi "` with length 3. Use `.strip()` when surrounding whitespace should not count.

10. B Python 3 refuses inconsistent tab/space mixing outright with `TabError`, a subclass of `IndentationError`. The other messages describe different indentation faults: a line indented for no reason, or a dedent to a level that matches no open block.

11. C `-O` (optimise) sets `__debug__` to `False` and strips asserts, which is why an `assert` must never guard something the program needs. `-i` opens the REPL afterwards, `-m` runs a module, `-u` unbuffers output.

12. B An uncaught exception prints the traceback to stderr and exits with status 1, which shells, CI and the judge read as failure. Only reaching the end of the file, or `sys.exit(0)`, gives 0.
