Module 12 of 20

Modules, packages and imports

What import does and the sys.modules cache, from-imports copying bindings, packages with __init__.py and relative imports, the standard-library map, virtual environments with pip and pyproject.toml, and command-line scripts with argparse, streams and exit codes.

A module is a file and a package is a directory, and everything else about Python's import system follows from how those two things are found, run once and cached. This module makes that mechanism explicit — the `sys.modules` cache, the difference between `import m` and `from m import x`, why top-level code runs at import time, why circular imports fail the way they do — and then covers the practical layer around it: package layout, relative imports and `python -m`, the standard library as a map to consult before writing anything, virtual environments and `pyproject.toml`, and the conventions that make a script a proper command-line tool.

Modules and imports covers the forms, the five steps of an import, binding-copy semantics, `__name__`, circular imports and style. Packages covers `__init__.py`, `__all__`, relative imports, `python -m`, `__main__.py`, the `src/` layout and `importlib.resources`. The standard-library map indexes the modules by job. Virtual environments and packaging covers `venv`, `pip`, version specifiers, `pyproject.toml`, editable installs and the tools. Scripts and the CLI covers `argparse`, the three streams, exit codes, environment variables and the testable `main(argv)` shape.

The exercises are whole programs: an import-order simulator with a cache, a real dynamically built module that shows binding copies, a package created on disk to show submodule attributes, a relative-import resolver, a census of imports against `sys.stdlib_module_names`, a module explorer, a requirements parser with version checks, a `tomllib` reader for `pyproject.toml`, an `argparse` driver fed from lines, and a `main(argv)` with exit codes. The checkpoint adds an import graph with cycle detection, a requirements checker, and a sub-command tool.

CONTENTS

1	Modules and imports — files, namespaces and sys.modules	14 min
2	Packages — directories, __init__.py, relative imports and project layout	14 min
3	The standard library map — where to look before you write it	14 min

4	Virtual environments and packaging — venv, pip and pyproject.toml	13 min
5	Scripts and the command line — argv, argparse, exit codes and streams	14 min
✓	Checkpoint — Modules, packages and imports	22 min

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

LESSON 1 · 14 MIN

Modules and imports — files, namespaces and sys.modules

A module is a file; importing it runs the file once and gives you an object whose attributes are the file's top-level names. That sentence covers most of what people find mysterious about imports — why a module's top-level code runs on first import and never again, why two `import`s of the same module give the same object, why `from x import y` copies a *binding* rather than linking to a variable, and why circular imports fail the way they do. This lesson covers the import forms, what `import` does step by step, the module cache, `__name__`, the import path, and the conventions for ordering and style.

The forms

```
import math                          # binds the name math to the module obje
import numpy as np                   # an alias
from math import sqrt, pi            # binds sqrt and pi directly
from math import sqrt as root        # aliased
from os.path import join             # a name from a submodule
from math import *                   # every public name — avoid outside the I
```

`import math` gives one name and you write `math.sqrt`; `from math import sqrt` gives the function itself. The first keeps the origin visible at every use and cannot collide; the second is shorter. Convention: `import module` for modules you use many names from or whose names are generic (`os`, `json`, `re`), `from module import name` for a few specific names, and the alias forms only for the community-standard ones (`np`, `pd`) or to dodge a clash.

What import does

`import x` performs, in order:

1. If `x` is already in `sys.modules` (the cache), bind the name to that object and stop.
2. Otherwise find the file: the current script's directory, then each entry of `sys.path` (`PYTHONPATH`, the standard library, `site-packages`).

Create an empty module object and put it in `sys.modules` *before* running the code (this is what makes circular imports partially work).

Execute the file top to bottom in the module's own namespace: every `def`, `class` and assignment binds a name on the module.

Bind `x` in the importing namespace.

So top-level code runs **once per process**, at the first import, wherever that happens. A module that prints or opens a file at top level does so at import time — one reason work belongs in functions and under `if __name__ == "__main__":`. Importing the same module again anywhere is a dictionary lookup.

---

from-import copies a binding

```
# config.py
DEBUG = False
def enable():
    global DEBUG
    DEBUG = True

# main.py
from config import DEBUG, enable
enable()
print(DEBUG)          # False — main's DEBUG still points at the old object

import config
config.enable()
print(config.DEBUG)   # True — reading through the module sees the rebinding
```

`from config import DEBUG` binds `main.DEBUG` to whatever object `config.DEBUG` was at import time. Rebinding `config.DEBUG` later does not touch `main.DEBUG`. Mutable objects are shared (a list imported by name reflects appends), rebinding is not. Access module-level state that changes through the module.

---

name and the main guard

Each module has `__name__`: its import name, or `"__main__"` for the script being run. The guard `if __name__ == "__main__":` (Module 1) is what lets one file be imported for its

functions and run for its behaviour. `python -m package.module` runs a module as `__main__` by import name, which is what makes relative imports (next lesson) work in a script.

The module object

```
import math
type(math)                  # <class 'module'>
math.__name__               # 'math'
math.__file__               # the path (built-in modules like sys have none)
dir(math)                   # every attribute
math.__dict__               # the namespace as a dict
vars(math) is math.__dict__  # True
```

A module is a plain object: attributes can be read with `getattr`, added by assignment (a bad habit outside tests), and the namespace inspected. `importlib.import_module("pkg.mod")` imports by a string name — for plugins and dynamic loading; `importlib.reload(mod)` re-executes a module in place, which is a REPL convenience and not something a program should rely on.

Circular imports

`a.py` imports `b`, and `b.py` imports `a`. Importing `a` puts a half-empty `a` in `sys.modules`, starts running `a`, hits `import b`, runs `b` — which finds `a` in the cache and gets the half-initialised object. If `b` only uses `a`'s names inside functions called later, it works; if `b` does `from a import something` at top level before `a` has defined it, `ImportError: cannot import name`. The fixes, in order of preference: move the shared thing into a third module both import; import inside the function that needs it; import the module (`import a`) and access `a.something` lazily. A cycle is usually a sign that two modules are one module or that a dependency points the wrong way.

Style

> Imports at the top of the file, one per line, in three groups separated by blank lines: standard library, third-party, your own — each alphabetical. `isort` and `ruff` do this automatically (Module 15).

No `import *` in modules: it hides where names come from and can shadow silently.

No import inside a function unless breaking a cycle or deferring an expensive optional dependency.

Names of modules are `lowercase`, short, without hyphens (a hyphen cannot be imported).

Never name your file after a standard module (`random.py`, `test.py`, `json.py`): the script's directory comes first on the path, and `import random` will find *yours*.

Pitfalls

Top-level side effects that run at import.

Expecting `from m import x` to track later rebindings of `m.x`.

A file named like a standard module shadowing it.

Circular top-level `from` imports.

`import *` hiding a name collision.

Relying on `reload` in a running program.

Key takeaways

A module is a file; `import` finds it, runs it once, caches it in `sys.modules` and binds a name; later imports are lookups.

`import m` keeps names qualified; `from m import x` copies the binding at import time — rebinding through the module does not reach it.

`__name__` is the import name or `"__main__"`; the guard separates library use from script use.

Circular imports fail when a `from` import needs a name not yet defined; restructure, or import lazily.

Imports at the top in three sorted groups; never `import *`; never shadow a standard module's name.

EXERCISE 1

Import order, simulated

Simulate the import system. Lines `module <name> imports <a> <b> …` declare what each module imports at its top level (a module may import nothing). Lines `run <name>` import that module: a module not yet in the cache has its body started — print its name — and its imports are processed in order (each recursively) before the body finishes; a module already in the cache is not run again. For each `run`, print the names of the modules whose bodies started, in order, or `(cached)` if none.

Input: lines. **Output:** one line per `run`.

```
module a imports b c
module b imports c
module c imports
run a
run c
run b
```

prints

```
a b c
(cached)
(cached)
```

INPUT 1

```
module a imports b c
module b imports c
module c imports
run a
run c
run b
```

OUTPUT 1

```
a b c
(cached)
(cached)
```

REFERENCE SOLUTION

```
import sys

imports = {}
cache = set()


def do_import(name, started):
    if name in cache:
        return
    cache.add(name)
    started.append(name)
    for dep in imports.get(name, []):
        do_import(dep, started)


for line in sys.stdin:
    parts = line.split()
    if not parts:
        continue
    if parts[0] == "module":
        imports[parts[1]] = parts[3:]
    elif parts[0] == "run":
        started = []
        do_import(parts[1], started)
        print(" ".join(started) if started else "(cached)")
```

EXERCISE 2

A binding is a copy

Build a real module at run time: create `types.ModuleType("config")`, execute the source `DEBUG = False` plus an `enable()` / `disable()` pair that rebind the module-level `DEBUG` with `global`, register it in `sys.modules`, then do `from config import DEBUG, enable, disable` **and** `import config`. For each command line (`enable` or `disable`) call the function and print the imported name `DEBUG` next to `config.DEBUG`, showing that the from-import copied the binding.

Input: command lines. **Output:** `imported=<bool> module=<bool>` per command.

```
enable
disable
```

prints

```
imported=False module=True
imported=False module=False
```

INPUT 1

```
enable
disable
```

OUTPUT 1

```
imported=False module=True
imported=False module=False
```

REFERENCE SOLUTION

```python
import sys
import types

SOURCE = """
DEBUG = False

def enable():
    global DEBUG
    DEBUG = True

def disable():
    global DEBUG
    DEBUG = False
"""

module = types.ModuleType("config")
exec(SOURCE, module.__dict__)
sys.modules["config"] = module

from config import DEBUG, enable, disable  # noqa: E402
import config  # noqa: E402

for line in sys.stdin:
    cmd = line.strip()
    if cmd == "enable":
        enable()
    elif cmd == "disable":
        disable()
    print(f"imported={DEBUG} module={config.DEBUG}")
```

QUIZ

1. How many times does a module's top-level code run when it is imported from five places in one program?

A. Five

B. Once — later imports find it in `sys.modules`

C. Zero

D. Depends on the import form

2. After `from m import x`, `m.x = 5` is executed elsewhere. What is your `x`?

A. `5`

B. Still the object it was bound to at import time

C. `None`

D. `NameError`

3. Why does naming your file `random.py` break `import random`?

A. It does not

B. The script's directory is first on `sys.path`, so your file shadows the standard module

C. Python forbids it

D. Only on Windows

4. What is `__name__` inside a module that was imported (not run)?

A. `"__main__"`

B. The module's import name

C. The file path

D. `None`

5. Which is the preferred fix for a circular import?

A. `import *`

B. Move the shared names into a third module both import, or import lazily inside the function

C. `reload`

D. Rename the modules

ANSWER KEY · MODULES AND IMPORTS — FILES, NAMESPACES AND SYS.MODULES

1. B The cache is checked first; only the first import executes the file.

2. B A from-import copies a binding; rebinding the module attribute does not reach it. Access changing state through the module.

3. B Never name a file after a standard module you or your dependencies import.

4. B Only the script being run gets `"__main__"`, which is what the main guard tests.

5. B A cycle usually means a dependency points the wrong way; restructure first, defer the import second.

LESSON 2 · 14 MIN

Packages — directories, __init__.py, relative imports and project layout

A package is a directory of modules with an `__init__.py`, importable by its directory name with dots for the levels: `import app.models.user`. Packages are how a project grows past one file without the names colliding, and their rules are few: what `__init__.py` does, how `from . import x` resolves, why `python -m` matters for scripts inside a package, and how to lay a project out so that tests, entry points and the importable code all find each other. This lesson covers those rules, `__all__`, `__main__.py`, namespace packages, and the standard `src/` layout.

A package

```
shop/
    __init__.py
    models.py
    pricing.py
    cli.py
    utils/
        __init__.py
        text.py
```

```
import shop.models                      # binds shop; shop.models is an attri
from shop.models import Product         # a name from a submodule
from shop.utils import text             # a submodule from a subpackage
import shop.utils.text as textutils
```

Importing `shop.models` first imports `shop` (running `shop/__init__.py`), then `shop/models.py`, and binds `models` as an attribute on the `shop` module. `import shop` alone does **not** import its submodules — `shop.models` is an `AttributeError` until something imports it — unless `__init__.py` does so.

init.py

The file runs when the package is imported. It may be empty (most are), or it may define the package's public interface by importing from submodules:

```
# shop/__init__.py
from .models import Product, Order
from .pricing import price_of

__all__ = ["Product", "Order", "price_of"]
__version__ = "1.2.0"
```

Now `from shop import Product` works without callers knowing which file holds it, and the internal layout can change without breaking them. Keep `__init__.py` to re-exports and metadata: logic there runs on every import of any submodule and is hard to test. `__all__` lists the names `from shop import *` exports and documents the public surface; tools read it too.

Relative imports

Inside a package, modules refer to their siblings with leading dots:

```
# shop/pricing.py
from .models import Product             # sibling module
from .utils.text import slug            # a subpackage's module
from . import models                    # the sibling module object
from ..other import thing               # the parent package's sibling (two do
```

Relative imports are resolved against the module's `__package__`, so they work wherever the package is installed or placed on the path, and they say "this is internal" at a glance. Absolute imports (`from shop.models import Product`) are equally correct and preferred by some style guides for their clarity; pick one per project. Two limits: a relative import cannot go above the top-level package, and it does not work in a file run *as a script by path* — which is the next point.

---

python -m and the script problem

Running `python shop/cli.py` sets `__name__` to `"__main__"` and `__package__` to nothing, puts `shop/` (not its parent) on the path, and every `from .models import …` fails with

`ImportError: attempted relative import with no known parent package`. The fix is to run modules by import name from the project root:

```
$ python -m shop.cli
```

`-m` imports `shop`, then runs `shop.cli` as `__main__` *with its package set*, so relative and absolute imports both work. A package can also define `shop/__main__.py`, which is what `python -m shop` runs — the conventional entry point for a package that is a command.

Project layout

```
myproject/
    pyproject.toml
    README.md
    src/
        shop/
            __init__.py
            models.py
            …
    tests/
        test_models.py
        test_pricing.py
```

The `src/` layout keeps the importable package out of the project root, so that tests import the *installed* package (`pip install -e .`, next lesson) rather than the working directory by accident — a class of "works on my machine" bugs disappears. Tests live outside the package, import it absolutely (`from shop.models import Product`), and are run with `python -m pytest` or `python -m unittest` from the root. The flat alternative (package directory beside `tests/` at the root) works for small projects; the `src/` layout scales.

Namespace packages

A directory *without* `__init__.py` is still importable (since 3.3) as a *namespace package* — several such directories with the same name on the path merge into one package. This is for large organisations splitting one namespace across distributions (`company.tool_a`, `company.tool_b`). For your own code, always add `__init__.py`: it marks the directory as

deliberate, makes the package findable by more tools, and avoids the merging behaviour when a stray same-named directory exists.

Finding out what is where

```
import shop
shop.__path__                # the directories that make up the package
shop.__file__                # its __init__.py
shop.models.__name__         # 'shop.models'
import importlib.resources   # read data files shipped inside a package
```

`importlib.resources.files("shop") / "data.json"` is the right way to open a file that lives *inside* a package, wherever it was installed — never a path relative to the current working directory, which is wherever the user happened to run the program.

Pitfalls

- `import pkg` and expecting `pkg.sub` to exist without importing it.
- Logic in `__init__.py` that runs on every import.
- Running a module inside a package by path; use `-m`.
- Relative imports in the top-level script.
- Data files opened relative to the working directory.
- Missing `__init__.py` making a package a namespace package by accident.

Key takeaways

- A package is a directory with `__init__.py`; `import a.b.c` imports each level and binds submodules as attributes only once imported.
- `__init__.py` re-exports the public API and sets `__all__`; keep logic out of it.
- `from .x import y` is a relative import inside a package; run package modules with `python -m pkg.mod`, never by path; `__main__.py` is `python -m pkg`.
- Use the `src/` layout with tests outside the package importing it absolutely.
- Package data is read with `importlib.resources`, not a working-directory path.

EXERCISE 1

A package on disk

Create a real package next to this script at run time: a directory `pkg_demo` with an empty `__init__.py` and one module file per name on the first input line, each containing `NAME = "<name>"`. Then process commands: `import pkg_demo` (print `imported pkg_demo`), `import pkg_demo.<x>` (print `imported pkg_demo.<x>`), and `has <x>` — print whether `pkg_demo` currently has the attribute `<x>`, which is only true once that submodule has been imported. Remove the directory at the end.

Input: a line of module names, then commands. **Output:** one line per command.

```
models pricing
import pkg_demo
has models
import pkg_demo.models
has models
has pricing
```

prints

```
imported pkg_demo
False
imported pkg_demo.models
True
False
```

INPUT 1

```
models pricing
import pkg_demo
has models
import pkg_demo.models
has models
has pricing
```

OUTPUT 1

```
imported pkg_demo
False
imported pkg_demo.models
True
False
```

REFERENCE SOLUTION

```
import importlib
import os
import shutil
import sys

base = os.path.dirname(os.path.abspath(__file__))
pkg_dir = os.path.join(base, "pkg_demo")
names = input().split()
os.makedirs(pkg_dir, exist_ok=True)
with open(os.path.join(pkg_dir, "__init__.py"), "w", encoding="utf-8") as f:
    f.write("")
for name in names:
    with open(os.path.join(pkg_dir, f"{name}.py"), "w", encoding="utf-8") as f:
        f.write(f'NAME = "{name}"\n')
if base not in sys.path:
    sys.path.insert(0, base)

try:
    for line in sys.stdin:
        cmd, *args = line.split()
        if cmd == "import":
            importlib.import_module(args[0])
            print(f"imported {args[0]}")
        elif cmd == "has":
            print(hasattr(sys.modules["pkg_demo"], args[0]))
finally:
    shutil.rmtree(pkg_dir, ignore_errors=True)
```

EXERCISE 2

Resolve a relative import

Given the dotted name of the **module** doing the import and a relative import target, compute the absolute module name. One leading dot means the module's own package; each additional dot goes up one package. The part after the dots (possibly empty) is appended. Going above the top-level package prints `error: beyond top-level package`; a module with no package at all (no dot in its name) cannot use relative imports — print the same error.

Input: lines `<module> <target>`. **Output:** one line per input line.

```
shop.pricing .models
shop.utils.text ..models
shop.cli ...x
```

prints

```
shop.models
shop.models
error: beyond top-level package
```

INPUT 1

```
shop.pricing .models
shop.utils.text ..models
shop.cli ...x
```

OUTPUT 1

```
shop.models
shop.models
error: beyond top-level package
```

REFERENCE SOLUTION

```python
import sys


def resolve(module, target):
    dots = len(target) - len(target.lstrip("."))
    rest = target[dots:]
    package_parts = module.split(".")[:-1]
    if not package_parts:
        return "error: beyond top-level package"
    up = dots - 1
    if up > len(package_parts) - 1:
        return "error: beyond top-level package"
    base = package_parts[: len(package_parts) - up]
    return ".".join(base + ([rest] if rest else []))


for line in sys.stdin:
    module, target = line.split()
    print(resolve(module, target))
```

QUIZ

1. After `import shop`, what is `shop.models` if nothing has imported the submodule?

A. The submodule, loaded lazily

B. `AttributeError` — submodules are bound only when imported

C. `None`

D. An empty module

2. What does `python -m shop.cli` do that `python shop/cli.py` does not?

A. Runs faster

B. Imports `shop` first and runs `cli` as `__main__` with its package set, so relative imports work

C. Skips `__init__.py`

D. Nothing

3. What is `__all__` for?

A. Listing all files in the package

B. Naming the public names `import *` exports and documenting the API

C. Preventing imports

D. Speeding up imports

4. Why prefer the `src/` layout?

A. It is required by pip

B. Tests import the installed package rather than the working directory by accident

C. It makes imports faster

D. It avoids `__init__.py`

5. How should a package read a data file it ships?

A. `open("data.json")`

B. `importlib.resources.files("pkg") / "data.json"`

C. `os.getcwd() + "/data.json"`

D. `sys.argv[0]`

ANSWER KEY · PACKAGES — DIRECTORIES, __INIT__.PY, RELATIVE IMPORTS AND PROJECT LAYOUT

1. B Importing a package runs `__init__.py` only; `from .models import …` there would bind it.

2. B Running by path puts the file's directory on the path and leaves `__package__` empty.

3. B Tools and readers use it as the package's declared surface.

4. B With the package under `src/`, `import shop` only works after `pip install -e .`, exactly as it will for users.

5. B The working directory is wherever the user ran the program; package resources are located relative to the package.

LESSON 3 · 14 MIN

The standard library map — where to look before you write it

"Batteries included" is Python's oldest slogan, and the batteries are the reason a Python program that parses a date, reads a CSV, hashes a file and serves it over HTTP is a hundred lines rather than a thousand. The skill is knowing the map: which module holds the thing you need, so that you look there before writing it yourself. This lesson is that map — the modules grouped by job, with the one or two functions each is used for — plus how to read the documentation and how to explore a module from the REPL. Later modules of this track go deep on the areas that matter most; this one is the index.

Text

Module	For
`string`	`ascii_letters`, `digits`, `punctuation`, `Template`
`re`	regular expressions (Module 5)
`textwrap`	`wrap`, `fill`, `dedent`, `shorten`, `indent`
`difflib`	`SequenceMatcher`, `unified_diff`, `get_close_matches`
`unicodedata`	`normalize`, `name`, `category`
`pprint`	`pprint`, `pformat` for nested data

Data structures and functional tools

Module	For
`collections`	`Counter`, `defaultdict`, `deque`, `namedtuple`, `OrderedDict`, `ChainMap`
`itertools`	`chain`, `islice`, `groupby`, `product`, `combinations`, `accumulate` (Module 11)
`functools`	`cache`, `partial`, `reduce`, `wraps`, `singledispatch` (Module 11)
`operator`	`itemgetter`, `attrgetter`, the operators as functions
`heapq`, `bisect`	priority queues, binary search in sorted lists (Module 7)
`enum`	`Enum`, `IntEnum`, `Flag`, `auto` (Module 13)
`dataclasses`	`dataclass`, `field`, `asdict` (Module 8)
`copy`	`copy`, `deepcopy`
`array`	compact homogeneous numeric arrays

Numbers

Module	For
`math`	`sqrt`, `gcd`, `lcm`, `isqrt`, `comb`, `factorial`, `log`, `floor`, `ceil`, `isclose`, `inf`, `pi`
`statistics`	`mean`, `median`, `mode`, `stdev`, `quantiles`
`fractions`	`Fraction` — exact rationals
`decimal`	`Decimal` — decimal floating point for money
`random`	`Random(seed)`, `randint`, `choice`, `shuffle`, `sample`
`secrets`	cryptographically strong tokens and choices
`numbers`	the numeric ABCs (`Number`, `Integral`)

Dates and times

Module	For
`datetime`	`date`, `time`, `datetime`, `timedelta`, `strptime` / `strftime` (Module 13)
`zoneinfo`	IANA time zones: `ZoneInfo("Asia/Kolkata")`
`time`	`perf_counter`, `monotonic`, `sleep`, `time`
`calendar`	month tables, leap years, weekday names

Files, paths and the OS

Module	For
`pathlib`	`Path` — the object-oriented file system API (Module 14)
`os`	`environ`, `getcwd`, `listdir`, process and OS calls
`os.path`	the older string-path functions
`shutil`	`copy`, `move`, `rmtree`, `which`, disk usage
`glob`, `fnmatch`	wildcard file matching
`tempfile`	`TemporaryDirectory`, `NamedTemporaryFile`
`io`	`StringIO`, `BytesIO` — in-memory files
`sys`	`argv`, `stdin` / `stdout` / `stderr`, `exit`, `path`, `version_info`, `setrecursionlimit`
`subprocess`	run other programs: `run(["ls", "-l"], capture_output=True, text=True)`
`argparse`	command-line parsing (lesson 5)
`logging`	structured logging (Module 15)

Data formats and persistence

Module	For
`json`	`loads` / `dumps`, `load` / `dump` (Modules 7, 14)
`csv`	`reader`, `writer`, `DictReader`, `DictWriter` (Module 14)
`tomllib`	read TOML (3.11); writing needs a third-party package
`configparser`	INI files
`pickle`	Python-object serialisation — never load untrusted data
`sqlite3`	an embedded SQL database (Module 14)
`struct`	pack/unpack binary records
`base64`, `hashlib`, `hmac`, `zlib`, `gzip`, `zipfile`, `tarfile`	encodings, digests, compression, archives
`xml.etree.ElementTree`, `html`	XML parsing, HTML escaping

Concurrency and networking

Module	For
`threading`, `queue`	threads, locks, thread-safe queues (Module 17)
`multiprocessing`, `concurrent.futures`	processes and pools (Module 17)
`asyncio`	the async event loop (Module 17)
`socket`, `ssl`, `selectors`	low-level networking
`http.client`, `http.server`, `urllib.request`	HTTP without third-party packages
`email`, `smtplib`	building and sending mail

Language, testing and tooling

Module	For
`typing`, `abc`, `collections.abc`	hints, ABCs, protocols (Modules 9, 15)
`contextlib`	context-manager helpers (Module 10)
`unittest`, `unittest.mock`, `doctest`	testing (Module 18)
`traceback`, `warnings`, `pdb`, `inspect`	debugging and introspection
`timeit`, `cProfile`, `tracemalloc`, `dis`	measurement (Module 19)
`importlib`	import by name, package resources
`ast`	parse Python source; `literal_eval`
`venv`, `ensurepip`	environments (lesson 4)

Reading the documentation

Every module's page at `docs.python.org/3/library/` follows one shape: a summary, the functions and classes with their signatures and version notes ("Changed in version 3.10"), then examples. Read the signature, then the version note if you support older interpreters, then the

example. In the REPL, `help(module)` prints the docstrings, `dir(module)` lists the names, and `help(module.function)` shows the signature — enough for most questions without leaving the terminal. For a function whose behaviour you are unsure of, the fastest check is to call it on a small value and look.

The habit

Before writing a helper, ask: is this a text job (`textwrap` , `difflib`), a collection job (`collections` , `itertools`), a number job (`math` , `statistics`), a file job (`pathlib` , `shutil`), a format job (`json` , `csv`)? Ten minutes with the map saves an hour and a bug: the library's version has been tested on the edge cases you have not thought of yet.

Pitfalls

- Writing a mean, a deque, a permutations generator or a CSV parser by hand.
- `pickle.load` on data from outside your program.
- `time.time` where `perf_counter` (timing) or `monotonic` (timeouts) was meant.
- `os.path` string juggling where `pathlib` reads better.
- Reaching for a third-party package before checking the standard library.
- Guessing at behaviour instead of a two-line REPL experiment.

Key takeaways

- The standard library covers text, collections, numbers, dates, files, formats, concurrency, networking, testing and tooling; know the map before writing helpers.
- `collections` , `itertools` , `functools` , `pathlib` , `json` , `csv` , `datetime` , `re` , `math` , `logging` , `unittest` are the modules every program touches.
- `help()` , `dir()` and a small experiment in the REPL answer most questions; the docs show signature, version notes, examples.
- `pickle` is for trusted data only; `secrets` for anything security-related; `perf_counter` for timing.
- Prefer the library's tested implementation to a hand-written one.

EXERCISE 1

Import census

Read Python import statements — `import a.b as c`, `import x, y`, `from p.q import r` — and classify each imported **top-level** module as `stdlib` (its first component is in `sys.stdlib_module_names`) or `other`. Print the distinct top-level names of each group, sorted, then how many statements there were.

Input: import statements, one per line. **Output:** `stdlib: <names>` (or `stdlib: none`), `other: <names>` (or `other: none`), `statements <n>`.

```
import os.path as osp
from collections import Counter
import numpy as np, json
from shop.models import Product
```

prints

```
stdlib: collections json os
other: numpy shop
statements 4
```

INPUT 1

```
import os.path as osp
from collections import Counter
import numpy as np, json
from shop.models import Product
```

OUTPUT 1

```
stdlib: collections json os
other: numpy shop
statements 4
```

REFERENCE SOLUTION

```python
import sys


def top_levels(statement):
    if statement.startswith("from "):
        module = statement.split()[1]
        return [module.split(".")[0]]
    body = statement[len("import "):]
    names = []
    for part in body.split(","):
        dotted = part.strip().split()[0]      # drop "as alias"
        names.append(dotted.split(".")[0])
    return names


stdlib, other, count = set(), set(), 0
for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
    count += 1
    for name in top_levels(line):
        (stdlib if name in sys.stdlib_module_names else other).add(name)
print("stdlib:", " ".join(sorted(stdlib)) if stdlib else "none")
print("other:", " ".join(sorted(other)) if other else "none")
print(f"statements {count}")
```

EXERCISE 2

Explore a module

Read a standard-library module name and a prefix. Import the module with `importlib.import_module`, list its public names (`dir`, excluding names starting with `_`) that start with the prefix, and print the callables sorted, then the non-callables sorted (`(none)` for an empty group).

Input: `module prefix`. **Output:** `callable: …` then `other: …`.

```
math is
```

prints

```
callable: isclose isfinite isinf isnan isqrt
other: (none)
```

INPUT 1

```
math is
```

OUTPUT 1

```
callable: isclose isfinite isinf isnan
isqrt
other: (none)
```

REFERENCE SOLUTION

```python
import importlib

module_name, prefix = input().split()
mod = importlib.import_module(module_name)
names = [n for n in dir(mod) if not n.startswith("_") and n.startswith(prefix)]
funcs = sorted(n for n in names if callable(getattr(mod, n)))
others = sorted(n for n in names if not callable(getattr(mod, n)))
print("callable:", " ".join(funcs) if funcs else "(none)")
print("other:", " ".join(others) if others else "(none)")
```

QUIZ

1. Which module holds `Counter`, `deque` and `defaultdict`?

A. `itertools`

B. `collections`

C. `functools`

D. `types`

2. Which function should time a piece of code?

A. `time.time()`

B. `time.perf_counter()`

C. `datetime.now()`

D. `time.ctime()`

3. Why must `pickle.load` never be used on data from outside your program?

A. It is slow

B. Unpickling can execute arbitrary code

C. It only works on lists

D. It is deprecated

4. Which module reads TOML in 3.11?

A. `toml`

B. `tomllib`

C. `configparser`

D. `json`

5. What is the fastest way to check what a standard function does?

A. Read the C source

B. `help(function)` in the REPL and a two-line experiment

C. Search a forum

D. Guess

ANSWER KEY · THE STANDARD LIBRARY MAP — WHERE TO LOOK BEFORE YOU WRITE IT

1. B `itertools` is iterator tools, `functools` function tools; the containers live in `collections`.

2. B `perf_counter` is a high-resolution monotonic clock; wall-clock time can jump.

3. B Pickle is a serialisation of Python objects, including callables; use JSON for untrusted data.

4. B `tomllib` is read-only and standard since 3.11; `configparser` is for INI files.

5. B The docstring shows the signature and behaviour; a small call confirms the edge case you care about.

LESSON 4 · 13 MIN

Virtual environments and packaging — venv, pip and pyproject.toml

Two projects on one machine need two different versions of the same library, and the interpreter's global `site-packages` can hold only one. The virtual environment solves that: a private directory with its own `python`, its own `pip` and its own packages, activated per project. Around it sits the packaging toolchain — `pip` to install, `requirements.txt` or `pyproject.toml` to declare what to install, and the build system that turns your package into something `pip` can install. This lesson covers creating and using a venv, `pip`'s commands and pinning, `pyproject.toml` as the one project file, editable installs, version specifiers, and the tools that automate it.

Virtual environments

```
$ python -m venv .venv                  # create: a directory with bin/ (Script
$ source .venv/bin/activate             # activate (bash/zsh);  .venv\Scripts\a
(.venv) $ python -c "import sys; print(sys.prefix)"   # points inside .venv
(.venv) $ pip install requests
(.venv) $ deactivate
```

Activation only edits `PATH` so that `python` and `pip` resolve to the venv's copies; the same effect comes from running `.venv/bin/python` directly, which is what scripts and CI usually do. Each venv is tied to the interpreter that created it; delete and recreate it rather than trying to move it. `.venv` is never committed — it is a build artefact, reproducible from the requirements. The rule that follows: **never `pip install` into the system interpreter** (`sudo pip` is how operating systems get broken), and never trust `python` without knowing which one it is; `python -m pip` always matches the interpreter you ran.

---

pip

```
pip install requests                    # latest compatible
pip install "requests⩾2.31,<3"          # a version constraint
pip install -r requirements.txt         # from a file
pip install -e .                        # this project, editable (see below)
pip uninstall requests

pip list                                # what is installed
pip show requests                       # version, location, dependencies
pip freeze > requirements.txt           # pin exactly what is installed now
```

`pip freeze` writes every installed package with `==` versions — a *lock*: anyone who installs from it gets identical versions. Distinguish two files: the *direct* dependencies you chose, with loose constraints (`requests⩾2.31`), and the *frozen* full set for reproducible deployment. Small projects keep one `requirements.txt` ; larger ones separate them, or use a tool that manages a lock file.

Version specifiers

Specifier	Meaning
`==1.4.2`	exactly
`⩾1.4`	at least
`⩾1.4,<2`	a range — the common form for a library dependency
`~=1.4`	compatible release: `⩾1.4, ==1.*`
`≠1.5.0`	exclude one

Semantic versioning — `MAJOR.MINOR.PATCH` , where a major bump may break compatibility — is a convention most packages follow, which is why `⩾1.4,<2` is the usual way to say "any 1.x from 1.4 on".

---

pyproject.toml

Since PEP 621 one file declares a project's metadata, dependencies and build system:

```
[project]
name = "shop"
version = "1.2.0"
description = "Inventory and pricing"
requires-python = "≥3.11"
dependencies = [
    "requests≥2.31,<3",
]

[project.optional-dependencies]
dev = ["pytest≥8", "mypy", "ruff"]

[project.scripts]
shop = "shop.cli:main"          # installs a `shop` command that calls shop.cl

[build-system]
requires = ["setuptools≥68"]
build-backend = "setuptools.build_meta"

[tool.ruff]
line-length = 100
```

`pip install .` builds and installs the package from it; `pip install -e .` installs it *editable* — a link to the source directory, so edits are live without reinstalling — which is how a project's own package is made importable during development (and why the `src/` layout works). `pip install ".[dev]"` adds the optional group. Tools read their settings from `[tool.*]` tables, so `pyproject.toml` replaces `setup.py`, `setup.cfg`, `requirements-dev.txt`, `.flake8` and the rest. `tomllib` (3.11) reads it from Python: `tomllib.load(open("pyproject.toml", "rb"))`.

Building and publishing, in outline

`python -m build` produces a *source distribution* (`.tar.gz`) and a *wheel* (`.whl`, a zip of the built package) in `dist/`; `twine upload dist/*` publishes them to PyPI, from where `pip install shop` fetches them. A wheel installs by unpacking, with no build step, which is why nearly everything on PyPI ships one. Version numbers come from `pyproject.toml` (or a plugin that reads git tags); a package name must be unique on PyPI.

The tools that automate it

`pip` + `venv` are always there and enough. `pipx` installs command-line tools (`black`, `httpie`) each in its own venv with the command on your path. `uv`, `poetry`, `pdm` and `hatch` manage the venv, the lock file and the build from `pyproject.toml` in one command and are what larger projects standardise on; `uv` in particular replaces `pip` and `venv` with a much faster implementation of the same operations. Whatever the tool, the artefacts are the same: a `pyproject.toml` you edit, a lock file you commit, a `.venv` you do not.

Pitfalls

Installing into the system Python, or into the wrong venv because `pip` and `python` disagree — use `python -m pip`.

Committing `.venv`.

No pins at all in deployment, so a fresh install pulls a breaking release.

Pinning `==` in a *library's* dependencies, so it cannot coexist with anything else.

Running tests against the working-directory package instead of the installed one; `pip install -e .` and the `src/` layout.

Naming a project after an existing PyPI package.

Key takeaways

One venv per project: `python -m venv .venv`, activate or call its `python` directly; never install into the system interpreter.

`pip install/uninstall/list/show`; `pip freeze` pins the full set; `-r` installs from a file; `-e .` installs your own package editable.

`>=x,<y` for library dependencies; `==` in a lock file for deployment.

`pyproject.toml` holds metadata, dependencies, optional groups, scripts, build system and tool settings.

`build` and `twine` make and publish wheels; `pipx`, `uv`, `poetry` automate the workflow around the same files.

EXERCISE 1

Requirement specifiers

Parse requirement lines of the form `name`, `name==1.2.0`, `name>=1.4,<2` (operators `==`, `!=`, `>=`, `<=`, `>`, `<`; versions are dotted integers). Then read `installed name version` lines and report for each requirement whether it is `missing`, `ok` or `conflict`, comparing versions as tuples of integers.

Input: requirement lines, a line `--`, then installed lines. **Output:** `<name>: ok|conflict|missing` per requirement, in input order.

```
requests>=2.31,<3
numpy==1.26.4
flask
--
installed requests 2.32.0
installed numpy 2.0.0
```

prints

```
requests: ok
numpy: conflict
flask: missing
```

INPUT 1

```
requests>=2.31,<3
numpy==1.26.4
flask
--
installed requests 2.32.0
installed numpy 2.0.0
```

OUTPUT 1

```
requests: ok
numpy: conflict
flask: missing
```

REFERENCE SOLUTION

```
import operator
import re
import sys

SPEC = re.compile(r"(==|!=|>=|<=|>|<)\s*([\d.]+)")
OPS = {"==": operator.eq, "!=": operator.ne, ">=": operator.ge, "<=": operator.le, '


def version(text):
    return tuple(int(p) for p in text.split("."))


def parse_requirement(line):
    m = re.match(r"[A-Za-z0-9_.-]+", line)
    name = m.group()
    constraints = [(op, version(ver)) for op, ver in SPEC.findall(line[m.end():])]
    return name, constraints


def satisfies(installed, constraints):
    return all(OPS[op](installed, ver) for op, ver in constraints)


lines = [l.strip() for l in sys.stdin if l.strip()]
split = lines.index("--")
requirements = [parse_requirement(l) for l in lines[:split]]
installed = {}
for l in lines[split + 1:]:
    _, name, ver = l.split()
    installed[name] = version(ver)
for name, constraints in requirements:
    if name not in installed:
        print(f"{name}: missing")
    else:
        print(f"{name}: {'ok' if satisfies(installed[name], constraints) else 'confl
```

EXERCISE 2

Read a pyproject

Read a `pyproject.toml` document from standard input with `tomllib.loads` and print: the project name and version, `requires-python` (or `any`), the dependencies sorted (or `none`), and each optional-dependency group as `<group>: <sorted packages>` in group-name order.

Input: TOML text. **Output:** `name <n> version <v>`, `python <spec>`, `deps <list>`, then the groups.

```
[project]
name = "shop"
version = "1.2.0"
requires-python = "⩾3.11"
dependencies = ["requests⩾2.31", "click"]

[project.optional-dependencies]
dev = ["pytest", "mypy"]
```

prints

```
name shop version 1.2.0
python ⩾3.11
deps click requests⩾2.31
dev: mypy pytest
```

INPUT 1

```
[project]
name = "shop"
version = "1.2.0"
requires-python = "⩾3.11"
dependencies = ["requests⩾2.31",
"click"]

[project.optional-dependencies]
dev = ["pytest", "mypy"]
```

OUTPUT 1

```
name shop version 1.2.0
python ⩾3.11
deps click requests⩾2.31
dev: mypy pytest
```

REFERENCE SOLUTION

```python
import sys
import tomllib

doc = tomllib.loads(sys.stdin.read())
project = doc["project"]
print(f"name {project['name']} version {project['version']}")
print(f"python {project.get('requires-python', 'any')}")
deps = sorted(project.get("dependencies", []))
print("deps", " ".join(deps) if deps else "none")
for group, packages in sorted(project.get("optional-dependencies", {}).items()):
    print(f"{group}: {' '.join(sorted(packages))}")
```

QUIZ

1. What does activating a venv actually do?

A. Copies the interpreter

B. Puts the venv's `bin` / `Scripts` first on `PATH` so `python` and `pip` resolve to its copies

C. Changes `sys.version`

D. Installs packages

2. What does `pip freeze` produce?

A. A list of direct dependencies with loose constraints

B. Every installed package pinned with `==` — a reproducible lock

C. The `pyproject.toml`

D. A wheel

3. What does `pip install -e .` do?

A. Installs the latest release from PyPI

B. Installs the current project as a link to its source, so edits are live

C. Exports requirements

D. Creates a venv

4. Which specifier means 'any 1.x from 1.4 on'?

A. `==1.4`

B. `>=1.4,<2`

C. `>1.4`

D. `~=2.0`

5. Why is `sudo pip install` a bad idea?

A. It is slow

B. It installs into the system interpreter that the OS depends on, risking breakage

C. It does not work

D. It skips dependencies

ANSWER KEY · VIRTUAL ENVIRONMENTS AND PACKAGING — VENV, PIP AND PYPROJECT.TOML

1. B Running `.venv/bin/python` directly has the same effect without activation.

2. B It captures the exact current state; direct dependencies with ranges belong in `pyproject.toml`.

3. B Editable installs are how a project's own package becomes importable during development.

4. B A range bounded by the next major version follows semantic versioning; `~=1.4` is equivalent.

5. B Per-project venvs isolate packages; `pipx` isolates command-line tools.

LESSON 5 · 14 MIN

Scripts and the command line — argv, argparse, exit codes and streams

A script is a program meant to be run from a shell, and a good one follows the conventions every other command-line tool follows: arguments and options parsed properly, `--help` that explains them, output on stdout and messages on stderr, an exit status that says whether it worked, and a structure that keeps the logic testable apart from the parsing. This lesson covers `sys.argv`, `argparse` with its argument kinds and sub-commands, the three streams and why errors go to stderr, exit codes, reading from a file or stdin interchangeably, environment variables, and the `main(argv)` shape that makes a script importable and testable.

---

sys.argv

```
import sys
print(sys.argv)          # ['tool.py', 'input.txt', '--verbose']
```

`sys.argv[0]` is the script path; the rest are the arguments as strings, split by the shell. For one or two positional arguments, reading `sys.argv[1:]` directly is fine. Anything with options, defaults or help wants `argparse`.

---

argparse

```
import argparse

def build_parser():
    p = argparse.ArgumentParser(prog="wc", description="Count lines, words and
    p.add_argument("files", nargs="*", help="input files (default: stdin)")
    p.add_argument("-l", "--lines", action="store_true", help="count lines onl
    p.add_argument("-n", "--top", type=int, default=10, metavar="N", help="sho
    p.add_argument("--format", choices=["text", "json"], default="text")
    p.add_argument("-v", "--verbose", action="count", default=0)
    return p

args = build_parser().parse_args()
args.files, args.lines, args.top, args.format, args.verbose
```

Positional arguments are named without dashes; options with `-x` / `--long`. `type=int` converts and validates (a bad value gives a clean error and exit status 2); `choices` restricts; `default` fills; `action="store_true"` makes a flag; `action="count"` makes `-vvv` a number; `nargs="*"` / `"+"` / `"?"` take several or optional values. `--help` is generated from the `help` strings, and a wrong invocation prints usage to stderr and exits with `2`. `parse_args(argv)` takes an explicit list, which is how a parser is tested without a real command line.

Sub-commands (`git commit`, `git push`) are `subparsers`:

```
sub = p.add_subparsers(dest="command", required=True)
add = sub.add_parser("add", help="add an item")
add.add_argument("name")
add.add_argument("--qty", type=int, default=1)
sub.add_parser("list", help="list items")
```

`args.command` says which ran; a `set_defaults(func=handler)` on each sub-parser lets `main` call `args.func(args)`.

The three streams

`stdin` is for data in, `stdout` for *results*, `stderr` for *messages* — progress, warnings, errors. The separation is what lets `tool input.txt > out.json` capture results while errors still reach the terminal, and lets `tool a | tool b` pipe results without messages corrupting them.

```
print(result)                                   # stdout
print("warning: skipped 3 lines", file=sys.stderr)
sys.exit("error: no input files")               # prints to stderr and exits 1
```

A program that reads either files or stdin uses `sys.stdin` when no file is named — the `-` convention — so it works in a pipeline:

```
import fileinput
for line in fileinput.input(args.files or ("-",)):     # each named file in tu
    ...
```

Exit codes

`0` success; `1` a general failure; `2` bad usage (argparse's choice); other small numbers by convention of the program. Shells, `make`, CI and callers test it: `tool && next` runs `next` only on success. `sys.exit(n)` sets it; an uncaught exception gives `1` plus a traceback, which is fine for bugs and wrong for expected failures — catch those at the top level and exit with a message:

```
def main(argv=None):
    args = build_parser().parse_args(argv)
    try:
        run(args)
    except FileNotFoundError as e:
        print(f"error: {e.filename} not found", file=sys.stderr)
        return 1
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

`main` returns the code and the guard passes it to `sys.exit`, so the function can be called in a test with a list of arguments and its return value checked.

Environment variables

`os.environ["HOME"]` reads one (`KeyError` if unset); `os.environ.get("LOG_LEVEL", "INFO")` with a default; `os.getenv` is the same. Configuration that should not be on the command

line — credentials, paths that differ per machine — comes from the environment, with the command line overriding it and a default under both. Never print a secret from the environment; never commit a `.env` file.

Structure

`build_parser()` builds the parser; `main(argv=None)` parses and dispatches; `run(args)` or per-command handlers do the work with plain arguments; the guard is one line.

Logic never calls `sys.exit` or reads `sys.argv` directly — it takes values and returns or raises, so it can be imported and tested.

Output formatting is a function of the data; `--format json` is a different formatter, not a different computation.

Progress and diagnostics go to stderr or `logging` (Module 15); results go to stdout.

`python -m shop.cli` runs it in a package (Module 12 lesson 2); a `[project.scripts]` entry (lesson 4) installs it as a command.

Testing a command-line tool

Because `build_parser()` and `main(argv)` take their input as arguments, a test is a plain function call: `main(["sum", "1", "2"])` returns `0`, and `contextlib.redirect_stdout(io.StringIO())` captures what it printed. Usage errors are testable too — construct the parser with `exit_on_error=False` (3.9) so bad input raises `argparse.ArgumentError` instead of exiting, or assert that `SystemExit` is raised with code 2. A tool whose logic lives in functions that never touch `sys.argv`, `sys.exit` or the real streams is one that a test suite can exercise completely without spawning a process.

Pitfalls

Errors printed to stdout, corrupting piped output.

`sys.exit` from deep inside the logic.

Parsing `sys.argv` by hand once options appear.

A script that cannot read stdin, so it cannot be used in a pipeline.

Secrets on the command line (visible in `ps` and shell history).

`main()` that takes no arguments and cannot be tested.

Key takeaways

`sys.argv[1:]` for trivial scripts; `argparse` for options, types, defaults, choices, flags and sub-commands, with `--help` for free.

Results to stdout, messages to stderr; read stdin when no file is given so the tool works in pipelines.

Exit `0` on success, `1` on failure, `2` on usage errors; `main(argv=None) → int` under `sys.exit(main())`.

Configuration from the environment via `os.environ.get`, overridable by options; never expose secrets.

Parser, `main`, and the logic are separate functions; the logic never touches `argv` or `exit`.

EXERCISE 1

argparse from lines

Build an `argparse` parser with sub-commands: `add name --qty N` (`qty` defaults to 1, must be an int), `remove name --force` (a flag) and `list`. Each input line is a command line; split it with `shlex.split` and parse it with `parse_args(argv)`. Print the parsed namespace as `command=add name=bolt qty=3` (attributes in the order `command`, `name`, `qty`, `force` — only those present). A line the parser rejects prints `usage error` — construct the parser with `exit_on_error=False` and catch both `argparse.ArgumentError` and `SystemExit`.

Input: command lines. **Output:** one line per input line.

```
add bolt --qty 3
remove nut --force
list
add
add nut --qty x
```

prints

```
command=add name=bolt qty=3
command=remove name=nut force=True
command=list
usage error
usage error
```

INPUT 1

```
add bolt --qty 3
remove nut --force
list
add
add nut --qty x
```

OUTPUT 1

```
command=add name=bolt qty=3
command=remove name=nut force=True
command=list
usage error
usage error
```

REFERENCE SOLUTION

```
import argparse
import contextlib
import io
import shlex
import sys


def build_parser():
    p = argparse.ArgumentParser(prog="inv", exit_on_error=False)
    sub = p.add_subparsers(dest="command", required=True)
    add = sub.add_parser("add", exit_on_error=False)
    add.add_argument("name")
    add.add_argument("--qty", type=int, default=1)
    remove = sub.add_parser("remove", exit_on_error=False)
    remove.add_argument("name")
    remove.add_argument("--force", action="store_true")
    sub.add_parser("list", exit_on_error=False)
    return p


parser = build_parser()
for line in sys.stdin:
    argv = shlex.split(line)
    try:
        with contextlib.redirect_stderr(io.StringIO()):
            args = parser.parse_args(argv)
    except (argparse.ArgumentError, SystemExit):
        print("usage error")
        continue
    parts = [f"{k}={v}" for k in ("command", "name", "qty", "force") if (v := getatt
    print(" ".join(parts))
```

EXERCISE 2

main returns the exit code

Write `main(argv) → int` around an `argparse` parser with one sub-command `sum` taking any number of tokens and a `--strict` flag: without `--strict`, non-integer tokens are skipped with a message to **stderr** and the sum is printed to stdout, returning `0`; with `--strict`, the first non-integer token prints `error: <token> is not an integer` to stderr and returns `1` without printing a sum. A usage error returns `2`. Each input line is an `argv`; call `main(shlex.split(line))` and print `exit <code>`.

Input: argv lines. **Output:** the sums on stdout and an `exit <code>` line per argv.

```
sum 1 2 x 3
sum --strict 1 x
bogus
```

prints

```
6
exit 0
exit 1
exit 2
```

INPUT 1

```
sum 1 2 x 3
sum --strict 1 x
bogus
```

OUTPUT 1

```
6
exit 0
exit 1
exit 2
```

REFERENCE SOLUTION

```
import argparse
import contextlib
import io
import shlex
import sys


def build_parser():
    p = argparse.ArgumentParser(prog="tool", exit_on_error=False)
    sub = p.add_subparsers(dest="command", required=True)
    s = sub.add_parser("sum", exit_on_error=False)
    s.add_argument("tokens", nargs="*")
    s.add_argument("--strict", action="store_true")
    return p


def main(argv):
    try:
        with contextlib.redirect_stderr(io.StringIO()):
            args = build_parser().parse_args(argv)
    except (argparse.ArgumentError, SystemExit):
        return 2
    total = 0
    for tok in args.tokens:
        try:
            total += int(tok)
        except ValueError:
            if args.strict:
                print(f"error: {tok} is not an integer", file=sys.stderr)
                return 1
            print(f"skipping {tok}", file=sys.stderr)
    print(total)
    return 0


for line in sys.stdin:
    code = main(shlex.split(line))
    print(f"exit {code}")
```

QUIZ

1. Which stream should progress messages use?

A. stdout

B. stderr — so piped results stay clean

C. A file

D. Either

2. What exit status does `argparse` use for a usage error?

A. 0

B. 1

C. 2

D. 127

3. What does `add_argument("-v", action="count", default=0)` give for `-vvv`?

A. `True`

B. `3`

C. `"vvv"`

D. An error

4. Why should `main` take `argv` and return an int?

A. Python requires it

B. So it can be called in tests with a list and its status checked, without `sys.argv` or `sys.exit`

C. For speed

D. To avoid `argparse`

5. Where should a secret such as an API key come from?

A. A command-line option

B. The environment (`os.environ.get`), never printed or committed

C. A global constant in the source

D. `input()`

ANSWER KEY · SCRIPTS AND THE COMMAND LINE — ARGV, ARGPARSE, EXIT CODES AND STREAMS

1. B Results on stdout, messages on stderr is the convention every shell pipeline relies on.

2. C Two is the conventional 'bad usage' status; one is a general failure.

3. B `count` increments per occurrence — the verbosity idiom.

4. B The guard does `sys.exit(main())`; everything else is an ordinary, testable function.

5. B Command-line arguments are visible in process lists and shell history.

CHECKPOINT · 22 MIN

Checkpoint — Modules, packages and imports

This checkpoint covers the whole module: what `import` does and the `sys.modules` cache, `from`-imports copying bindings, `__name__` and circular imports, packages with `__init__.py`, relative imports and `python -m`, the standard-library map, virtual environments with `pip` and `pyproject.toml`, and command-line scripts with `argparse`, the three streams and exit codes.

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:

- How many times does a module's top-level code run in one process, and why?
- After `from config import DEBUG`, what does rebinding `config.DEBUG` do to your `DEBUG`?
- What does `import shop` do about `shop.models`?
- Why does `python shop/cli.py` break relative imports, and what is the fix?
- What is the difference between `requests>=2.31,<3` in `pyproject.toml` and `requests==2.31.0` in a lock file?
- Which stream do error messages go to, and what exit code does argparse use for bad usage?
- Where does a data file inside a package get opened from?

The three programs are a dependency resolver that orders modules by their imports and detects cycles, a requirements parser that checks installed versions against specifiers, and a command-line tool whose arguments are parsed by `argparse` from lines of input.

EXERCISE 1

Import graph

Read lines `<module>: <imports…>` (possibly none) and an entry module on the last line `entry <name>`. Compute the order in which module bodies **finish** executing when the entry is imported — a depth-first walk where each module's imports complete before it does, each module once — and print it. If the walk meets a module that is currently being imported (a cycle), print `cycle: a → b → … → a` for the first cycle found, using the path from that module back to itself, and stop.

Input: lines, then `entry <name>`. **Output:** `order: <names>` or the `cycle:` line.

```
a: b c
b: c
c:
entry a
```

prints

```
order: c b a
```

INPUT 1

```
a: b c
b: c
c:
entry a
```

OUTPUT 1

```
order: c b a
```

REFERENCE SOLUTION

```python
import sys

graph = {}
entry = None
for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
    if line.startswith("entry "):
        entry = line.split()[1]
    else:
        name, _, rest = line.partition(":")
        graph[name.strip()] = rest.split()

done = []
in_progress = []   # the current import stack


def visit(name):
    if name in done:
        return None
    if name in in_progress:
        start = in_progress.index(name)
        return in_progress[start:] + [name]
    in_progress.append(name)
    for dep in graph.get(name, []):
        cycle = visit(dep)
        if cycle:
            return cycle
    in_progress.pop()
    done.append(name)
    return None


cycle = visit(entry)
if cycle:
    print("cycle:", " → ".join(cycle))
else:
    print("order:", " ".join(done))
```

EXERCISE 2

Requirements check

Requirements arrive as `name<specifiers>` lines and the installed set as `name==version` lines after a `--` line. Specifiers are comma-separated `op version` pairs with `==`, `!=`, `>=`, `<=`, `>`, `<`, or `~=`

(compatible release: `~=1.4` means `>=1.4,<2`; `~=1.4.2` means `>=1.4.2,<1.5`). Print each requirement's status and finish with `problems <n>` counting the missing and conflicting ones.

Input: lines, `--`, lines. **Output:** `<name>: ok|conflict|missing` per requirement, then `problems <n>`.

```
requests~=2.31
numpy>=1.26,<2
mypy
--
requests==2.32.1
numpy==2.0.0
```

prints

```
requests: ok
numpy: conflict
mypy: missing
problems 2
```

INPUT 1

```
requests~=2.31
numpy>=1.26,<2
mypy
--
requests==2.32.1
numpy==2.0.0
```

OUTPUT 1

```
requests: ok
numpy: conflict
mypy: missing
problems 2
```

REFERENCE SOLUTION

```
import operator
import re
import sys

SPEC = re.compile(r"(~=|==|!=|>=|<=|>|<)\s*([\d.]+)")
OPS = {"==": operator.eq, "!=": operator.ne, ">=": operator.ge, "<=": operator.le, '


def version(text):
    return tuple(int(p) for p in text.split("."))


def expand(op, ver):
    if op != "~=":
        return [(op, version(ver))]
    v = version(ver)
    upper = v[:-1]
    upper = upper[:-1] + (upper[-1] + 1,)
    return [(">=", v), ("<", upper)]


def parse(line):
    m = re.match(r"[A-Za-z0-9_.-]+", line)
    constraints = []
    for op, ver in SPEC.findall(line[m.end():]):
        constraints.extend(expand(op, ver))
    return m.group(), constraints


lines = [l.strip() for l in sys.stdin if l.strip()]
split = lines.index("--")
installed = {}
for l in lines[split + 1:]:
    name, _, ver = l.partition("==")
    installed[name] = version(ver)
problems = 0
for l in lines[:split]:
    name, constraints = parse(l)
    if name not in installed:
        status = "missing"
    elif all(OPS[op](installed[name], v) for op, v in constraints):
        status = "ok"
    else:
        status = "conflict"
    if status != "ok":
        problems += 1
    print(f"{name}: {status}")
print(f"problems {problems}")
```

EXERCISE 3

An inventory command

Build a sub-command tool with `argparse` (`exit_on_error=False`): `add name --qty N` (default 1), `remove name [--all]` (removes one unit, or every unit with `--all`), `show` (prints `name qty` lines in name order, or `empty`). Each input line is a command line split with `shlex.split` ; a usage error prints `usage error` ; removing an unknown item prints `unknown item` . Keep the state across lines.

Input: command lines. **Output:** one line per `show` row, plus the error lines.

```
add bolt --qty 3
add nut
remove bolt
remove screw
show
remove nut --all
show
```

prints

```
unknown item
bolt 2
nut 1
bolt 2
```

INPUT 1

```
add bolt --qty 3
add nut
remove bolt
remove screw
show
remove nut --all
show
```

OUTPUT 1

```
unknown item
bolt 2
nut 1
bolt 2
```

REFERENCE SOLUTION

```
import argparse
import contextlib
import io
import shlex
import sys


def build_parser():
    p = argparse.ArgumentParser(prog="inv", exit_on_error=False)
    sub = p.add_subparsers(dest="command", required=True)
    add = sub.add_parser("add", exit_on_error=False)
    add.add_argument("name")
    add.add_argument("--qty", type=int, default=1)
    remove = sub.add_parser("remove", exit_on_error=False)
    remove.add_argument("name")
    remove.add_argument("--all", action="store_true")
    sub.add_parser("show", exit_on_error=False)
    return p


stock = {}
parser = build_parser()
for line in sys.stdin:
    try:
        with contextlib.redirect_stderr(io.StringIO()):
            args = parser.parse_args(shlex.split(line))
    except (argparse.ArgumentError, SystemExit):
        print("usage error")
        continue
    if args.command == "add":
        stock[args.name] = stock.get(args.name, 0) + args.qty
    elif args.command == "remove":
        if args.name not in stock:
            print("unknown item")
        elif args.all or stock[args.name] == 1:
            del stock[args.name]
        else:
            stock[args.name] -= 1
    elif args.command == "show":
        if stock:
            for name in sorted(stock):
                print(name, stock[name])
        else:
            print("empty")
```

CHECKPOINT QUESTIONS

1. What happens on the second `import json` in a program?

A. The file is re-read

B. `sys.modules["json"]` is returned; nothing runs

C. An error

D. A reload

2. Which import form keeps the origin of a name visible at every use?

A. `from m import x`

B. `import m` and `m.x`

C. `from m import *`

D. `import m as _`

3. What does `__init__.py` do?

A. Nothing; it is optional decoration

B. Marks the directory as a package and runs when the package is imported — typically re-exporting the API

C. Lists the submodules

D. Runs the tests

4. `shop/utils/text.py` contains `from ..models import Product`. What does it import?

A. `shop.utils.models`

B. `shop.models`

C. `models` from the top level

D. It is an error

5. Which statement about `sys.stdlib_module_names` is true?

A. It lists installed third-party packages

B. It is a frozenset of standard-library top-level module names (3.10+)

C. It is a list of imported modules

D. It does not exist

6. What is a wheel?

A. A source archive

B. A built distribution (`.whl` , a zip) that installs by unpacking, with no build step

C. A virtual environment

D. A lock file

7. Where do a project's dependencies and tool settings live today?

A. `setup.py`

B. `pyproject.toml`

C. `requirements.txt` only

D. `Makefile`

8. What does `nargs="*"` mean in `add_argument` ?

A. Exactly one value

B. Zero or more values, collected into a list

C. A flag

D. A required value

9. Which exit code should a script return when it succeeded?

A. 1

B. 0

C. 2

D. -1

10. What does `parse_args()` do with an invalid argument by default?

A. Returns `None`

B. Prints usage to stderr and exits with status 2

C. Raises `ValueError`

D. Ignores it

11. Which is the right way for a tool to accept input from a pipe?

A. Require a file argument

B. Read `sys.stdin` when no file is given

C. Use `input()` with a prompt

D. Read the clipboard

12. Why must `.venv` not be committed?

A. It is too small

B. It is a machine-specific build artefact, reproducible from the requirements

C. Git cannot store it

D. It contains secrets

ANSWER KEY · CHECKPOINT — MODULES, PACKAGES AND IMPORTS

1. B Modules are cached by name in `sys.modules`.

2. B Qualified access documents where `x` came from and cannot collide with local names.

3. B It may be empty, but it is what makes the directory a regular package.

4. B Two dots go from the package `shop.utils` up to `shop`.

5. B `sys.modules` is what has been imported; `stdlib_module_names` is what ships with the interpreter.

6. B Nearly every package on PyPI ships wheels so `pip install` needs no compiler.

7. B PEP 621 metadata, optional groups, scripts, the build system and `[tool.*]` tables are all in one file.

8. B `+` is one or more, `?` is optional single; `*` is any number.

9. B Zero is success everywhere; non-zero values are failures that shells and CI act on.

10. B `exit_on_error=False` makes it raise `ArgumentError` instead, for programs that must keep running.

11. B That convention (often with `-` meaning stdin) is what makes tools composable.

12. B Commit the requirements or lock file; recreate the environment anywhere from it.

CodeKairo · Python study plan · Module 12: Modules, packages and imports. Solve the exercises online to track your progress and earn XP.