

Lecture 11

Modular Programming and Command-Line Arguments

Why modularize?

Modular Programming and Command-Line Arguments · Lecture 11

Learning outcomes

- Explain why programs are split into modules
- Distinguish definitions from execution and predict what happens on import
- Build a two-file program using meaningful imports
- Read command-line arguments with `sys.argv` and `argparse`
- Locate optional topics: packages, virtual environments and distribution

Today: functions → modules → entry points → CLI → project structure

A module is a file of Python code; a function is a named reusable operation.

From script to application

- A large script mixes data collection, calculations and reporting
- Changes become risky because unrelated responsibilities share one file
- Modularity means organizing related functions together
- Separating core logic from interface lets the same calculations serve different programs

ONE SCRIPT

read input → compute → print

TWO MODULES

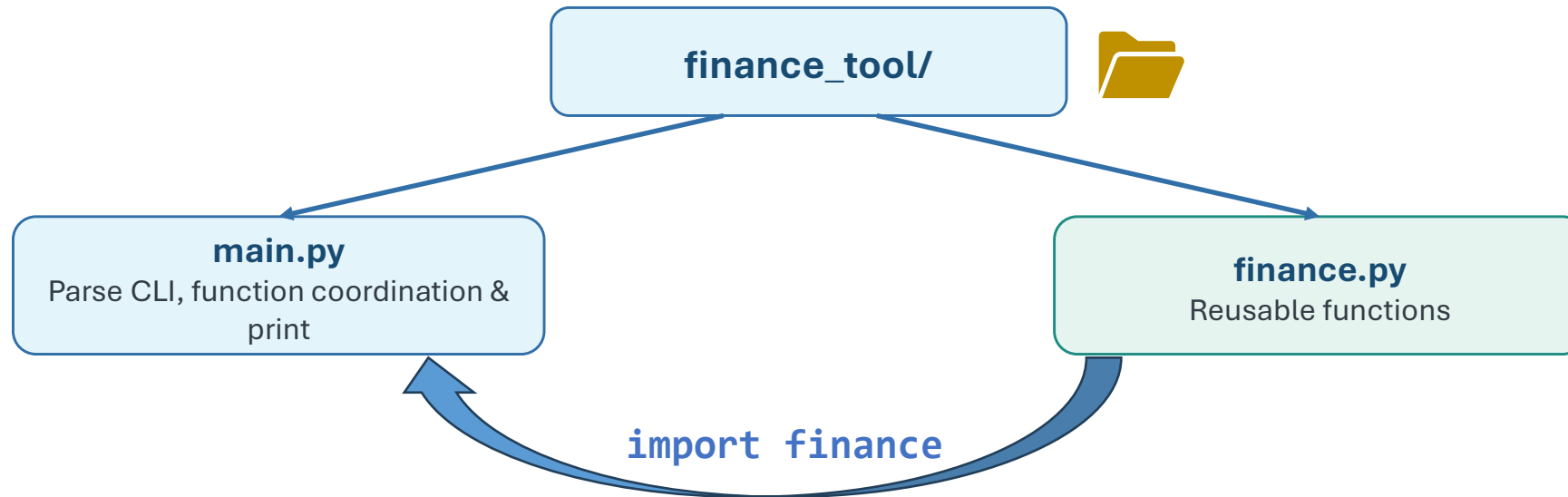
main.py → analysis.py

interface → reusable logic

A module is not a replacement for good functions: it organizes them.

Organizing a small Python application

A module tree is an organizational tool: code lives in files with distinct responsibilities.



Running `main.py` invokes the interface; importing `finance.py` gives access to calculations without prompting the user.

A sensible folder structure clarifies module ownership, not just file location

What is a Python module?

- A .py file is normally **importable** as a Python **module**
- Its top-level statements are evaluated when first imported
- Definitions declared in it can be used from another file (functions, constants...)
- File and module names should be simple, descriptive and different from Standard Library module names.

`analysis.py`

```
def total(values):  
    return sum(values)
```

`main.py`

```
import analysis  
print(analysis.total([2, 3])) # 5
```

`analysis.total` explicitly states where the function belongs

A namespace prevents name clashes

- Two files may define **functions with the same name**
- Using import module keeps each **name attached to its module**, so we can distinguish their origin
- `from <module> import <function>` is concise but ownership is less visible
- Do not use `from module import *` in beginner projects
- If the name is too long, you may use aliases

```
import math  
print(math.sqrt(25))
```

```
from math import sqrt  
print(sqrt(25))
```

```
import math as m  
print(m.sqrt(25))
```

Prefer `import module` in the course whenever possible

Imports execute top-level code

- Python **executes the imported file** once per normal process (then caches it)
- Function bodies are **defined, not called**, during import
- Any top-level `print()`, `input()` or file write is a side effect of importing
- This is why the entry-point code belongs under a guard (see next slides)

`helper.py`

```
print("Importing helper")  
  
def square(x):  
    return x * x
```

`main.py`

```
import helper # prints once  
  
print(helper.square(4))
```

Do not place interactive work at import time; a module contains only constants and definitions

Organize a two-file project

- Place the *entry-point* (the “main” file that you run) and helper file in **one project folder**
- Start `main.py` from that folder so imports resolve predictably
- Use descriptive lower-case module names, no spaces or hyphens (use `_` to separate words in case)
- Use separate files for temporary experiments or practice code that is not part of the main program

```
sales_tool/  
    main.py  
    analysis.py
```

Project folder

Python program

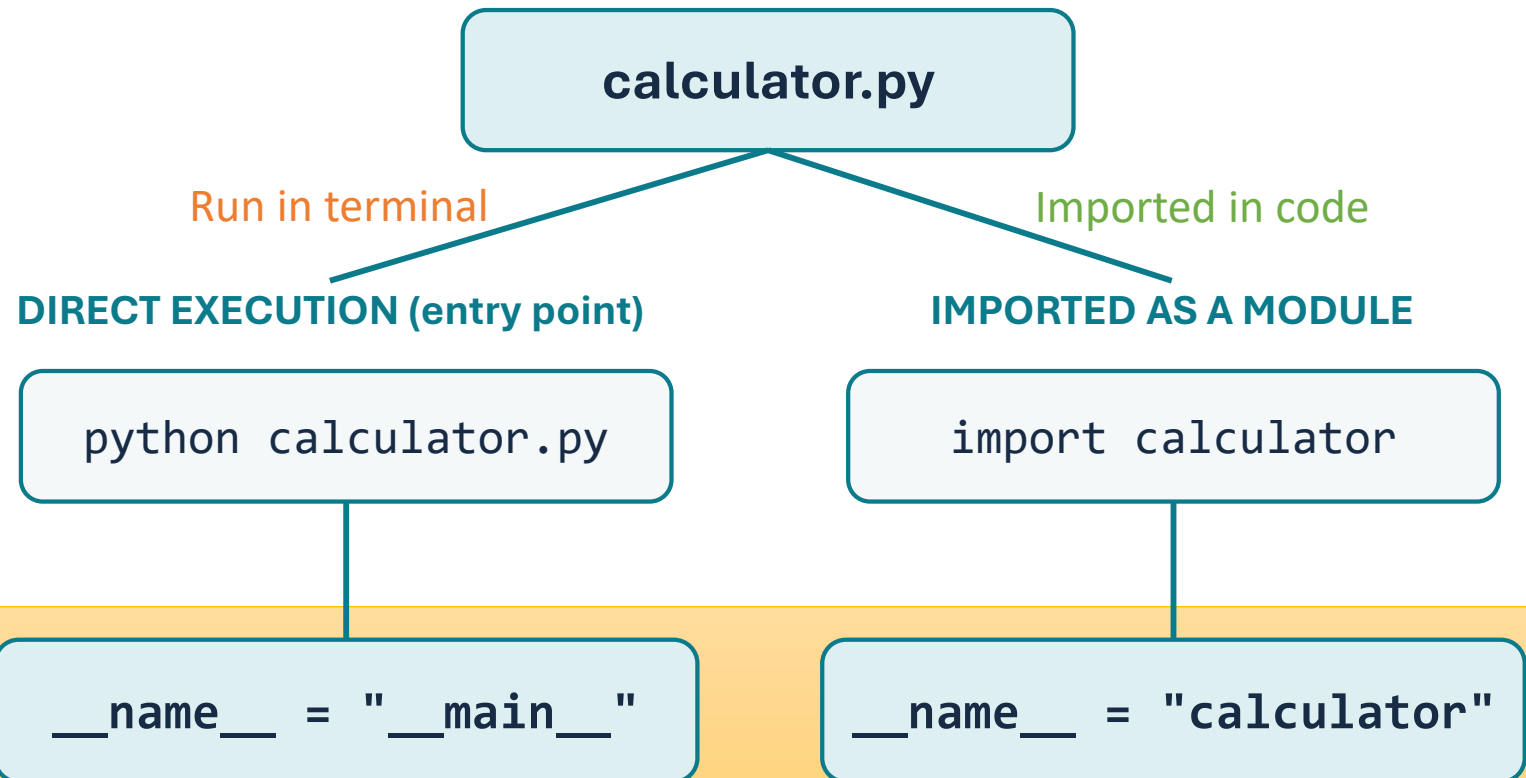
```
main.py  
import analysis
```

The two files should run together without copying the helper code

A special variable `__name__` to know who I am

Python automatically assigns a special variable named `__name__` to every module. Its value depends on how the module is loaded

- If you are checking the value of `__name__` in your **main file** (the one you run directly), the value is set by default to `"__main__"`
- If you are checking the value of `__name__` in a **module** (a file that you have imported in the main code), the value is set by default to the **name of the file itself**.



If you check the variable `__name__` inside `calculator.py` code, you will see:

`"__main__"` is a special STRING identifying the top-level execution environment, not a function.

Why does importing a file sometimes print something?

At the first normal import, Python executes statements at module level, including `print()` and function calls.

`calculator.py`

```
def add(a, b):  
    return a + b  
  
print("Calculator ready!")  
print(add(10, 5))
```

`report.py`

```
import calculator  
  
result = calculator.add(3, 4)  
print("Result:", result)
```

The two `print()` calls in `calculator.py` also run during the import

Output in terminal

```
>> python report.py  
Calculator ready!  
15  
Result: 7
```

Defining a function does NOT execute its body; a top-level function CALL does execute

The main guard: run code only when executed directly

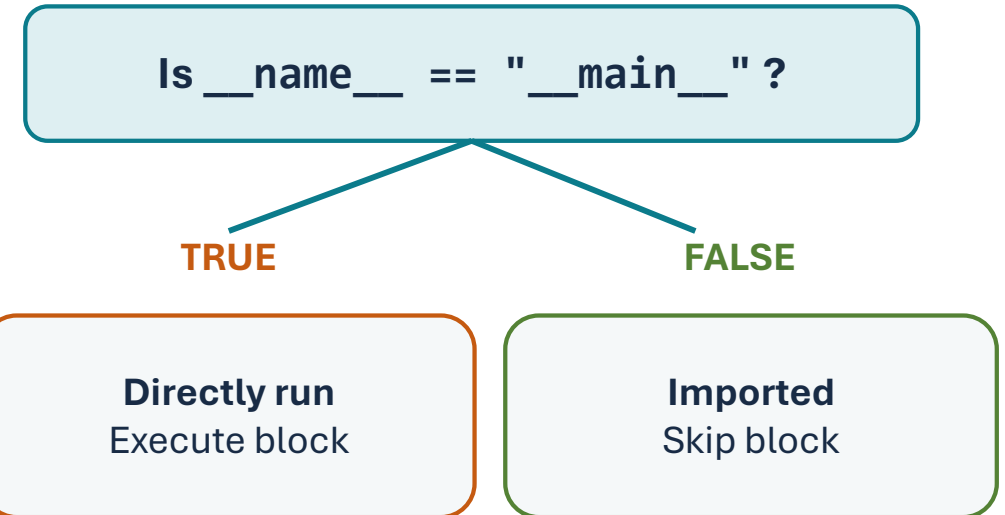
We protect the part intended to start the program with a conditional statement called **the main guard**

calculator.py – revised

```
def add(a, b):  
    return a + b  
  
if __name__ == "__main__":  
    print("Calculator ready!")  
    print(add(10, 5))
```

report.py

```
import calculator  
  
if __name__ == "__main__":  
    result = calculator.add(3, 4)  
    print("Result:", result)
```



Both paths still define `add()`, so another file can import and call it

Running `calculator.py` prints two lines. Importing it from `report.py` prints only "Result: 7".

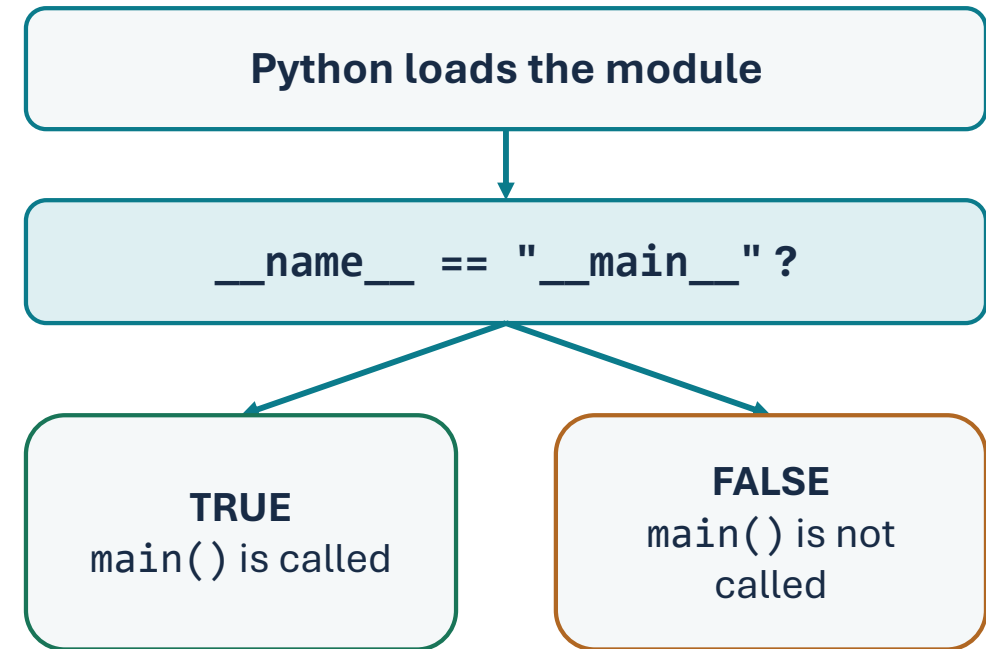
main() vs "__main__": two different things

main() is an ordinary function chosen by the programmer to coordinate the program

A complete example

```
def add(a, b):  
    return a + b  
  
def main():  
    x = float(input("First: "))  
    y = float(input("Second: "))  
    print("Sum:", add(x, y))  
  
if __name__ == "__main__":  
    main()
```

"__main__" is a special string assigned by Python to __name__



The functions remain available after import

A main() function is optional; the if __name__ == "__main__": guard is also optional

Main guard: the basic pattern

- `__name__` identifies the execution context of a module
- It equals `"__main__"` in a directly executed file
- It equals its module name when imported
- A guard prevents an imported file from launching its interface
- This is for any file you write. However, some Python files are intended only to be imported rather than executed directly, so the `main()` function and the `if` statement are completely omitted

```
def main():  
    print("Sales report")  
  
if __name__ == "__main__":  
    main()
```

Defining `main()` does not run it: the guarded call does

Functions and the execution workflow

Separating responsibilities

- **Calculation function:** performs a specific operation using parameters and returns its result (no `input()` or `print()`)
- **Main function:** reads user inputs, coordinates function calls without duplicate calculation, and displays the result
- Why separate them? The same calculation can be reused with different inputs and tested independently.
- If you prefer, you do not need to write a `main()` function, but simply write your code directly in the `if __name__ == "__main__":` it is just a question of style.

```
def calculate_total(price, quantity):  
    return price * quantity  
  
def main():  
    price = float(input("Price: "))  
    quantity = int(input("Quantity: "))  
  
    total = calculate_total(price, quantity)  
    print("Total:", total)  
  
if __name__ == "__main__":  
    main()
```

Avoid `input()` inside general-purpose analysis functions

Import vs. direct execution

The same Python file can play two different roles

calculator.py

```
def add(a, b):  
    return a + b  
  
if __name__ == "__main__":  
    print("Calculator ready!")  
    print(add(10, 5))
```

report.py

```
import calculator  
  
if __name__ == "__main__":  
    result = calculator.add(3, 4)  
    print("Result:", result)
```

Definitions are available in both cases; the protected program entry point runs only when executed directly

Import vs. direct execution

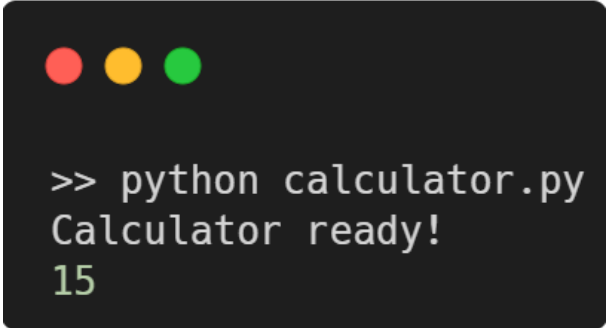
The same Python file can play two different roles

calculator.py

```
def add(a, b):  
    return a + b  
  
if __name__ == "__main__":  
    print("Calculator ready!")  
    print(add(10, 5))
```

report.py

```
import calculator  
  
if __name__ == "__main__":  
    result = calculator.add(3, 4)  
    print("Result:", result)
```



```
>> python calculator.py  
Calculator ready!  
15
```

calculator.py is the entry-point
→ `__name__ == "__main__"` inside calculator.py

→ Main code with the two prints is executed

report.py is not involved in the execution

Definitions are available in both cases; the protected program entry point runs only when executed directly

Import vs. direct execution

The same Python file can play two different roles

calculator.py

```
def add(a, b):  
    return a + b  
  
if __name__ == "__main__":  
    print("Calculator ready!")  
    print(add(10, 5))
```

report.py

```
import calculator  
  
if __name__ == "__main__":  
    result = calculator.add(3, 4)  
    print("Result:", result)
```



```
>> python report.py  
Result: 7
```

report.py is the entry-point

calculator.py is imported as a module

→ `__name__ == "__main__"` inside report.py

→ `__name__ == "calculator"` inside calculator.py

→ Main code of report.py is executed

→ Main code of calculator.py is not executed

Definitions are available in both cases; the protected program entry point runs only when executed directly

Predict `__name__` in two situations

- `__name__` is not a keyword; it is a normal variable
- If you want to see its value, you can print it
- This guard is a **convention**, not a new Python keyword. According to Python's philosophy, until you follow conventions, it guarantees it works, but you can also change the rules at your own risk.

```
print(__name__)
```

```
# Direct execution: __main__
```

```
# Imported module as analysis: analysis
```

Importing a module should not unexpectedly ask for input

Command-line interface

Modular Programming and Command-Line Arguments · Lecture 11

Two ways to supply input

- `input()` waits **while a program is already running**
- Command-line arguments are written **when launching the program**
- They make runs repeatable, easy to automate and easy to document
- A CLI (Command Line Interface) is an interface, not a different computation algorithm
- NOTE: when you click on the “play” button in VScode, in the terminal you are actually running the python command with the path of your current file, without additional arguments.

INTERACTIVE

```
python tool.py
```

```
Amount: 50
```

value typed by the user at running time

COMMAND LINE

```
python tool.py 50
```

CLI arguments are supplied before Python starts executing `main()`.

The terminal command anatomy

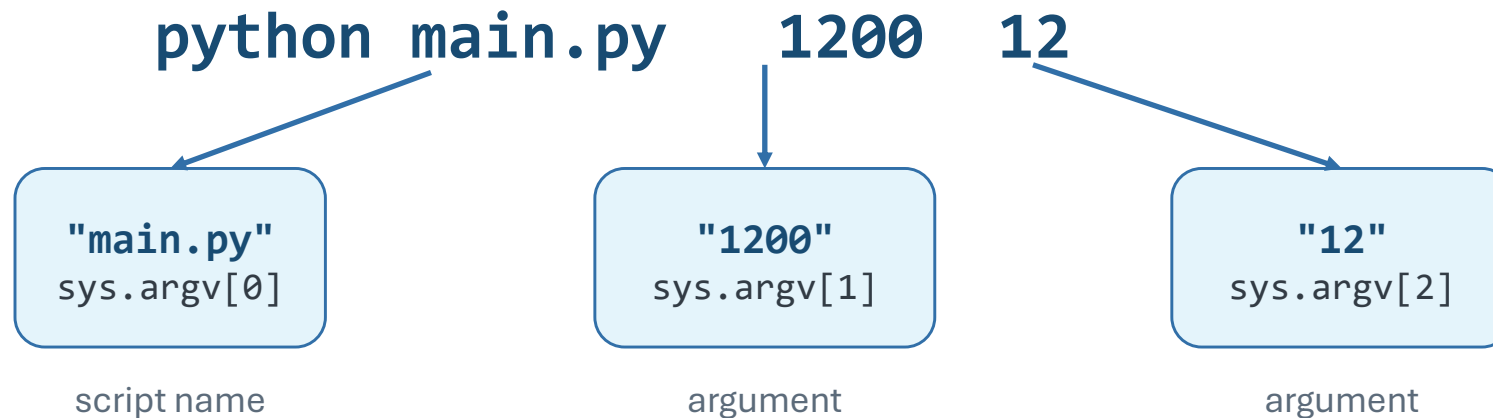
- The first word invokes Python; the next identifies a **script**; remaining tokens are **arguments**
- **Spaces** delimit separate arguments; quotes can group values containing spaces
- The operating-system Shell (the interpreter used in terminal for the prompt) passes the arguments **as text**
- All arguments require validation and conversion by the program.

```
python main.py 1200 12
|           |      |  ^ arg2
|           |      ^  arg1
|           ^  entry-point script
^  interpreter
```

From a terminal command to `sys.argv`

A terminal splits the command into tokens; Python receives argument strings contained in a special list.

- To read the arguments in the code, we need to import the module `sys`
- `sys` provides a special variable `argv` (argument vector) of type **list** containing the argument strings in the given order
- **Index 0** identifies the **name of the script** (e.g. `main.py`), while 1, 2, ... identify its arguments.



`len(sys.argv) == 3` because index 0 is the script name

As usual, convert with `float()` or `int()` on the values you need for calculations

A number written on the command line is still a string inside `sys.argv`

`sys.argv`: inspect the arguments

- Import the `sys` module from the Standard Library
- `sys.argv` is a list of strings
- Index 0 identifies the script invocation; indexes 1, 2, ... are user values
- Use `len(sys.argv)` before accessing a potentially absent position (as usual with lists)
- Convert possible numeric values before any computation
- Domain Validation as usual with input

```
import sys
```

```
print(sys.argv)
```

```
# python demo.py 50 1.2
```

```
# ["demo.py", "50", "1.2"]
```

sys.argv: complete minimal example

- Check how many tokens were supplied before indexing: are all the argument you need present?
 - Print the correct usage when argument count is wrong
- Convert using `float()` or `int()`, depending on the specification
- Handle malformed numeric strings through validation or exceptions when taught

```
import sys
if len(sys.argv) != 3:
    print("Usage: python main.py PRICE QTY")
else:
    price = float(sys.argv[1])
    quantity = int(sys.argv[2])
    print(price * quantity)
```

The script name counts in `len(sys.argv)`: two inputs mean length 3.

An alternative solution: argparse module

- Manual positional indexing becomes difficult when the interface grows
- We would like to pass arguments to a script like we do for functions; Something like:
 - `python main.py 1200 --year 2026 --month 1 --risk_level low`
- We would also like to **parse the arguments** automatically without checking explicitly wrong entries
- argparse is a module that allows to do this in a simple way.

Why argparse?

- argparse declares expected values once, in your code, and produces help text
- It can **convert types**, enforce **choices** (only some values allowed) and provide **defaults values**
- It is a Standard Library module; no installation is required

```
import argparse
parser = argparse.ArgumentParser(
    description="Invoice tool"
)
parser.add_argument("amount", type=float)
args = parser.parse_args()

original_amount = args.amount
```

argparse deals with parsing; your functions still perform computations.

[argparse — Parser for command-line options, arguments and subcommands — Python 3.15.0 documentation](#)

Standard procedure for using argparse

- Import the module

```
import argparse
parser = argparse.ArgumentParser(
    description="Invoice tool"
)
parser.add_argument("amount", type=float)
args = parser.parse_args()

original_amount = args.amount
```

argparse deals with parsing; your functions still perform computations.

Standard procedure for using argparse

- Import the module
- Create a parser object calling `ArgumentParser()`
 - It is possible to give an optional description string used for prints

```
import argparse
parser = argparse.ArgumentParser(
    description="Invoice tool"
)
parser.add_argument("amount", type=float)
args = parser.parse_args()

original_amount = args.amount
```

argparse deals with parsing; your functions still perform computations.

Standard procedure for using argparse

- Import the module
- Create a parser object calling `ArgumentParser()`
 - It is possible to give an optional description string used for prints
- call `.add_argument()` for each argument you want to create providing
 - the name (mandatory)
 - optional constraints (e.g. mandatory type)

```
import argparse
parser = argparse.ArgumentParser(
    description="Invoice tool"
)
parser.add_argument("amount", type=float)
args = parser.parse_args()

original_amount = args.amount
```

argparse deals with parsing; your functions still perform computations.

Standard procedure for using argparse

- Now, you can return the parsed arguments using method `.parse_args()`
- the returned object is of type `Namespace`
- it can be stored in a variable and allows to obtain the individual arguments

```
import argparse
parser = argparse.ArgumentParser(
    description="Invoice tool"
)
parser.add_argument("amount", type=float)
args = parser.parse_args()

original_amount = args.amount
```

Standard procedure for using argparse

- Now, you can return the parsed arguments using method `.parse_args()`
- the returned object is of type `Namespace`
- it can be stored in a variable and allows to obtain the individual arguments
- Each argument is an *attribute* of the `Namespace` object with the same name given to the argument. It can be retrieved using dot-notation like methods (without `()`)

```
import argparse
parser = argparse.ArgumentParser(
    description="Invoice tool"
)
parser.add_argument("amount", type=float)
args = parser.parse_args()

original_amount = args.amount
```

Standard procedure for using argparse

- Now, you can return the parsed arguments using method `.parse_args()`
- the returned object is of type `Namespace`
- it can be stored in a variable and allows to obtain the individual arguments
- Each argument is an *attribute* of the `Namespace` object with the same name given to the argument. It can be retrieved using dot-notation like methods (without `()`)

```
import argparse
parser = argparse.ArgumentParser(
    description="Invoice tool"
)
parser.add_argument("amount", type=float)
args = parser.parse_args()

original_amount = args.amount
```

- *Attribute* is a generic name exposed by an object you can access with dot notation
- Methods (`l.append`) or functions and constants imported from a module (`math.sqrt`, `math.pi`) are attributes

Positional arguments with argparse

Base case: positional arguments

- **Positional arguments are required** unless specified otherwise
- Their declaration order determines their command-line order
- `type=float` and `type=int` request automatic conversion
- Parsed values are available as attributes of the result namespace.

EXAMPLE

```
parser = argparse.ArgumentParser()  
  
parser.add_argument("price", type=float)  
parser.add_argument("quantity", type=int)  
args = parser.parse_args()  
print(args.price * args.quantity)
```



```
$> python main.py 12.5 3
```

Try omitting an argument: `argparse` prints a usage error.

Options and defaults

- An argument beginning with `--` is a **named option**
- Optional arguments may be omitted when they have a default (similar to func. params)
- `choices` parameter restricts input to a declared list
 - A message is automatically displayed if the input does not match the available choices.
- Using a special argument `--help` automatically displays supported arguments and descriptions (no definition required)

```
parser.add_argument(  
    "--mode",  
    choices=["gross", "net"],  
    default="gross"  
)
```

```
python main.py 10 --mode net
```

```
python main.py --help  
Explains the usage of the script
```

Boolean options with `action="store_true"`

- Some command-line options are **switches**: their presence enables a behaviour, **without requiring an additional value**
- With `argparse`, `action="store_true"` creates this kind of **optional flag**
 - When the flag is absent, its attribute is `False`
 - When it appears, `argparse` sets that attribute to `True`
- Unlike a normal argument, `--loud` is **written alone**: do not add `True` or `False` after it

`greet.py`

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--loud",
                    action="store_true")
args = parser.parse_args()
print("HELLO!" if args.loud else "Hello!")
```

Terminal: flag absent

```
$ python greet.py
Hello!
```

Terminal: flag present

```
$ python greet.py --loud
HELLO!
```

How many values? The nargs parameter

nargs tells argparse how many command-line values belong to one argument.

VALUE OF nargs	BEHAVIOUR
omitted	One value (the usual default)
nargs="?"	Zero or one value
nargs="*"	Zero or more values → list
nargs="+"	One or more values → list
nargs=3	Exactly three values → list

With type=float, each received value is converted to float.

EXAMPLE · SALES REPORT

```
parser.add_argument(  
    "sales", type=float, nargs="+"  
)  
args = parser.parse_args()
```

Terminal:

```
python main.py 120 85 95
```

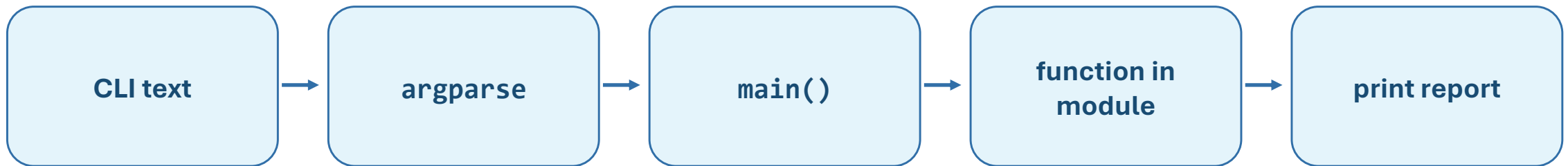
Result:

```
args.sales == [120.0, 85.0, 95.0]
```

KEY IDEA: nargs controls the number of values; type controls how each value is converted.

One responsibility per stage

A clear program is a sequence of responsibilities.



Separate parsing, processing and reporting. This makes individual functions testable.

Remember that:

- `argparse` handles missing arguments and invalid type conversions
- Your own functions must still reject impossible domain values (e.g. months ≤ 0)

argparse with a custom conversion function

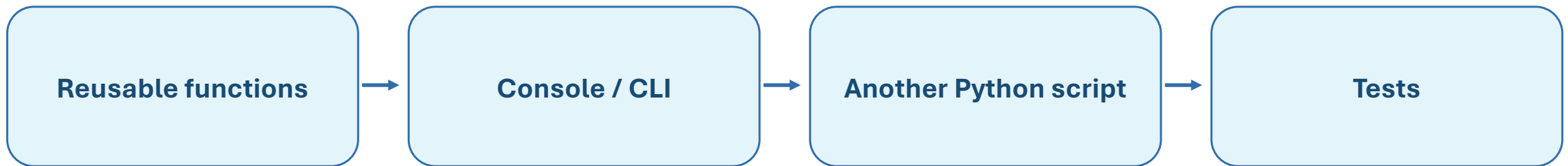
- For simple inputs, `type=float` or `type=int` is enough
- For a comma-separated list we need to convert several items at once
 - We can pass a `function` name to `type=`; `argparse` calls it on the argument string (similar to `key for sorted()`)
- In the example, if `float()` raises `ValueError`, `argparse` presents an invalid-value message.
- This avoids requiring to write `try/except` before Lecture 13.

```
def numeric_list(text):  
    l = [ float(p.strip())  
          for p in text.split(",") ]  
    return l  
  
parser.add_argument("numbers", type=numeric_list)
```

The custom parser handles syntax; domain rules are checked separately.

Modularity: one design, multiple interfaces

A module can be imported in different entry points.



Reuse calculation functions without copying their implementation in multiple scripts .py used as entrypoint. When you make changes to a function, the new functionality is shared among all the scripts. Otherwise, you may have many duplicates of the same code that you have to maintain separately.

Engineering and extensions

Modular Programming and Command-Line Arguments · Lecture 11

The import dependency graph

- An arrow $A \rightarrow B$ means that A imports and uses B
- Prefer a simple direction: interface $\rightarrow$ domain logic
- Avoid circular dependencies where modules import each other
- A helper should not import `main.py` to fetch user input.

```
main.py  →  finance.py  
    |  
    └──→ validation.py
```

Do NOT make:

```
main.py ↔ finance.py
```

Small acyclic designs are easier to reason about.

Packages: a larger unit

ADVANCED EXTENSION · no package implementation required for the test

- A package groups modules within a named folder
- A regular package traditionally contains `__init__.py` even if empty
- The dot syntax accesses submodules inside the package namespace
- We will only recognize the idea; exercises use simple neighboring `.py` modules, but this may be **useful for the project**.

```
reports.py
```

```
finance/
```

```
    __init__.py
```

```
    fees.py
```

```
from finance import fees
```

```
# or
```

```
import finace.fees
```

Where does Python functionality come from?

ADVANCED EXTENSION · further reading

Python provides many modules already, in the Standard Library (meaning you can immediately import them in your code); other packages can be added when a project needs them

Python Standard Library

Included with a normal Python installation
Examples: sys, argparse, csv, json, math

```
import json  
  
# No separate pip installation  
# needed
```

Third-party packages

Developed and distributed separately
Examples: requests, pandas, matplotlib

```
import requests  
  
# Must first be available in your  
# environment
```

A *package* is a way to distribute extra functionality; a *dependency* is a package that a project needs

What is pip, and what is Conda?

ADVANCED EXTENSION · further reading

Sometimes, we need something more.

Both **pip** and **Conda** help manage software, but they have different roles and scopes

pip

Installs Python packages, usually from PyPI

```
python -m pip --version  
python -m pip install pandas
```

pip (Python package installer):

- add/remove/update third-party packages
- downloads from the Python Package Index (PyPI), and installs them **into the selected Python environment**
- It **does not manage environments**

Conda

Manages environments and packages, including Python itself

```
conda --version  
conda install numpy
```

Conda: package + environment manager

- add/remove/update third-party packages
- download from Conda channels
- creates **separate environments**, each with its own Python version and installed packages; ("base" default environment)

pip, venv and Conda: what can they do?

ADVANCED EXTENSION · further reading

venv is a module included with Python: it creates a separate environment for a project, where **pip** can install packages without affecting other projects. **Conda** can manage both packages and environment.

Three tools with different purposes: installing packages, creating isolated environments, and managing Python versions.

Capability	pip	venv	Conda
Install / update / remove Python packages	Yes	No	Yes
Create isolated project environments	No	Yes	Yes
Install / manage Python versions	No	No ¹	Yes
Manage non-Python software packages	No ²	No	Yes

¹ venv uses an existing Python interpreter. ² pip primarily manages Python packages and their dependencies.

What does (base) mean in the terminal?

ADVANCED EXTENSION · further reading

Conda environments are separate workspaces containing a Python interpreter and installed packages. This helps different projects use different package versions without interfering with one another.

Example terminal prompt:

```
(base) C:\Users\Student> python --version
```

The exact appearance of the prompt varies between Windows, macOS, Linux and terminal settings.

Conda environment: base

Default Conda environment

Python interpreter

The Python version that runs scripts

Available packages

Installed in that environment

(base) indicates which environment is active; it is not part of the Python command and not an error. "base" is the default environment used by Conda, but you may create many of them for your projects.

To activate your environment use:

```
$> conda activate env_name
```

Why use different environments?

ADVANCED EXTENSION · further reading

Different projects may require different package versions, which can conflict, or even different Python versions.

Project A

Package X — version 1
Other project-specific packages
Requires Python 3.9

Project A could use its own environment.

Conda environment: base

The default shared environment

Conda environment: envA

Requirements suitable for Project A

Conda environment: envB

Requirements suitable for Project B

Project B

Package X — version 2
Other project-specific packages
Requires Python 3.14

Project B could use another environment.

A virtual environment is not a virtual machine: it does not create another operating system. It separates Python and the packages available to a project.

For our lessons, we will continue working in the preconfigured Conda base environment.

Sharing source versus publishing a package

ADVANCED EXTENSION · further reading

- Potentially, your package can be published and ready to be downloaded by any programmer
- Publishing to PyPI requires additional packaging metadata and steps
- Distribution is outside the assessed scope.

```
pip install mypackage
```

Learn more – official documentation

ADVANCED EXTENSION · further reading

Optional resources: open the links if you want to explore beyond the course.

[Python Tutorial — Modules](#)

How Python finds and imports modules.

[Python Tutorial — Virtual Environments and Packages](#)

An introduction to venv, pip and requirements files.

[Python Packaging User Guide — Installing Packages](#)

Step-by-step explanations of package installation and pip.

[Conda Documentation — Managing Environments](#)

What base means, and how Conda environments are created and activated.

Reading only: none of these tools or commands is required for the assessed programming exercises.