Python for Absolute Beginners

From **zero programming experience** to writing and shipping your own Python programs.

13
Sections
Setup to classes, in order

295+
Concepts
Each one is a demo you can run

60+
Practice Problems
You write the code

60+
Workshop Problems
A bigger task per step
optional, for more practice

2
Real Projects
Built with you, line by line

No experience needed

We install Python together, then start at **print()**. Nothing is assumed.

Every solution included

Try it yourself first, then compare with the **full solution file**.

Every line on screen

You watch each line being **typed and run**. Nothing is pasted in already finished.

© Kalyan Reddy Daida

Your Python Journey

33h 01m OF VIDEO

13 sections, 295+ concepts and 2 real projects. Every concept taught on screen, line by line.

Part 1 Foundations

01 **Setup** — 6 steps — 28m

→

02 **Variables & Data Types** — 32 concepts — 1h 43m

→

03 **Strings & f-strings** — 20 concepts — 1h 26m

→

04 **Operators & Conditionals** — 19 concepts — 1h 38m

→

05 **Collections** — 51 concepts — 3h 37m

→

06 **Loops & Comprehensions** — 45 concepts — 4h 08m

→

07 **Functions** — 47 concepts — 5h 16m

→

MILESTONE PROJECT 1

CLI Expense Tracker

Uses only Sections 1-7

7 build steps

2h 18m

Part 2 Leveling Up

08 **Type Hints** — 9 concepts — 38m

→

09 **Error Handling** — 12 concepts — 1h 31m

→

10 **Files, JSON & Paths** — 16 concepts — 1h 15m

→

11 **Standard Library** — 21 concepts — 1h 09m

→

12 **Virtual Envs & Packaging** — 7 steps — 1h 38m

→

13 **Classes & OOP** — 25 concepts — 2h 13m

→

MILESTONE PROJECT 2

CLI Persistent Expense Tracker

Uses all Sections 1-13

11 build steps

3h 29m

© Kalyan Reddy Daida

StackSimplify

Python Keywords: 30 of 35 by the End

Python has only 35 keywords, and you will have written 30 of them yourself by the end of this course. The five in grey (assert, del, yield, async, await) and the four soft keywords are for later courses.

Values
`True` `False` `None`

Logic & Membership
`and` `or` `not` `in` `is`

Conditionals
`if` `elif` `else`

Loops
`for` `while` `break` `continue`

Functions
`def` `return` `lambda`

Classes
`class`

Error Handling
`try` `except` `finally` `raise` `assert`

Modules & Imports
`import` `from` `as`

Scope
`global` `nonlocal`

Context Managers
`with`

Cleanup & Flow
`pass` `del`

Generators & Async
`yield` `async` `await`

Soft Keywords
`match` `case` `_` `type`

30 / 35
KEYWORDS COVERED

`keyword` You write this in the course

`keyword` Not covered here

© Kalyan Reddy Daida

How You Learn Here

The same rhythm in every section, and it ends in a **real project**, not an exercise.

STEP 1 - LEARN

Concept

One small idea, a **demo you run**, and its exact output.

WHO WRITES IT

Kalyan

Every line, typed on screen

295+ concepts

→

STEP 2 - PRACTICE

Practice

A short problem for that concept. **You write the code.**

WHO WRITES IT

You

Solution file to compare

60+ problems

→

STEP 3 - COMBINE

Workshop

A bigger task that **puts the concepts together**.

optional, for more practice

WHO WRITES IT

You

Solution file to compare

60+ problems

→

STEP 4 - BUILD

Project

Milestone 1 builds the app. **Milestone 2** rebuilds it properly.

WHO WRITES IT

Kalyan

Built with you, line by line

2 milestone projects

TRY FIRST, THEN CHECK Every practice and workshop problem ships with a **full solution file**. Attempt it yourself, then compare.

© Kalyan Reddy Daida

Running the Concept Demos

Every concept is a few lines you can run. **Two ways to get them onto your machine.**

Way 1 From the course website

1. Open the section page and find the concept
2. **Copy** the code block, it is one click
3. Paste it into that section's `sNN_lecture.py`
4. **Run it** and compare with the output on the page

WEBSITE BOOKMARK IT

`https://python-for-beginners.stacksimplify.com`

Way 2 From the GitHub repo

1. Clone `python-for-absolute-beginners` once
2. Open the section's `python-learn/` folder
3. Every step is a file: `s05_step01_lists.py`
4. **Run it as it is**, nothing to copy

REPO BOOKMARK IT

`https://github.com/stacksimplify/python-for-absolute-beginners`

WHICH ONE Typing it yourself sticks better, so **Way 1 while you learn**. Keep the repo open as the finished reference, and for the days you just want to run something fast.

© Kalyan Reddy Daida

Practice and Workshop Problems

Same two routes as the demos, but here the point is that **you write the code first**.

Way 1 On the course website

Question The task and its **expected output**

Hint Stuck? Open **Need a hint?** for a nudge, not the answer

Solution Open **Show the solution** only after you have tried

Everything is on one page, so nothing to download to get started.

WEBSITE **BOOKMARK IT**

`https://python-for-beginners.stacksimplify.com`

Way 2 In the GitHub repo

Question file `python-practice/s05_practice01_lists.py`

You write The file holds **the task as comments**, and empty space below

Solution file `solutions/s05_practice01_sol_lists.py`

Workshops work the same way, under `python-workshop/`.

REPO **BOOKMARK IT**

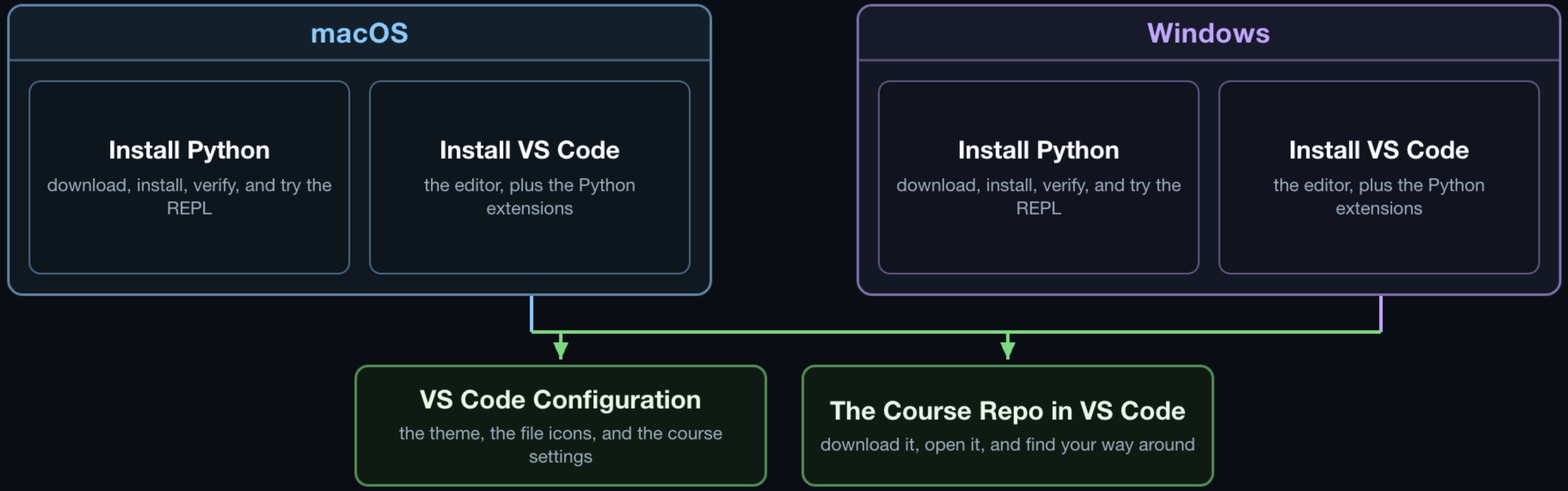
`https://github.com/stacksimplify/python-for-absolute-beginners`

THE ONE RULE The solution is always one click or one folder away, which makes it easy to peek. **Write your own version first, even a wrong one.** That attempt is what makes the solution stick.

© Kalyan Reddy Daida

Section 01: Python Environment Setup

Pick your operating system, install two things, then everyone meets again for the same two steps.



WATCH THE VIDEOS FOR YOUR OWN OS

On **macOS**, follow the macOS install videos. On **Windows**, follow the Windows ones. From **VS Code Configuration** onward, every video is the same for everyone.

© Kalyan Reddy Daida

Section 02: Variables and Data Types Concept Map

The full path through this section - we open here before each concept, then dive in.

32
CONCEPT DEMOS

VARIABLES

01	Your First Program	C1	`print()` `top to bottom` `.py file`
02	Comments	C1 to C4	`#` `inline` `comment out` `"""..."""`
03	print() Options and Escapes	C1 to C7	`many values` `sep` `end` `\n \t` `\\ \"`
04	Variables and Naming	C1 to C8	`name = value` `reassign` `valid names` `snake_case / UPPER_CASE` `case sensitive`
05	Data Types and type()	C1 to C5	`str int float bool` `type()` `.__name__` `round()`
06	Type Conversion	C1 to C4	`int()` `float()` `str()` `input() is str`
07	Keywords	C1 to C3	`import keyword` `35 hard + 4 soft` `kwlist / softkwlist` `iskeyword()`

© Kalyan Reddy Daida

Section 03: Strings and f-strings Concept Map

The full path through this section - we open here before each concept, then dive in.

20
CONCEPT DEMOS

STRINGS

01 — Making Strings (C1 to C5)
- `single/double quotes`
- `join with +`
- `len()`
- `"""multiline"""`

02 — f-strings (C1 to C5)
- `f"{var}"`
- `math in braces`
- `format spec :.2f`
- `align <>^`

03 — String Methods (C1 to C5)
- `.upper()` `.lower()`
- `.strip()` `.replace()`
- `.title()` `.count()` `.isdigit()`
- `.startswith()` `.endswith()` `.find()` `.index()`

04 — Indexing and Slicing (C1 to C5)
- `word[0], word[-1]`
- `slice [start:stop]`
- `step [::2] reverse [::-1]`
- `immutable`

© Kalyan Reddy Daida

Section 04: Operators, Booleans, and Conditionals Concept Map

The full path through this section - we open here before each concept, then dive in.

19
CONCEPT DEMOS

OPERATORS

01 Arithmetic — C1 to C4
`+ - * /` `// % **` `+= -= *=` `abs() / divmod()`

02 Comparison — C1 to C3
`== !=` `> < >= <=` `returns bool` `is / is None`

03 Boolean Logic — C1 to C9
`and / or / not` `in / not in` `truthy / falsy`
`all() / any()` `chained / precedence`

04 Making Decisions — C1 to C3
`if / elif / else` `nested if` `indentation`
`ternary one-liner`

© Kalyan Reddy Daida

Section 05: Collections Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Section 06: Loops and Comprehensions Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Section 07: Functions Concept Map (1 of 2)

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Section 07: Functions Concept Map (2 of 2)

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Milestone Project-1: What You Build

A real program you run from the terminal, built with **dictionaries and functions** and nothing you have not already met in Sections 01 to 07.

ON SCREEN



Six choices, and you write every one of them

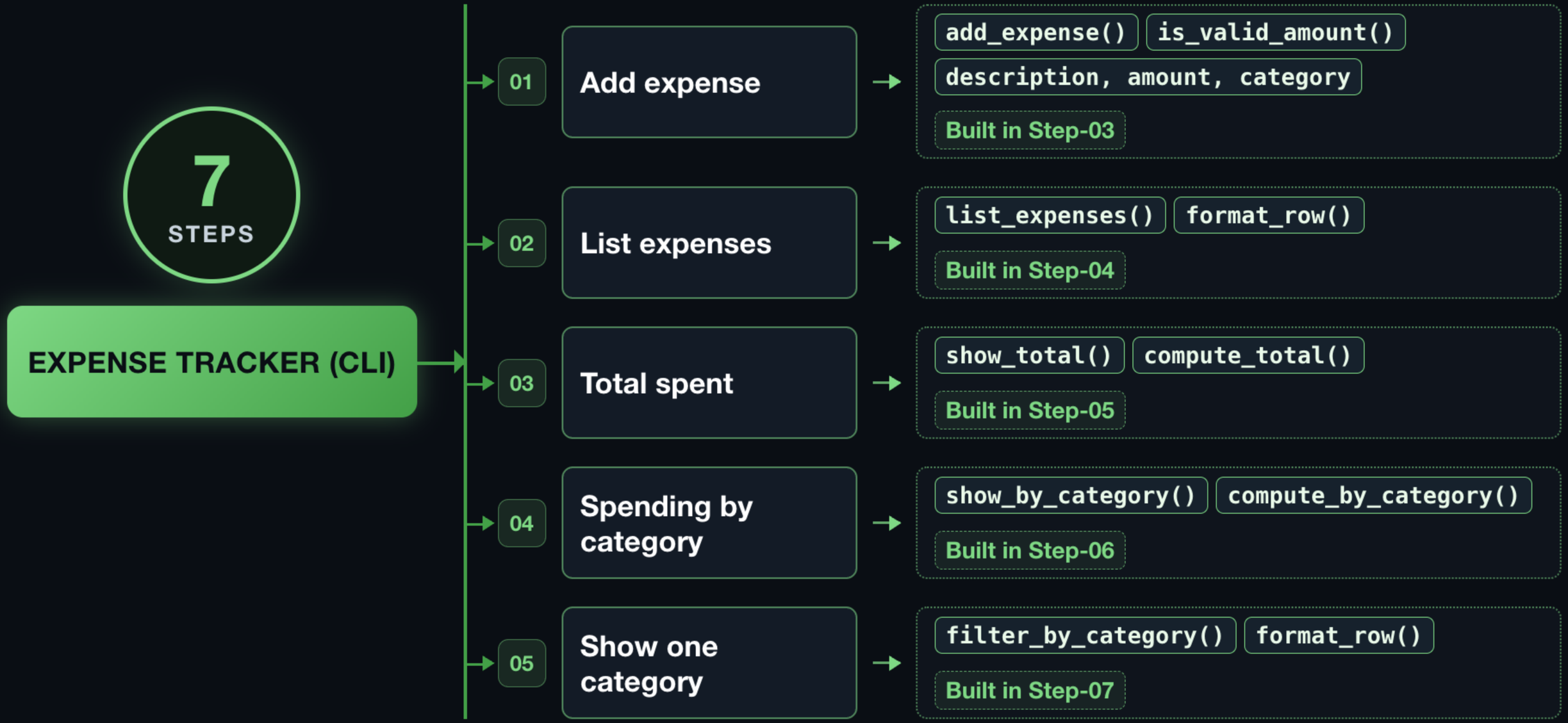
BEHIND IT	WHAT YOU WRITE IN PROJECT 1
One expense	`{ "amount": 3.5 }` a dictionary with three keys
All of them	`[ ... ]` a plain list, held in main()
The five actions	`f5_actions.py` one function per menu choice
The helpers	`f1 · f2 · f3 · f4` validate, format a row, total, totals by category

PROJECT 1 After **Section 07**, in **7 build steps**

© Kalyan Reddy Daida

Milestone Project-1: CLI Expense Tracker - What You Will Build?

The five menu actions, and the functions that make each one work.



© Kalyan Reddy Daida

Section 08: Type Hints Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Section 09: Error Handling and Debugging Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Section 10: Files, JSON, and Paths Concept Map

The full path through this section - we open here before each concept, then dive in.

16
CONCEPT DEMOS

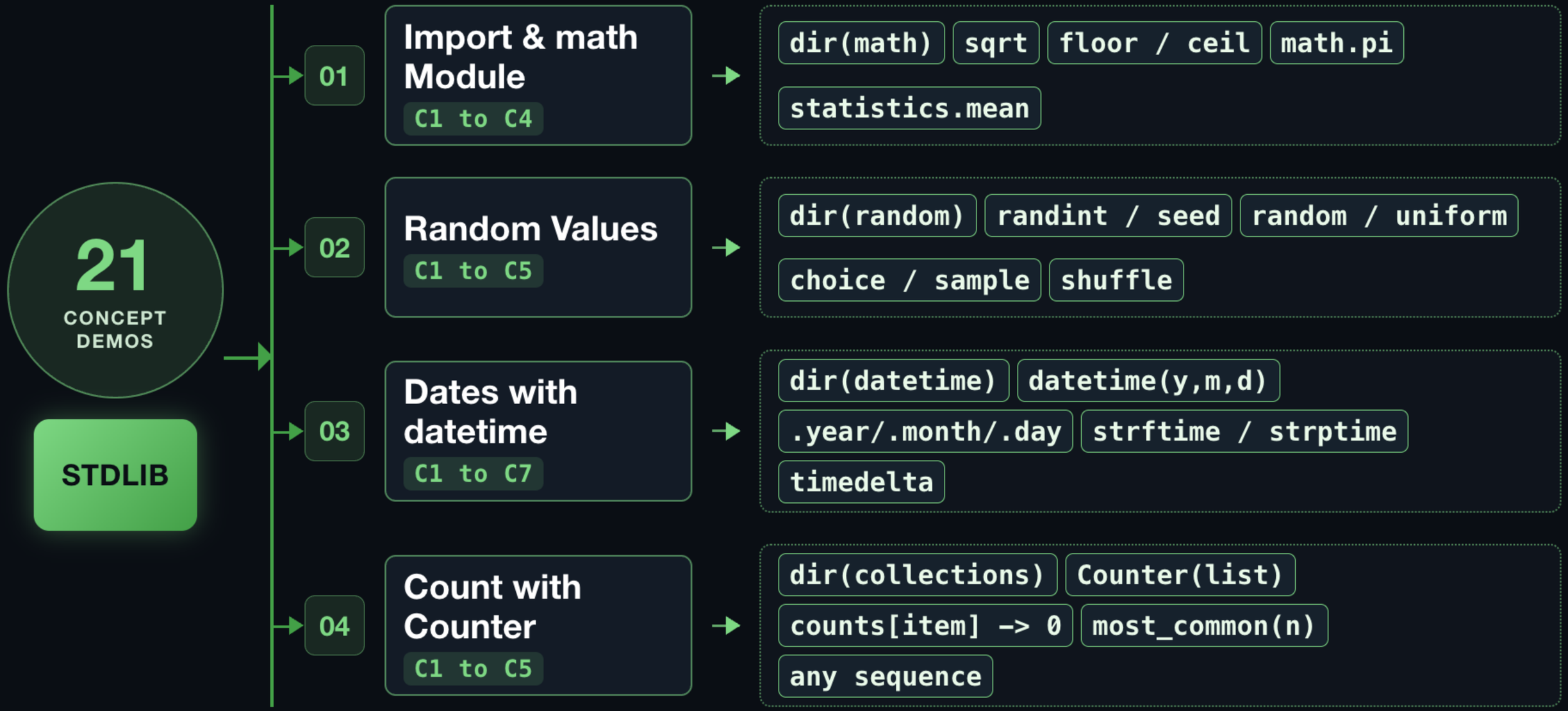
FILES

01	Write and Read a Text File	C1 to C3	`with open()` `mode "w" / "r"` `f.read()` `\n newlines` `encoding="utf-8"`
02	Append and Read Lines	C1 to C4	`mode "a"` `for line in f` `.strip()` `readlines()` `seek(0)`
03	JSON Save and Load	C1 to C3	`json.dump` `json.load` `indent=2` `dict to file`
04	Paths with pathlib	C1 to C4	`Path() / join` `.name / .suffix` `.stem / .parent` `.exists()` `read_text() / write_text()`
05	Save Next to Your Script	C1 and C2	`__file__` `.parent` `mkdir()` `rmdir()`

© Kalyan Reddy Daida

Section 11: Standard Library Essentials Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Section 12: Virtual Environments and Packaging Concept Map

The path through Section 12: the habits that turn loose scripts into an isolated, installable project.



© Kalyan Reddy Daida

Section 13: Classes and OOP Basics Concept Map

The full path through this section: we open here before each concept, then dive in.

25
CONCEPT DEMOS

CLASSES

01	Why Objects, First Class C1 to C4	loose variables	class blueprint	p1 = Product()	0x... address
02	__init__, self, Attributes C1 to C4	__init__ auto-runs	self IS the object	parameters	self.name
03	Class vs Instance Attrs C1 to C4	shared class attr	Product.store_name	per-object data	instance wins
04	Methods C1 to C4	def inside class	method changes data	return vs act	missing () gotcha
05	Readable Objects (__str__, __repr__) C1 to C6	address is back	plain method ignored	repr default too	__str__ + __repr__
06	Class & Static Methods C1 to C3	@classmethod (cls)	from_dict	@staticmethod	is_valid_price

© Kalyan Reddy Daida

Milestone Projects 1 and 2: What You Build

One app, built twice. Project 1 with dictionaries and functions. Project 2 **rebuilt with classes**, with its data saved to a file.

ON SCREEN

Expense Tracker

```
==== Expense Tracker 2.0 ====
1) Add expense
2) List expenses
3) Total spent
4) Spending by category
5) Show one category
6) Quit   Save and quit in Project 2
Choose 1-6: _
```

Green marks what changes in **Project 2**

BEHIND IT	PROJECT 1		PROJECT 2
One expense	`{ "amount": 3.5 }` a dictionary, three keys	→	`class Expense` one object, four attributes
All of them	`[ ... ]` a plain list in main()	→	`class ExpenseBook` holds the list and the methods
Choice 6	**Quit** just exits	→	**Save and quit** writes the file first
After you quit	**gone** it lived in memory	→	`data/expenses.json` still there next run

PROJECT 1 After **Section 07**, in **7 build steps**

PROJECT 2 After **Section 13**, in **11 build steps**

© Kalyan Reddy Daida

Milestone Project-2: Upgrade CLI Expense Tracker to Persistent Expense Tracker

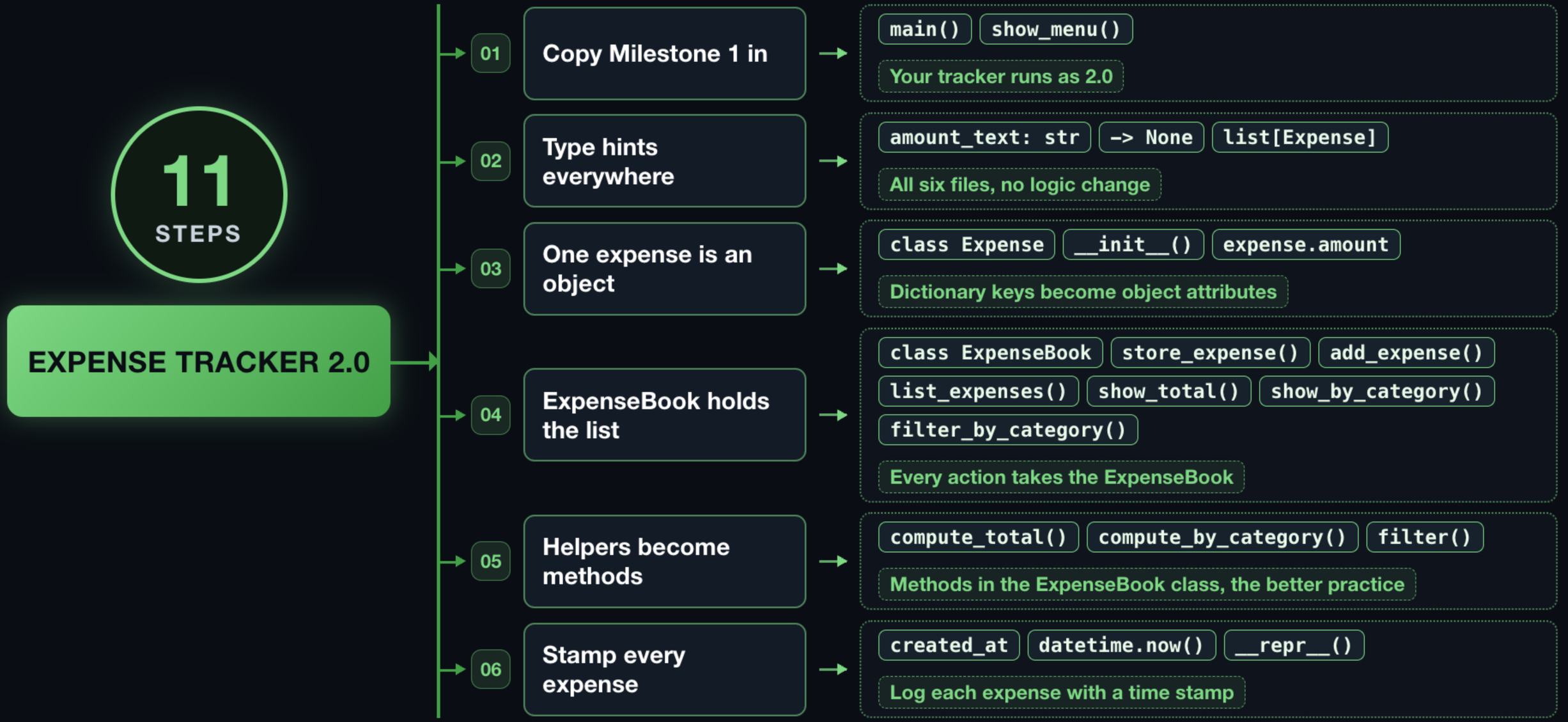
The same Expense Tracker and the same menu. Milestone 2 changes how it works inside, using Sections 08 to 13.

	MILESTONE 1: DICTIONARY-BASED	UPGRADE	MILESTONE 2: OBJECTS + JSON + ERROR HANDLING
One expense	a dictionary with three keys	Section 13 →	an **`Expense`** object with attributes
Who owns the list	**`main()`** holds **`expenses = []`**	Section 13 →	**`ExpenseBook`** owns it: **`book = ExpenseBook(...)`**
Read a value	**`expense["amount"]`**	Section 13 →	**`expense.amount`**
Total spent	**`compute_total(expenses)`**	Section 13 →	**`book.compute_total()`**
By category	**`compute_by_category(expenses)`**	Section 13 →	**`book.compute_by_category()`**
One category	a list comprehension inside **`filter_by_category()`**	Section 13 →	**`book.filter()`**, called by the same action
Types	*no type hints*	Section 08 →	**`amount_text: str`**, **`-> bool`**, **`list[Expense]`**
Quit and restart	everything is **gone**	Section 10 →	**`book.save()`** and **`book.load()`**: **`data/expenses.json`**
A broken file	*no file, so nothing to break*	Section 09 →	**`try`** / **`except`**: a warning, **no crash**
New answers	*not there yet*	Section 11 →	**`created_at`** time stamp on every expense
One package	*nothing but Python*	Section 12 →	**`.venv`** and **`pip install rich`**: a real table *(bonus)*

© Kalyan Reddy Daida

Milestone Project-2: CLI Persistent Expense Tracker - What You Will Build? 1 of 2

Eleven steps build it. Steps 01 to 06 turn your Milestone 1 code into objects and a book.



© Kalyan Reddy Daida

Milestone Project-2: CLI Persistent Expense Tracker - What You Will Build? 2 of 2

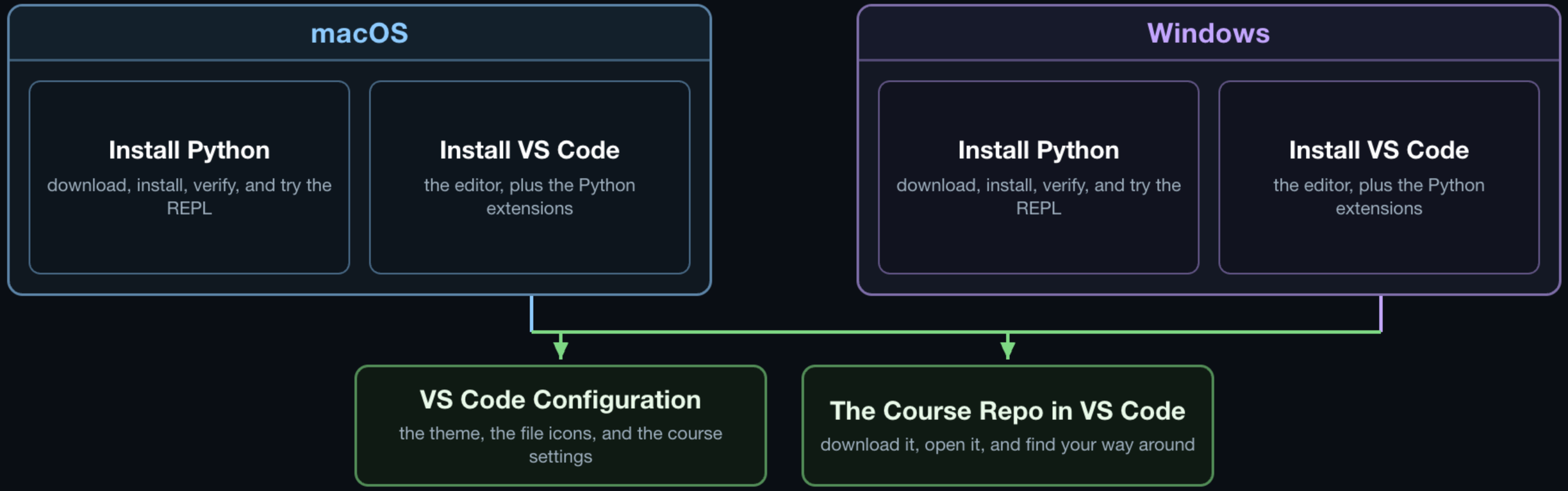
Steps 07 to 11 give it a file, a guard, a full run, and the optional rich bonus.



© Kalyan Reddy Daida

Section 01: Python Environment Setup

Pick your operating system, install two things, then everyone meets again for the same two steps.



WATCH THE VIDEOS FOR YOUR OWN OS

On **macOS**, follow the macOS install videos. On **Windows**, follow the Windows ones. From **VS Code Configuration** onward, every video is the same for everyone.

© Kalyan Reddy Daida

Setup: Install Python and the REPL

The same three steps on either system. Only the command name changes.

macOS Linux is the same

1. **Download** from python.org/downloads, the official site.
2. **Install** it: double-click the package and follow the installer.
3. **Verify**, then start the shell. macOS ships an older Python, so always say `python3`.

```
$ python3 --version
Python 3.14.6
$ python3
>>> 2 + 2
4
```

Windows Windows Terminal

1. **Download** from python.org/downloads, the official site.
2. **Install** it, and on the FIRST installer screen:

Tick "Add python.exe to PATH" or Python will not run

3. **Verify**, then start the shell. On Windows the command is `python`, with no 3.

```
> python --version
Python 3.14.6
> python
>>> 2 + 2
4
```

That `>>>` is the REPL: it runs one line at a time and answers straight away.

Setup: What We Will Do in VS Code

Three things, in this order. Then you are ready to write Python.

1 Install VS Code

Download it, install it, and open it until the **Welcome** tab appears.

- code.visualstudio.com
- the .dmg, or the Windows installer
- the Welcome tab

This step differs by operating system

2 Add four extensions

Python support, a fast linter, a dark theme, and clear file icons.

- Python
- Ruff
- One Dark Pro
- Material Icon Theme

Same on every operating system

3 Change three settings

Roomier folder indents, line wrapping, and **auto save**, so you never lose a change.

- `tree indent`
- `word wrap`
- `auto save`

Same settings, only the keys differ

Do this once now, and the editor stays set up for the whole course.

© Kalyan Reddy Daida

Setup: Install VS Code

The editor you will write every line of this course in. Install it once.

macOS Linux is similar

1. **Download** from code.visualstudio.com.
2. **Open the .dmg** and drag Visual Studio Code into **Applications**.
3. **Launch it**: press `Cmd + Space`, type Visual Studio Code, press Enter.

You are done when the Welcome tab opens

Windows the installer does more

1. **Download** from code.visualstudio.com.
2. **Run the installer** and tick **Add to PATH** and both **Open with Code** options.
3. **Launch it** from the **Start menu**.

You are done when the Welcome tab opens

© Kalyan Reddy Daida

Setup: Install the VS Code Extensions

Extensions icon in the left sidebar, search the name, click Install. Check the publisher.

INSTALL THESE FOUR YOURSELF

Python Microsoft
Run and debug code, pick your interpreter, autocomplete. The must-have one.

Ruff Astral Software
A very fast linter. It underlines mistakes as you type.

One Dark Pro binaryify
A clean dark colour theme that is comfortable to read.

Material Icon Theme Philipp Kief
Coloured icons per file type, so you spot a .py at a glance.

PYTHON BRINGS THESE ALONG

Pylance
The engine behind autocomplete, type hints and go to definition.

Python Debugger
Set breakpoints and step through your code to find bugs.

Python Environments
Create and manage virtual environments and packages from the UI.

You do not install these three. They arrive with the Python extension.

COMMAND PALETTE macOS `Cmd + Shift + P` Windows `Ctrl + Shift + P` How you run any command in VS Code

THEN TURN THEM ON Installing does not switch them on. Open the Command Palette above, then pick **Preferences: Color Theme** and **Preferences: File Icon Theme**.

© Kalyan Reddy Daida

Setup: Update Important VS Code Settings

Only the key to open Settings differs. The three changes are the same everywhere.

macOS

Linux is similar

`Cmd + ,` open Settings

Windows

`Ctrl + ,` open Settings

THEN TYPE EACH ONE IN THE SEARCH BOX AND SET THE VALUE

SEARCH FOR	SET IT TO	WHY
`tree indent`	`20`	More space between nested folders, so the project is easy to read.
`word wrap`	`on`	Long lines wrap instead of scrolling off the right edge.
`auto save`	`afterDelay`	**Off by default, so this one really matters.** Saves your file one second after you stop typing.

© Kalyan Reddy Daida

Setup: Get the Course Code Repo

Download it once. From section 02 on, every section folder looks the same inside.

COURSE CODE REPO `https://github.com/stacksimplify/python-for-absolute-beginners` **BOOKMARK IT**

1 Download
Open the link above, click the green **Code** button, then **Download ZIP**.

2 Unzip
Put it in **Documents**, not Downloads.
macOS double-click · Windows Extract All

3 Rename
Drop the **-main** branch suffix. See below.

4 Open
In VS Code use **File** then **Open Folder**. Open the **folder**, never a single file.

RENAME ~~`python-for-absolute-beginners-main`~~ → `python-for-absolute-beginners` -main is the branch name, not part of the project

WHAT YOUR EXPLORER SHOULD SHOW

```
python-for-absolute-beginners/
  01-Python-Environment-Setup   README only, no code
  02-Variables-and-Data-Types
  ... one folder per section ...
  13-Classes-and-OOP-Basics
  milestone-projects   the projects you build
  pdfs   section slides to keep
  images   you will not open this
```

INSIDE EVERY SECTION FROM 02 ON

```
05-Collections/
  python-learn   concept demos
    s05_lecture.py   you type along here
  python-practice
    solutions
  python-workshop
    solutions
```

12 is command line work, so it has no python-workshop

Leave this folder open in VS Code for the whole course. Every section you start is a folder you already have.

© Kalyan Reddy Daida

Section 02: Variables and Data Types Concept Map

The full path through this section - we open here before each concept, then dive in.

32
CONCEPT DEMOS

VARIABLES

01	Your First Program	C1	`print()` `top to bottom` `.py file`
02	Comments	C1 to C4	`#` `inline` `comment out` `"""..."""`
03	print() Options and Escapes	C1 to C7	`many values` `sep` `end` `\n \t` `\\ \"`
04	Variables and Naming	C1 to C8	`name = value` `reassign` `valid names` `snake_case / UPPER_CASE` `case sensitive`
05	Data Types and type()	C1 to C5	`str int float bool` `type()` `.__name__` `round()`
06	Type Conversion	C1 to C4	`int()` `float()` `str()` `input() is str`
07	Keywords	C1 to C3	`import keyword` `35 hard + 4 soft` `kwlist / softkwlist` `iskeyword()`

© Kalyan Reddy Daida

Variables: Keywords Covered

Section 02: Variables and Data Types - the keywords in bright green are the new ones we cover here.

Values
`True` `False` `None`

Logic & Membership
`and` `or` `not` `in` `is`

Conditionals
`if` `elif` `else`

Loops
`for` `while` `break` `continue`

Functions
`def` `return` `lambda`

Classes
`class`

Error Handling
`try` `except` `finally` `raise` `assert`

Modules & Imports
`import` `from` `as`

Scope
`global` `nonlocal`

Context Managers
`with`

Cleanup & Flow
`pass` `del`

Generators & Async
`yield` `async` `await`

Soft Keywords
`match` `case` `_` `type`

`keyword` New in this section `keyword` Covered earlier `keyword` Not covered yet

© Kalyan Reddy Daida

Variables: First Program and Comments

Write your first line of Python with **print()**, and use **comments** - notes that Python ignores - to explain your code.

#	CONCEPT	SAMPLE CODE
1	**Your first program** `print()` print() writes text out to the screen.	`print("I am learning Python today")`
2	**Comment line** `#` A line that starts with # is ignored by Python.	`# explain your code in plain words`
3	**Inline comment** Add a quick note at the end of a code line.	`print("Hi")  # a note for people`
4	**Comment out a line** Put # in front to switch a line off without deleting it.	`# print("this line will not run")`
5	**Multi-line comment** `"""` Triple quotes hold a note that spans several lines.	`"""a longer note over many lines"""`

© Kalyan Reddy Daida

Variables: The print() Function

print() shows values on the screen - your first tool for seeing results and finding bugs. In this step you learn everything one print() call can do.

#	CONCEPT	SAMPLE CODE
1	**Print several values** Commas join every value with one space.	`print("Python", "is", "Fun")`
2	**Custom separator** `sep=` Change what goes between the values.	`print("2026", "01", "01", sep="-")`
3	**Custom line ending** `end=` Default ending is a new line; change it here.	`print("same", end=" ")`
4	**Blank line** `print()` No arguments prints an empty line.	`print()`
5	**New line** `\n` Breaks one string onto the next line.	`print("Line One\nLine Two")`
6	**Tab** `\t` Inserts a tab to line up columns.	`print("Name:\tKalyan")`
7	**Backslash and quotes** `\\` `\"` Escape a backslash or a quote to print it.	`print("C:\\Users\\Kalyan")`

© Kalyan Reddy Daida

Variables: Creating and Naming (1 of 2)

A variable is a name attached to a value with =, so you can store data once and reuse it throughout your program.

#	CONCEPT	SAMPLE CODE
1	**Create a variable** `=` Attach a value to a name using =.	`name = "Kalyan"`
2	**Show a variable** `print()` print() displays the value a name holds.	`print(name)`
3	**Text with a variable** Show a label and its value in one line.	`print("age is", age)`
4	**Reassign a variable** `=` Assign again with = to update the value.	`age = 31`

© Kalyan Reddy Daida

Variables: Creating and Naming (2 of 2)

A variable is a name attached to a value with =, so you can store data once and reuse it throughout your program.

#	CONCEPT	SAMPLE CODE
5	**Legal name rules** Start with a letter or _, then letters, digits, or _.	`first_name = "Kalyan"`
6	**snake_case style** `snake_case` Lowercase words joined by underscores.	`favorite_color = "blue"`
7	**Constants** `UPPER_CASE` UPPER_CASE marks a value that never changes.	`MAX_SCORE = 100`
8	**Case sensitive names** age and Age are two different variables.	`print("age is", age, "and Age is", Age)`

© Kalyan Reddy Daida

Variables: Data Types and type()

Every value has a data type - str, int, float, or bool - and **type()** tells you which one a value is.

#	CONCEPT	SAMPLE CODE
1	**Four core data types** Text is str, whole number int, decimal float, True/False bool.	`title = "Python Basics"`
2	**Check a value's type** `type()` type(value) returns the data type of a value.	`print(type(lessons))`
3	**Short type name** `.__name__` Get the plain name like str, not the full class text.	`print(type(rating).__name__)`
4	**Boolean values** `True` `False` Only True or False, each with a capital first letter.	`is_member = True`
5	**Round a decimal** `round()` round(value, digits) shortens a long decimal number.	`print(round(pi, 2))`

© Kalyan Reddy Daida

Variables: Type Conversion

Type conversion changes a value from one type to another - text into numbers with int() and float(), numbers into text with str() - so your data is the right type for the job.

#	CONCEPT	SAMPLE CODE
1	**Text to integer** `int()` Turn typed text like "25" into a whole number.	`age = int(age_text)`
2	**Text to decimal** `float()` Turn text like "19.99" into a decimal number.	`price = float(price_text)`
3	**Number to text** `str()` Join a number into a text sentence without an error.	`print("I have " + str(count) + " items")`
4	**Input is always text** `input()` input() hands back a str, so a + b joins them.	`a = input("Enter value for a: ")`
5	**Convert then compute** `int(input())` Wrap input() in int() first to do real math.	`a = int(input("Enter value for a: "))`

© Kalyan Reddy Daida

Variables: Python Keywords

Keywords are words Python reserves for its own grammar, so you cannot use them as variable names.

#	CONCEPT	SAMPLE CODE
1	**The reserved word list** `keyword.kwlist` The keyword module holds Python's full list of keywords.	`print(keyword.kwlist)`
2	**A keyword as a name fails** Naming a variable after a keyword is a SyntaxError.	`class = "Math"`
3	**Test a word** `iskeyword()` Check whether a word is a keyword before you use it.	`keyword.iskeyword("for")`

© Kalyan Reddy Daida

Section 03: Strings and f-strings Concept Map

The full path through this section - we open here before each concept, then dive in.

20
CONCEPT DEMOS

STRINGS

01 — Making Strings (C1 to C5)
- `single/double quotes`
- `join with +`
- `len()`
- `"""multiline"""`

02 — f-strings (C1 to C5)
- `f"{var}"`
- `math in braces`
- `format spec :.2f`
- `align <>^`

03 — String Methods (C1 to C5)
- `.upper()` `.lower()`
- `.strip()` `.replace()`
- `.title()` `.count()` `.isdigit()`
- `.startswith()` `.endswith()` `.find()` `.index()`

04 — Indexing and Slicing (C1 to C5)
- `word[0], word[-1]`
- `slice [start:stop]`
- `step [::2] reverse [::-1]`
- `immutable`

© Kalyan Reddy Daida

Strings: Making Strings

A **string** is a piece of text wrapped in quotes; you can use single, double, or triple quotes, join strings with +, and count characters with len().

#	CONCEPT	SAMPLE CODE
1	**Text in quotes** `" "` `' '` Single or double quotes both make a string.	`first_name = "Kalyan"`
2	**Quote inside the text** Use the other quote style so inner quotes work.	`sentence = "It's a sunny day"`
3	**Join strings** `+` The + operator joins strings (concatenation).	`full_name = "Kalyan" + " " + "Reddy"`
4	**Count characters** `len()` len() gives how many characters are in a string.	`print(len("Python"))`
5	**Multiline string** `"""..."""` Triple quotes keep text across several lines.	`bio = """Kalyan Reddy ... Hyderabad"""`

© Kalyan Reddy Daida

Strings: Building Text with f-strings

An **f-string** has the letter **f** before the quotes and drops variables, expressions, and formatting straight into the text with **{braces}**.

#	CONCEPT	SAMPLE CODE
1	**Drop values into text** `f"..."` Put f before the quotes, then use {variable}.	`print(f"My name is {name} and I am {age}")`
2	**Math inside braces** `{ }` Run an expression right inside the braces.	`print(f"Next year I will be {age + 1}")`
3	**Fixed decimals** `:.Nf` A colon plus .Nf sets the number of decimal places.	`f"{price:.2f}"`
4	**Thousands separator** `:,.2f` A comma groups the digits into thousands.	`f"{price:,.2f}"`
5	**Percentage format** `:.N%` A percent sign scales the value and adds the % sign.	`f"{ratio:.1%}"`

© Kalyan Reddy Daida

Strings: f-string Number Formatting

After a colon inside the braces, a format spec controls how a number looks - decimals, commas, percent.

Decimal places

`:.Nf` rounds to N decimals: `.3f`, `.2f`, `.1f`, `.0f`.

Thousands

`:,` groups thousands, like `1,234.43`.

Percent

`:.N%` shows a fraction as a percent, like `87.5%`.

FORMAT SPECS

```
price = 1234.4321

f"{price:.2f}"    # 1234.43

f"{price:.0f}"    # 1234

f"{price:,.2f}"   # 1,234.43

ratio = 0.875432

f"{ratio:.1%}"    # 87.5%
```

A format spec changes how the number LOOKS (and gives back text) - the value itself is unchanged.

© Kalyan Reddy Daida

Strings: String Methods

A **method** is an action you call on a string with a dot, like **text.upper()**, and it returns a new string.

#	CONCEPT	SAMPLE CODE
1	**Trim and change case** `.strip()` `.upper()` `.replace()` Remove edge spaces, switch case, or swap text.	`print(name.strip().upper())`
2	**Title case** `.title()` Capitalize the first letter of every word.	`print("kalyan reddy".title())`
3	**Count matches** `.count()` Count how many times a piece of text appears.	`print("kalyan reddy".count("a"))`
4	**Digits-only check** `.isdigit()` True only when every character is a digit.	`print("30".isdigit())`
5	**Check ends, locate text** `.startswith()` `.endswith()` `.find()` Test the start or end, or find where text sits.	`print(email.startswith("kalyan"))`

© Kalyan Reddy Daida

Strings: Indexing and Slicing

Indexing reads one character by its position, and **slicing** reads a range of characters using square brackets.

#	CONCEPT	SAMPLE CODE
1	**Index one character** `[i]` Each character has a position, starting at 0.	`print(word[0])`
2	**Slice a range** `[start:stop]` start is included, stop is not.	`print(word[0:3])`
3	**Slice with a step** `[::step]` Step over characters; a step of -1 reverses.	`print(word[::-1])`
4	**Negative index** `[-N:]` Count from the end; take the last N characters.	`print(plate[-4:])`
5	**Strings are immutable** Read a character, but you cannot change one in place.	`print("J" + word[1:])`

© Kalyan Reddy Daida

Section 04: Operators, Booleans, and Conditionals Concept Map

The full path through this section - we open here before each concept, then dive in.

19
CONCEPT DEMOS

OPERATORS

01 Arithmetic — C1 to C4
`+ - * /` `// % **` `+= -= *=` `abs() / divmod()`

02 Comparison — C1 to C3
`== !=` `> < >= <=` `returns bool` `is / is None`

03 Boolean Logic — C1 to C9
`and / or / not` `in / not in` `truthy / falsy`
`all() / any()` `chained / precedence`

04 Making Decisions — C1 to C3
`if / elif / else` `nested if` `indentation`
`ternary one-liner`

© Kalyan Reddy Daida

Operators: Keywords Covered

Section 04: Operators, Booleans and Conditionals - the keywords in bright green are the new ones we cover here.

Values
`True` `False` `None`

Logic & Membership
`and` `or` `not` `in` `is`

Conditionals
`if` `elif` `else`

Loops
`for` `while` `break` `continue`

Functions
`def` `return` `lambda`

Classes
`class`

Error Handling
`try` `except` `finally` `raise` `assert`

Modules & Imports
`import` `from` `as`

Scope
`global` `nonlocal`

Context Managers
`with`

Cleanup & Flow
`pass` `del`

Generators & Async
`yield` `async` `await`

Soft Keywords
`match` `case` `_` `type`

`keyword` New in this section `keyword` Covered earlier `keyword` Not covered yet

© Kalyan Reddy Daida

Operators: Arithmetic Operators

Arithmetic operators are the symbols Python uses to do math on numbers - from add, subtract, multiply, and divide to floor division, modulo, and power.

#	CONCEPT	SAMPLE CODE
1	**Everyday math operators** `+` `-` `*` `/` Add, subtract, multiply, and divide two numbers.	`print(a + b)  # 22: addition`
2	**Three more operators** `//` `%` `**` The whole-number quotient, the remainder, and powers.	`print(a // b)  # 3: quotient`
3	**Number helpers** `abs()` `divmod()` Size without a sign; quotient and remainder together.	`print(divmod(17, 5))  # (3, 2)`
4	**Augmented assignment** `+=` `-=` `*=` Update a variable using its own value, in shorthand.	`score += 50  # score = score + 50`

© Kalyan Reddy Daida

Operators: Comparison Operators

A comparison operator tests how two values relate and answers with a Boolean - **True** or **False**.

#	CONCEPT	SAMPLE CODE
1	**Equal and not equal** `==` `!=` Check if two values are equal or not equal.	`print(age == 20)`
2	**Ordering comparisons** `>` `<` `>=` `<=` Ask which value is greater or smaller.	`print(age > 18)`
3	**Compare with None** `is` `is not` Test for None with is, never with ==.	`print(result is None)`

© Kalyan Reddy Daida

Operators: Boolean Logic (1 of 2)

Boolean logic combines True and False values with operators like and, or, and not so a program can make decisions.

#	CONCEPT	SAMPLE CODE
1	**Logical AND** `and` True only when both sides are true.	`print(is_member and age >= 18)`
2	**Logical OR** `or` True when at least one side is true.	`print(is_member or age < 18)`
3	**Logical NOT** `not` Flips a True/False value to its opposite.	`print(not is_member)`
4	**Membership test** `in` Tests if a value sits inside text or a collection.	`print("a" in "Kalyan")`
5	**Truthy and falsy** `bool()` Empty values are false; everything else is true.	`print(bool(0), bool(""))`

© Kalyan Reddy Daida

Operators: Boolean Logic (2 of 2)

Boolean logic combines True and False values with operators like and, or, and not so a program can make decisions.

#	CONCEPT	SAMPLE CODE
6	**Every / any item** `all()` `any()` all() needs every item true; any() needs just one.	`print(all(checks))`
7	**Negated membership** `not in` True when a value is NOT inside.	`print("z" not in "Kalyan")`
8	**Chained comparison** Test a whole range in one line, like a score 0 to 100.	`print(0 <= score <= 100)`
9	**Operator precedence** Math first, then comparisons, then and / or.	`print(2 + 3 == 5)`

© Kalyan Reddy Daida

Operators: Truthy and Falsy

Empty things count as False; anything with content counts as True. This is how if treats any value.

Falsy

`0`, `""`, `[]`, `None` act as False.

Truthy

Any number (not 0) or non-empty text acts as True.

Why it helps

`if name:` means "if name is not empty".

TRUTHY / FALSY

```
bool(0)      # False
bool("")     # False
bool(5)      # True
bool("hi")   # True
```

Watch out: the text '0' is truthy - only the number 0 is falsy.

© Kalyan Reddy Daida

Operators: if / elif / else

if / elif / else checks conditions from top to bottom and runs the block under the first one that is True; **else** handles every remaining case.

#	CONCEPT	SAMPLE CODE
1	**if / elif / else chain** Turn a student's marks into a letter grade.	`if marks >= 90:`
2	**Ternary one-liner** `if..else` Choose one of two values in a single line.	`label = "adult" if age >= 18 else "minor"`
3	**Nested if** Check an inner condition only after the outer one is True.	`    if is_admin:`

© Kalyan Reddy Daida

Conditionals: One-Line if/else

When you just pick between two values, write it on one line: value_if_true if condition else value_if_false.

Compact

A whole if/else choice on a single line.

Reads left to right

`A if test else B`.

Good for

Picking one of two values to store or print.

TERNARY

```
age = 20

tag = "adult" if age >= 18 else "kid"

print(tag)   # adult
```

Great for a simple pick - use a real if when it gets complex.

Section 05: Collections Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Collections: Lists - Ordered and Changeable (1 of 3)

A list is an ordered, changeable collection that holds many values in one place, written inside square brackets.

#	CONCEPT	SAMPLE CODE
1	**Create a list** `[ ]` Hold many values in order, in square brackets.	`names = ["Sam", "Tom", "Ben"]`
2	**Access by index** `[i]` Read an item by position; 0 is first, -1 is last.	`print(names[0])`
3	**Change a list** Add, replace, or remove items in place.	`names.append("Joe")`
4	**Sort in place** `.sort()` Arrange items from low to high, in place.	`scores.sort()`
5	**String to list** `.split()` Cut a string into a list at a separator.	`date_text.split("-")`
6	**List to string** `.join()` Join list items into one string with a separator.	`" ".join(words)`
7	**Insert at index** `.insert()` Put an item at a position, shifting the rest right.	`names.insert(1, "Amy")`

© Kalyan Reddy Daida

Collections: Lists - Ordered and Changeable (2 of 3)

A list is an ordered, changeable collection that holds many values in one place, written inside square brackets.

#	CONCEPT	SAMPLE CODE
8	**Add many items** `.extend()` Add every item from another list at once.	`names.extend(["Ben", "Joe"])`
9	**Remove and return** `.pop()` Remove an item and get it back.	`remove_ben = names.pop()`
10	**Empty the list** `.clear()` Remove everything, leaving an empty list.	`names.clear()`
11	**Reverse order** `.reverse()` Flip the list order in place.	`names.reverse()`
12	**Find position** `.index()` Get the index where a value sits.	`names.index("Tom")`
13	**Count a value** `.count()` Count how many times a value appears.	`letters.count("a")`

© Kalyan Reddy Daida

Collections: Lists - Ordered and Changeable (3 of 3)

A list is an ordered, changeable collection that holds many values in one place, written inside square brackets.

#	CONCEPT	SAMPLE CODE
14	**Copy a list** `.copy()` Make a separate list; a plain = only aliases.	`separate = original.copy()`
15	**Index and slice** `[a:b:c]` start:stop:step, exactly like slicing strings.	`print(nums[1:3])`
16	**Membership test** `in` Check whether a value is in the list.	`"Sam" in ["Sam", "Tom"]`
17	**Not in test** `not in` Check whether a value is not in the list.	`"Zoe" not in ["Sam", "Tom"]`
18	**Nested lists** `[i][j]` A list inside a list; two indexes reach inside.	`print(grid[0][2])`
19	**Nested with strings** Group lists together and pick one item.	`print(groceries[1][2])`

© Kalyan Reddy Daida

Collections: Split and Join

The bridge between text and lists: split() turns a string into a list, join() turns a list back into a string.

split()

A string becomes a list, cut at the separator: `"a-b".split("-")`.

join()

A list becomes a string, glued by a separator: `" ".join(words)`.

Opposites

`split()` goes text to list; `join()` goes list back to text.

SPLIT AND JOIN

```
"2026-06-17".split("-")
# ['2026', '06', '17']


" ".join(["Python", "is", "fun"])
# Python is fun
```

No argument? split() cuts on any whitespace.

© Kalyan Reddy Daida

Collections: Tuples - Fixed and Unpackable (1 of 2)

A tuple is an ordered group of values written in round brackets that cannot change after you create it.

#	CONCEPT	SAMPLE CODE
1	**Create a tuple** `()` Group values that belong together in round brackets.	`player = ("Sam", 25)`
2	**Immutable, cannot change** Changing a tuple item is an error, so it stays safe.	`player[0] = "Tom"  # TypeError`
3	**Two methods** `.count()` `.index()` count() how many times a value appears; index() where.	`print(scores.count(20))`
4	**Slice a tuple** `[a:b:c]` start:stop:step; a slice always gives a tuple.	`print(scores[0:3])`
5	**Membership test** `in` `not in` Check whether a value is in the tuple (read-only).	`print(20 in scores)`

© Kalyan Reddy Daida

Collections: Tuples - Fixed and Unpackable (2 of 2)

A tuple is an ordered group of values written in round brackets that cannot change after you create it.

#	CONCEPT	SAMPLE CODE
6	**Commas make it `,`** Commas build the tuple; brackets are optional.	`player = "Sam", 25`
7	**Unpack a tuple** Give each item its own name in one line.	`name, age, city = player`
8	**Swap in one line** Swap two variables, then save the result as a new tuple.	`a, b = b, a`
9	**Mutable list inside** Slots are fixed, but a list inside a slot can still change.	`player[1].append(100)`

© Kalyan Reddy Daida

Collections: Dictionaries (1 of 2)

A **dictionary** stores each value under a named **key**, so you look things up by a label instead of a position.

#	CONCEPT	SAMPLE CODE
1	**Create a dictionary** `{}` Keys label each value instead of a position.	`student = {"name": "Sam", "age": 20}`
2	**Look up by key** `[key]` Square brackets read the value for a key.	`print(student["name"])`
3	**Safe lookup** `.get()` Returns None or a default, never a crash.	`print(student.get("email", "n/a"))`
4	**Add or update a pair** Assign to a key to add it or replace it.	`student["email"] = "sam@example.com"`
5	**Set many pairs** `.update()` Merge several pairs from another dict at once.	`student.update({"age": 21, "city": "Mumbai"})`
6	**Build from empty** `{}` Start with {} and add pairs one by one.	`student = {}`
7	**Count the pairs** `len()` len() returns how many pairs are stored.	`print(len(student))`

© Kalyan Reddy Daida

Collections: Dictionaries (2 of 2)

A **dictionary** stores each value under a named **key**, so you look things up by a label instead of a position.

#	CONCEPT	SAMPLE CODE
8	**All keys and values** `.keys()` `.values()` See every label or every value at once.	`print(list(student.keys()))`
9	**All key-value pairs** `.items()` Get each pair as a (key, value) tuple.	`print(list(student.items()))`
10	**Is a key present?** `in` in and not in check keys, not values.	`print("name" in student)`
11	**Remove a key** `del` `.pop()` del drops a pair; .pop() returns its value.	`old_age = student.pop("age")`
12	**Value is a list** A value can be a list you index or grow.	`student["scores"].append(100)`
13	**Nested dictionary** A value can be another dict you chain into.	`print(student["address"]["city"])`
14	**List of dictionaries** Many records with the same labels, like rows.	`print(people[0]["name"])`

© Kalyan Reddy Daida

Collections: Sets for Unique Values (1 of 2)

A set is an unordered collection of unique values, so it never stores duplicates and has no index to reach items by position.

#	CONCEPT	SAMPLE CODE
1	**Set of unique values** `{ }` Curly braces hold unique, unordered values.	`sam_skills = {"python", "ml", "data"}`
2	**Empty set** `set()` Use set(); {} makes an empty dict instead.	`empty = set()`
3	**Membership test** `in` `not in` Fast check if a value is in the set.	`print("ml" in sam_skills)`
4	**Add one value** `.add()` Grows the set; a duplicate add does nothing.	`sam_skills.add("sql")`

© Kalyan Reddy Daida

Collections: Sets for Unique Values (2 of 2)

A set is an unordered collection of unique values, so it never stores duplicates and has no index to reach items by position.

#	CONCEPT	SAMPLE CODE
5	**Add many values** `.update()` Bulk-adds items from a list or another set.	`sam_skills.update(["sql", "go"])`
6	**Remove a value** `.discard()` `.remove()` discard is safe; remove raises KeyError.	`sam_skills.discard("java")`
7	**Set math** `|` `&` `-` Union, intersection, and difference.	`sam_skills | tom_skills`
8	**Symmetric difference** `^` Values in exactly one set, not both.	`sam_skills ^ tom_skills`
9	**Reach the items** `for ... in` No index on a set, so loop over it (Section 06).	`for skill in sorted(sam_skills):`

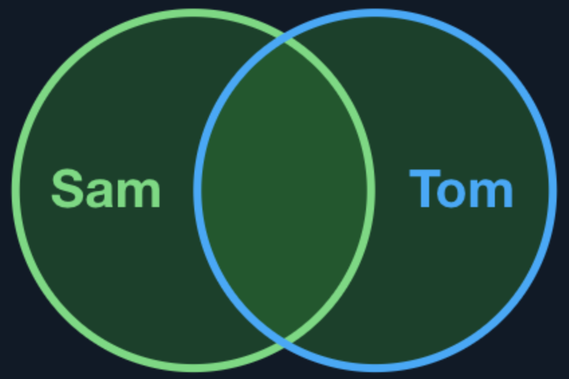
© Kalyan Reddy Daida

Collections: Set Math - Combine Two Sets

Sam's and Tom's skills, combined three ways. The shaded area is the result; we print it with sorted().

Sam = {python, ml, data} | **Tom** = {python, java, sql} | shared: **python**

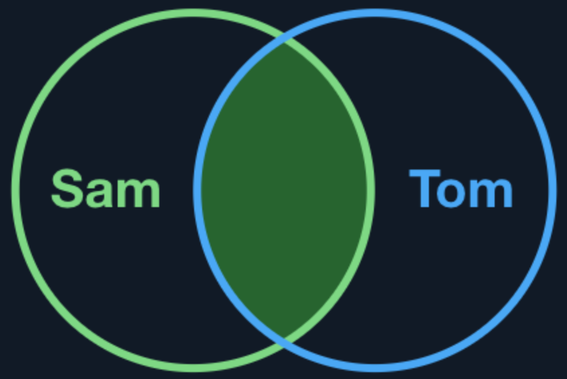
Union |




```
{data, java, ml,
 python, sql}
```

everything in EITHER set

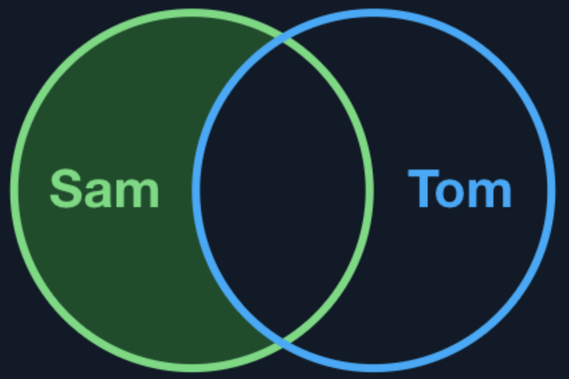
Intersection &




```
{python}
```

only what is in BOTH sets

Difference −




```
{data, ml}
```

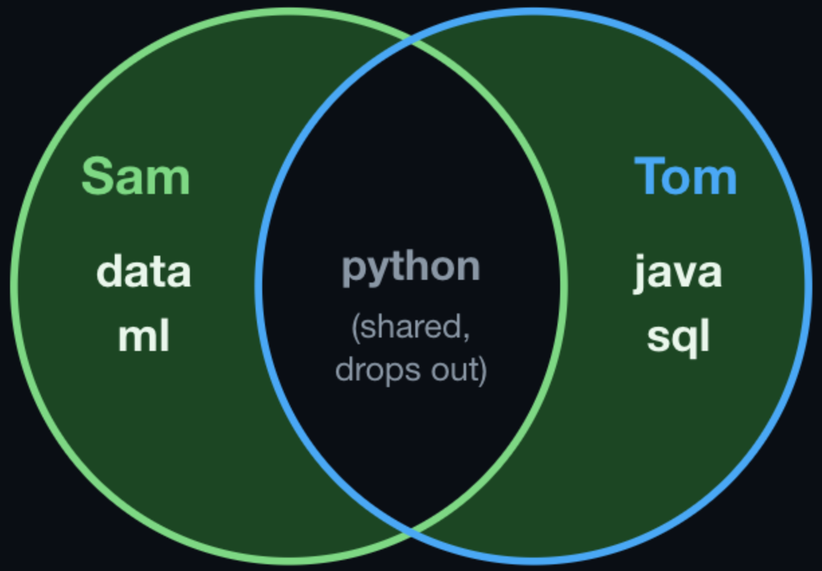
in Sam, but NOT Tom
(Sam - Tom)

© Kalyan Reddy Daida

Collections: Symmetric Difference - in Exactly One Set

The two skills each person has ALONE - everything except what they share.

Sam = {python, ml, data} | **Tom** = {python, java, sql}



Sam ^ Tom

`{data, java, ml, sql}`

Keep everything that is in **exactly one** set.

python is in BOTH, so it is the ONE value that drops out.

© Kalyan Reddy Daida

Section 06: Loops and Comprehensions Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Loops: Keywords Covered

Section 06: Loops and Comprehensions - the keywords in bright green are the new ones we cover here.

Values
`True` `False` `None`

Logic & Membership
`and` `or` `not` `in` `is`

Conditionals
`if` `elif` `else`

Loops
`for` `while` `break` `continue`

Functions
`def` `return` `lambda`

Classes
`class`

Error Handling
`try` `except` `finally` `raise` `assert`

Modules & Imports
`import` `from` `as`

Scope
`global` `nonlocal`

Context Managers
`with`

Cleanup & Flow
`pass` `del`

Generators & Async
`yield` `async` `await`

Soft Keywords
`match` `case` `_` `type`

`keyword` New in this section `keyword` Covered earlier `keyword` Not covered yet

© Kalyan Reddy Daida

Loops: The while Loop

A **while** loop runs the same block over and over as long as its condition stays True, so you decide when it stops.

#	CONCEPT	SAMPLE CODE
1	**Count up while a condition holds** `while` Loops while the test holds; update the variable to stop.	`while count <= 5:`
2	**Loop until false** Stop when the test fails - no fixed count needed.	`value = value * 2`
3	**Count down** `-=` Subtract each turn to move toward the stop.	`n -= 1`
4	**Running total** `+=` Add each value into one running total.	`total += i`
5	**Walk a list by index** `len()` Step through a list by its index positions.	`while i < len(scores):`
6	**Loop while truthy** Any truthy value works - a non-empty list is True.	`while names:`

© Kalyan Reddy Daida

Loops: What is a while Loop?

What is a while loop?

A **while** loop keeps running the same block **while its condition is True**. Something inside must change the condition, **or the loop never stops**.

WITHOUT a loop — 5 lines for 5 numbers

CODE

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

OUTPUT

```
1
2
3
4
5
```

WITH a while loop — 4 lines for any count

CODE

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

TURN BY TURN

```
count = 1   True    prints 1   count = 2
count = 2   True    prints 2   count = 3
count = 3   True    prints 3   count = 4
count = 4   True    prints 4   count = 5
count = 5   True    prints 5   count = 6
count = 6   False   loop stops
```

Now print 1 to 100?

Write 100 print lines

`print(1) ... print(100)`

Change 5 to 100 (1 line code change)

`while count <= 100:`

© Kalyan Reddy Daida

Loops: The for Loop

A **for** loop runs the same block of code once for every item in a list, string, tuple, or other sequence.

#	CONCEPT	SAMPLE CODE
1	**Loop over a list** Runs the block once per item, no counter needed.	`for score in scores:`
2	**Loop over a string** Walks a string one character at a time.	`for letter in "PYTHON":`
3	**Loop over a tuple** Reads a tuple's values one by one.	`for value in (10, 20, 30):`
4	**Running total** `+=` Add each item into a running total.	`total += score`
5	**Count matches** `if` An if inside the loop counts only matches.	`if score >= 80:`
6	**Tuple unpacking** `for a, b in` Split each pair into two variables.	`for name, score in pairs:`
7	**for vs while** Use for when you already have the items.	`while i < len(names):`

© Kalyan Reddy Daida

Loops: What is a for Loop?

What is a for loop?

A **for** loop takes the items **one at a time** and runs the block for each one. You write **no counter and no stop test** - it ends by itself when the items run out.

WITHOUT a loop one line per item

CODE

```
scores = [70, 95, 60, 88]
print(scores[0])
print(scores[1])
print(scores[2])
print(scores[3])
```

OUTPUT

```
70
95
60
88
```

WITH a for loop two lines for any list

CODE

```
scores = [70, 95, 60, 88]
for score in scores:
    print(score)
```

TURN BY TURN

```
score = 70   prints 70
score = 95   prints 95
score = 60   prints 60
score = 88   prints 88
items run out  loop stops
```

Now the list has 100 scores?

Write 100 index lines

`print(scores[0]) ... print(scores[99])`

Change nothing (0 line code change)

`for score in scores:`

© Kalyan Reddy Daida

Loops: for vs while

Use a for loop when you know the items or the count. Use a while loop when you loop until a condition is met.

Use for

You have the items or a count - a list, a string, a range.

Use while

You loop until a condition, and can't list the repeats up front.

Same list?

If you already have it, for is cleaner - no index to manage.

FOR VS WHILE

```
colors = ["red", "green", "blue"]
for color in colors:
    print(color)


balance = 1
while balance <= 100:
    balance = balance * 2
```

for always ends on its own; while you must stop yourself.

© Kalyan Reddy Daida

Loops: The range() Function

range() builds a sequence of numbers on demand, so a for loop can repeat an exact number of times.

#	CONCEPT	SAMPLE CODE
1	**Repeat n times** `range(n)` Counts 0 up to n-1, so it runs n times.	`for i in range(5):`
2	**Start and stop** `range(start, stop)` Counts from start and stops before the stop.	`for n in range(2, 6):`
3	**Step by an amount** `step` A step of 2 skips the odd numbers.	`for n in range(0, 11, 2):`
4	**Count down** `-step` A negative step walks backward.	`for i in range(5, 0, -1):`
5	**Sum a run** `+=` Add each number into one running total.	`total += n`
6	**See the numbers** `list()` range is lazy; list() reveals the numbers.	`nums = list(range(1, 6))`

© Kalyan Reddy Daida

Loops: What is range()?

What is range()?

range() hands the loop a run of numbers, so you never type them out. **range(5)** gives 0, 1, 2, 3, 4 - it starts at **0** and **stops before** the number you give.

WITHOUT range() type every number

CODE

```
numbers = [0, 1, 2, 3, 4]
for i in numbers:
    print(i)
```

OUTPUT

```
0
1
2
3
4
```

WITH range() one number, any count

CODE

```
for i in range(5):
    print(i)
```

TURN BY TURN

```
i = 0   prints 0
i = 1   prints 1
i = 2   prints 2
i = 3   prints 3
i = 4   prints 4
5 is NOT included  loop stops
```

Now repeat 100 times?

Type 100 numbers

```
numbers = [0, 1, 2, ... 99]
```

Change 5 to 100 (1 line code change)

```
for i in range(100):
```

© Kalyan Reddy Daida

Loops: break and continue (1 of 2)

break exits a loop right away and **continue** jumps to the next turn - the keywords that control the flow of a for or while loop.

#	CONCEPT	SAMPLE CODE
1	**Stop the loop early** `break` Leave the loop the moment you are done.	`break`
2	**Break out of while True** `while True` The one clean way out of an endless loop.	`while True:`
3	**Skip to the next turn** `continue` Jump to the next item without leaving the loop.	`continue`
4	**Guard with continue** `continue` Skip bad values before the real work.	`if score < 80:`
5	**continue in a while loop** `continue` Move the counter first or it loops forever.	`n += 1  # move counter first`

© Kalyan Reddy Daida

Loops: break and continue (2 of 2)

break exits a loop right away and **continue** jumps to the next turn - the keywords that control the flow of a for or while loop.

#	CONCEPT	SAMPLE CODE
6	**Do-nothing placeholder** `pass` Fill a block you will write later.	`pass  # fill in later`
7	**Loop else runs** `for...else` Runs only if the loop never hit break.	`else:`
8	**break skips the else** `for...else` A break makes the loop else stay silent.	`break`
9	**while else runs** `while...else` Runs only if the while ended without break.	`else:`

© Kalyan Reddy Daida

Loops: What is break?

What is break?

break leaves the loop the moment you are done. Without it the loop always walks the whole list, even when the answer turned up on the second turn.

WITHOUT break checks all 4 scores

CODE

```
scores = [70, 95, 60, 88]
for score in scores:
    print("checking:", score)
    if score >= 90:
        print("found:", score)
```

OUTPUT

```
checking: 70
checking: 95
found: 95
checking: 60
checking: 88
```

WITH break stops at the answer

CODE

```
scores = [70, 95, 60, 88]
for score in scores:
    print("checking:", score)
    if score >= 90:
        print("found:", score)
        break
```

TURN BY TURN

```
score = 70   under 90    keep going
score = 95   90 or more  break
score = 60   never runs
score = 88   never runs
```

Now 1000 scores, match on turn 2?

Still checks all 1000

998 turns of wasted work

Stops after 2

break

© Kalyan Reddy Daida

Loops: What is while True?

What is a while True loop?

Same job as the last concept - find the first score of 90 or above - but with no **for** loop this time. **while True** never ends on its own, so **break** is the only way out. The older trick is a **flag**, and a flag only stops the loop at the NEXT trip to the top.

STOP WITH A FLAG the turn still finishes

CODE

```
scores = [70, 95, 60, 88]
i = 0
running = True
while running:
    if scores[i] >= 90:
        print("first 90+:", scores[i])
        running = False
    print("still checking:", scores[i])
    i += 1
```

OUTPUT

```
still checking: 70
first 90+: 95
still checking: 95
```

STOP WITH break the turn ends right there

CODE

```
scores = [70, 95, 60, 88]
i = 0
while True:
    if scores[i] >= 90:
        print("first 90+:", scores[i])
        break
    print("still checking:", scores[i])
    i += 1
```

TURN BY TURN

```
70  under 90  keep going
95  90 or more  break
60  never reached
88  never reached
```

95 is the answer. Did the loop stop when it found it?

No

the flag lets the turn finish, so 95 is still checking

Yes

break leaves on the line that found it

© Kalyan Reddy Daida

Loops: What is continue?

What is continue?

continue skips the rest of THIS turn and jumps straight to the next item. Take that one line away and Tom falls through to the very line you meant to skip.

WITHOUT continue — Tom is skipped, and still served

CODE

```
names = ["Sam", "Tom", "Ben"]
for name in names:
    if name == "Tom":
        print("skipping", name)

    print("Give chocolate to:", name)
```

OUTPUT

```
Give chocolate to: Sam
skipping Tom
Give chocolate to: Tom
Give chocolate to: Ben
```

WITH continue — the turn really does stop

CODE

```
names = ["Sam", "Tom", "Ben"]
for name in names:
    if name == "Tom":
        print("skipping", name)
        continue
    print("Give chocolate to:", name)
```

TURN BY TURN

```
Sam   chocolate
Tom   continue   skipped
Ben   chocolate
finished   all 3 turns
```

One line is the only difference

without it

Tom gets a chocolate anyway

with it

the turn ends right there

© Kalyan Reddy Daida

Loops: continue in a while Loop

What is the continue trap?

continue works in a **while** loop too, but it jumps straight back to the test. If the counter has not moved yet, the loop skips **the same value forever**.

COUNTER AFTER continue the loop freezes

CODE

```
n = 0
while n < 7:
    if n % 2 == 0:
        continue
    print("odd:", n)
    n += 1
```

OUTPUT

```
no output
runs forever
n is stuck at 0
```

COUNTER FIRST skips and moves on

CODE

```
n = 0
while n < 7:
    n += 1
    if n % 2 == 0:
        continue
    print("odd:", n)
```

TURN BY TURN

```
n = 1   odd: 1
n = 2   even   continue
n = 3   odd: 3
n = 5, 7   odd: 5, odd: 7
n reaches 7   loop ends
```

Move the counter after the continue?

n never changes

`frozen on 0, forever`

n moves first

`so the skip is always safe`

© Kalyan Reddy Daida

Loops: What is pass?

What is pass?

Python needs at least one line inside every block. **pass** is the do-nothing line that fills a branch you have not written yet - it says **empty on purpose**, not forgotten.

WITHOUT pass the file will not run

CODE

```
names = ["Sam", "Tom", "Ben"]
for name in names:
    if name == "Tom":

    else:
        print("greet", name)
```

OUTPUT

```
IndentationError:
expected an indented
block after the if
```

WITH pass empty on purpose

CODE

```
names = ["Sam", "Tom", "Ben"]
for name in names:
    if name == "Tom":
        pass
    else:
        print("greet", name)
```

TURN BY TURN

```
name = Sam   greet Sam
name = Tom   pass   nothing happens
name = Ben   greet Ben
file runs   the gap is marked
```

Why not just leave it blank?

Nothing runs at all

`IndentationError on the empty if`

Runs, and the gap is deliberate

`pass`

© Kalyan Reddy Daida

Loops: What is for ... else?

What is a for ... else?

A **for** loop can carry an **else**. It runs **only when the loop finished without a break** - the "searched everything, found nothing" case. If break fires, the else is skipped.

WITHOUT for ... else track it with a flag

CODE

```
names = ["Sam", "Tom", "Ben"]
found = False
for name in names:
    if name == "Amy":
        found = True
        break
if not found:
    print("Amy not found")
```

OUTPUT

```
Amy not found
```

WITH for ... else the loop tracks it for you

CODE

```
names = ["Sam", "Tom", "Ben"]
for name in names:
    if name == "Amy":
        print("found Amy")
        break
else:
    print("Amy not found")
```

TURN BY TURN

```
name = Sam   not Amy
name = Tom   not Amy
name = Ben   not Amy
no break fired   else RUNS
had Amy been there  break, else SKIPPED
```

Forget the found = True?

It reports Amy not found
even when Amy IS in the list

Nothing to forget
else is wired to break itself

© Kalyan Reddy Daida

Loops: What is while ... else?

What is a while ... else?

The same **else** rule you met on a **for** loop also works on a **while**. Both runs below carry the SAME break - only the number it checks differs. So the else turns on whether the break **FIRED**, not on whether a break is **there**.

n == 2, THE BREAK FIRES — else is SKIPPED

CODE

```
n = 3
while n >= 1:
    if n == 2:
        break
    print("countdown", n)
    n -= 1
else:
    print("finished")
```

OUTPUT

```
countdown 3
```

n == 4, IT NEVER FIRES — else RUNS

CODE

```
n = 3
while n >= 1:
    if n == 4:
        break
    print("countdown", n)
    n -= 1
else:
    print("finished")
```

TURN BY TURN

```
n = 3   countdown 3
n = 2   countdown 2
n = 1   countdown 1
no break fired   else RUNS
prints   finished
```

Same code, same break. So what decides?

The break FIRED

`n reached 2, so the loop was cut short`

The break never fired

`a countdown from 3 never reaches 4`

© Kalyan Reddy Daida

Loops: enumerate() and Dict Items

enumerate() gives each item its position number, and a dictionary can be looped by its keys, values, or key-value pairs with **.items()**.

#	CONCEPT	SAMPLE CODE
1	**Index and item** `enumerate()` enumerate() numbers each item, starting at 0.	`for index, name in enumerate(names):`
2	**Start the count at 1** `start=` Make a human-friendly numbered list.	`enumerate(names, start=1)`
3	**Loop a dict's keys** A plain dict loop yields keys in order.	`for field in student:`
4	**Key and value** `.items()` .items() gives the key and value together.	`for field, value in student.items():`
5	**Sum the values** `.values()` Add every value with a running total.	`for score in scores_by_name.values():`
6	**Pair two lists** `zip()` zip() lines up two lists side by side.	`for name, score in zip(names, scores):`

© Kalyan Reddy Daida

Loops: What is enumerate()?

What is enumerate()?

enumerate() hands the loop the **position and the item together**, so you never keep a counter of your own - and never forget to move it on.

WITHOUT enumerate() keep your own counter

CODE

```
names = ["Sam", "Tom", "Ben"]
index = 0
for name in names:
    print(index, name)
    index += 1
```

OUTPUT

```
0 Sam
1 Tom
2 Ben
```

WITH enumerate() the position comes free

CODE

```
names = ["Sam", "Tom", "Ben"]
for index, name in enumerate(names):
    print(index, name)
```

TURN BY TURN

```
index = 0   name = Sam   prints 0 Sam
index = 1   name = Tom   prints 1 Tom
index = 2   name = Ben   prints 2 Ben
items run out  loop stops
```

Forget the index += 1?

Every line shows 0

```
0 Sam 0 Tom 0 Ben
```

Cannot happen

there is no counter to forget

© Kalyan Reddy Daida

Loops: Nested Loops

A nested loop is a loop placed inside another loop, so the **inner** loop runs fully on every turn of the **outer** loop.

#	CONCEPT	SAMPLE CODE
1	**Loop inside a loop** The inner loop runs fully on every outer turn.	`for value in row:`
2	**Every pair from two lists** Count of pairs is len(a) * len(b).	`print(letter, number)`
3	**Multiplication table** `range()` Nested range() builds a numeric grid.	`print(f"{row} x {col} = {row * col}")`
4	**break in inner loop** `break` break stops only the innermost loop.	`if col == 2: break`
5	**continue in inner loop** `continue` continue skips one inner item, keeps looping.	`if col == 2: continue`

© Kalyan Reddy Daida

Loops: What is a Nested Loop?

What is a nested loop?

A loop inside a loop, used to walk a grid. The **outer** loop picks the row, the **inner** loop walks that row - so 3 rows of 3 values is 3 x 3 = **9 visits**.

WITHOUT nesting one loop per row

CODE

```
numbers = [[10,20,30], [40,50,60], [70,80,90]]
for number in numbers[0]:
    print(number)
for number in numbers[1]:
    print(number)
for number in numbers[2]:
    print(number)
```

OUTPUT

```
10
20
30
40
50
60
70
80
90
```

WITH nesting same code, any grid

CODE

```
numbers = [[10,20,30], [40,50,60], [70,80,90]]
for row in numbers:
    print(row)
    for number in row:
        print(number)
```

OUTPUT

```
[10, 20, 30]
10
20
30
[40, 50, 60]
40
50
60
[70, 80, 90]
70
80
90
```

Now the grid has 100 rows?

Write 100 loops

one for every row

Change nothing

for row in grid:

© Kalyan Reddy Daida

Loops: List, Dict, Set Comprehensions

A **comprehension** builds a list, dict, or set in one line by transforming or filtering an existing collection.

#	CONCEPT	SAMPLE CODE
1	**List comprehension** `[ ]` Build a list in one line, item by item.	`doubled = [n * 2 for n in scores]`
2	**Filter with a test** `if` A trailing if keeps only items that pass.	`passing = [n for n in scores if n >= 80]`
3	**Dict comprehension** `{k: v}` Build a lookup of key-value pairs.	`score_map = {n: n * 2 for n in scores}`
4	**Filter a dict** `if` Add an if to keep only some pairs.	`{n: n * 2 for n in scores if n >= 80}`
5	**Set comprehension** `{ }` Builds a set and drops duplicate values.	`unique_scores = {n for n in scores}`
6	**Pick with if / else** `if/else` `list + dict` Choose between two values per item - in a list, or in a dict.	`["pass" if n >= 80 else "fail" for n in scores]` `{n: "pass" if n >= 80 else "fail" for n in scores}`

© Kalyan Reddy Daida

Loops: What is a Comprehension?

What is a comprehension?

A **comprehension** replaces the empty-list-then-append pattern with a single expression. Read it in the order you say it: **n times 2, for each n in scores**.

THE SHAPE `[expression for item in collection]`

WITHOUT a comprehension 4 lines and an empty list

CODE

```
scores = [10, 20, 30, 40]
doubled = []
for n in scores:
    doubled.append(n * 2)
print(doubled)
```

OUTPUT

```
[20, 40, 60, 80]
```

WITH a comprehension 1 line

CODE

```
scores = [10, 20, 30, 40]
doubled = [n * 2 for n in scores]
print(doubled)
```

READ IT AS

```
n = 10   10 * 2   20
n = 20   20 * 2   40
n = 30   30 * 2   60
n = 40   40 * 2   80
list is built  in one line
```

Same job, 4 lines or 1?

4 lines

`empty list, loop, append, print`

1 line

```
doubled = [n * 2 for n in scores]
```

© Kalyan Reddy Daida

Section 07: Functions Concept Map (1 of 2)

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Section 07: Functions Concept Map (2 of 2)

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Functions: Keywords Covered

Section 07: Functions - the keywords in bright green are the new ones we cover here.

Values
`True` `False` `None`

Logic & Membership
`and` `or` `not` `in` `is`

Conditionals
`if` `elif` `else`

Loops
`for` `while` `break` `continue`

Functions
`def` `return` `lambda`

Classes
`class`

Error Handling
`try` `except` `finally` `raise` `assert`

Modules & Imports
`import` `from` `as`

Scope
`global` `nonlocal`

Context Managers
`with`

Cleanup & Flow
`pass` `del`

Generators & Async
`yield` `async` `await`

Soft Keywords
`match` `case` `_` `type`

`keyword` New in this section `keyword` Covered earlier `keyword` Not covered yet

© Kalyan Reddy Daida

Functions: What is a Function?

What is a function?

A **function** is a **named block of code** you run again and again. **def** defines it; the body runs only when you **call** it with **()**.

WITHOUT a function — copy the block every time

CODE

```
print("Hello!")
print("Enjoy Python")

print("Hello!")
print("Enjoy Python")
```

OUTPUT

```
Hello!
Enjoy Python
Hello!
Enjoy Python
```

WITH a function — write it once, call it

CODE

```
def greet():
    print("Hello!")
    print("Enjoy Python")
greet()
greet()
```

WHAT RUNS

```
def greet()   no output
greet()       2 lines
greet()       2 lines
the () is what RUNS it
```

Now change the message to "Welcome!"?

Edit every copy

`2 places now, 20 later`

Edit one line (inside the def)

`def greet(): ...`

© Kalyan Reddy Daida

Functions: Define and Call a Function

A **function** is a named block of code you define once with **def**, then run many times by calling its name with **()**.

#	CONCEPT	SAMPLE CODE
1	**Define a function, then call it** `def` `()` def names a reusable block once; the () after the name is what RUNS the code inside.	`def greet():`
2	**Docstring** `"""..."""` A triple-quoted first line documents the function; __doc__ and help() read it back.	`"""Print a friendly welcome message."""`
3	**Define once, call many times** Reuse the function instead of copy-pasting the same lines.	`divider()`

© Kalyan Reddy Daida

Functions: What is a Parameter and an Argument?

One is a name, the other is a value

The **parameter** is the name in the **def** line. The **argument** is the value you pass when you **call**. The value travels in under the parameter's name.

Pass NOTHING the parameter has no value

CODE

```
def greet(name):
    print(f"Hi, {name}!")
greet()
```

OUTPUT

```
TypeError
missing name
```

Pass an ARGUMENT one function, any name

CODE

```
def greet(name):
    print(f"Hi, {name}!")
greet("Sam")
student = "Amy"
greet(student)
```

WHAT TRAVELS IN

```
name  parameter
"Sam"  argument
Hi, Sam!
Hi, Amy!
a variable works
```

So how many values do you pass? One per parameter

Too few or too many = TypeError

`greet() / greet("Sam","Tom")`

One each, left to right (by position)

`describe("Sam", "Hyderabad")`

© Kalyan Reddy Daida

Functions: One Argument

A **parameter** is the name in the function's def line; an **argument** is the value you pass in the call - so one function can work on any value you give it.

#	CONCEPT	SAMPLE CODE
1	**Parameter and argument** The def name is the parameter; the call value is the argument.	`def greet(name):`
2	**Argument from a variable** The value you pass can be a variable, not just typed text.	`greet(student)`
3	**Positional arguments** Values fill parameters by position, left to right.	`describe("Sam", "Hyderabad")`

© Kalyan Reddy Daida

Functions: `print` vs `return`

Where does the value GO?

print sends the value to the **screen** and hands your code back **None**. **return** hands the value **back to your code**, so you can store it and use it again.

PRINT ONLY you can see it, you cannot use it

```
def show(a, b):
    print(a + b)
got = show(2, 3)
```

→ THE SCREEN SHOWS **5** — you can see it → YOUR CODE GETS **None** — dead end - nothing to reuse

RETURN the value comes back to your code

```
def add(a, b):
    return a + b
total = add(2, 3)
```

→ YOUR CODE GETS **5** — a real value in total → SO YOU CAN REUSE IT `total * 10 = 50` — the value keeps going

print SHOWS, then hands back None

`got = None`

return hands the value back

`total = add(2, 3)`

© Kalyan Reddy Daida

Functions: print Shows, return Gives Back

return sends a value back from a function so you can store and reuse it, while **print()** only shows a value on screen and hands back None.

#	CONCEPT	SAMPLE CODE
1	**print() hands back** `None` You cannot store or reuse what print() shows.	`print(returned)  # None`
2	**Send a value back** `return` Hands a real value back to store and reuse.	`return a + b`
3	**Return a yes/no answer** `bool` Return a comparison to get back True or False.	`return n % 2 == 0`

© Kalyan Reddy Daida

Functions: Parameters and Logic

A function can take several **parameters** and run real **logic** inside - branches, loops, and early returns - to compute the value it returns.

#	CONCEPT	SAMPLE CODE
1	**Multiple parameters** Pass several inputs at once; they fill by position.	`def area(width, height):`
2	**Logic and return** `if/elif/else` Branch inside the function; return a value per case.	`if marks >= 90:`
3	**Loop and return** `for` Loop over items, build a running total, then return it.	`running_total = running_total + number`
4	**Return early** `return` Stop when found; the not-found answer comes after the loop.	`return False  # only after the loop`

© Kalyan Reddy Daida

Functions: where does `return False` belong?

One indent level changes the answer

return stops the function immediately. Inside the loop, **`return False`** decides on the **FIRST** item. After the loop, it decides only once **every** item has been checked.

THE BUG return False inside the loop

```
def has_even_buggy(numbers):
    for number in numbers:
        if number % 2 == 0:
            return True
        else:
            return False
```

the band is the loop body : return False is INSIDE it

```
has_even_buggy([1, 3, 4])
1 odd return False, function ends
3 never checked
4 never checked (and it is even)
```

`False` WRONG

THE FIX return False after the loop

```
def has_even(numbers):
    for number in numbers:
        if number % 2 == 0:
            return True
    return False
    # only after every item
```

the band is the loop body : return False is AFTER it

```
has_even([1, 3, 4])
1 odd keep going
3 odd keep going
4 even return True
```

`True` CORRECT

© Kalyan Reddy Daida

Functions: Default Parameters and Keyword Arguments

Default values make an argument optional, and **keyword arguments** let you pass values by name so their order does not matter.

#	CONCEPT	SAMPLE CODE
1	**Default value** `=value` Used when the caller skips that argument.	`def greet(name, greeting="Hello"):`
2	**Defaults come last** A default cannot come before a normal parameter.	`def bad(greeting="Hi", name): # error`
3	**Keyword arguments** `name=value` Pass by name, so the order does not matter.	`describe(city="Hyderabad", name="Sam")`
4	**Mutable-default trap** `=None` A default list is built once - use None instead.	`def add_item_safe(item, basket=None):`

© Kalyan Reddy Daida

Functions: why does the default list remember?

The default is built ONCE, when def runs

A default value is created at **definition** time, not on every call. So one `[]` is shared by **every** call forever. Use **None** and build a fresh list **inside** the function.

THE TRAP basket=[]

DEF TIME - RUNS ONCE

```
def add_item(item, basket=[]):
    basket.append(item)
    return basket
```

ONE list, made now
`[]`

CALL TIME - EVERY CALL SHARES THAT ONE LIST

```
add_item("apple") ['apple']
add_item("banana") ['apple', 'banana']
add_item("cherry") ['apple', 'banana', 'cherry']
```

it never clears - it only grows

you can see the shared list

```
add_item.__defaults__ (['apple', 'banana', 'cherry'],)
```

THE FIX basket=None

DEF TIME - RUNS ONCE

```
def add_item_safe(item, basket=None):
    if basket is None:
        basket = []
    basket.append(item)
    return basket
```

call 1 `[]` | call 2 `[]` | call 3 `[]`

CALL TIME - A FRESH LIST EACH TIME

```
add_item_safe("apple") ['apple']
add_item_safe("banana") ['banana']
add_item_safe("cherry") ['cherry']
```

every call starts empty

nothing is shared

None cannot hold items, so there is nothing to leak

© Kalyan Reddy Daida

Functions: how do you return TWO values?

You do not. A function always returns ONE object.
The **commas** pack your values into a single **tuple** - the brackets are not what does it. Then you choose how to receive that one tuple.

INSIDE THE FUNCTION

```
return min(numbers), max(numbers)
```

the comma builds the tuple

→

ONE VALUE COMES BACK

```
(2, 9)
```

a tuple, len 2

1. Catch it whole

```
result = low_high([4, 9, 2, 7])
```

`result (2, 9)` one name, the whole tuple

2. Unpack it

```
lo, hi = low_high([4, 9, 2, 7])
```

`lo 2 hi 9` two names, one each

The names must match the count `lo, hi, extra = low_high(...)` `ValueError: not enough values to unpack (expected 3, got 2)`

Need three? Exactly the same rule more commas, more names - still ONE tuple

```
return min(numbers), max(numbers), len(numbers)
```

→ `(2, 9, 4)`

```
lo, hi, count = low_high_count([4, 9, 2, 7])
```

→ `lo 2 hi 9 count 4`

© Kalyan Reddy Daida

Functions: Return Many Values

A function can hand back several results at once by separating them with commas, and Python packs them into one **tuple** you can **unpack** into separate variables.

#	CONCEPT	SAMPLE CODE
1	**Return a tuple** `a, b` Commas pack the values into one tuple.	`return min(numbers), max(numbers)`
2	**Unpack into names** `lo, hi =` Catch each returned value in its own variable.	`lo, hi = low_high([4, 9, 2, 7])`
3	**Return three values** The same trick scales to three or more.	`lo, hi, count = low_high_count([4, 9, 2, 7])`

© Kalyan Reddy Daida

Functions: *args - the problem it solves

You are writing ONE function. The caller might have 3 values, or 10, or 40.
A normal `def` locks the count in: an extra value means **editing the function**, a different count simply **crashes**.

THE CALLER HAS	THE PROBLEM - A FIXED COUNT `def total(a, b, c)` Every parameter is typed by hand, so the count is locked in.	THE FIX - ANY COUNT `def total(*nums)` `*args` collects any number of positional values into ONE tuple. The name is yours - `*args` is just the convention; `*nums` works the same.
CASE 1 **3** VALUES	`def total(a, b, c):` `  return a + b + c` `print(total(1, 2, 3))` `print(total(10, 20, 30, 40))` `6` `TypeError: total() takes 3 positional arguments but 4 were given`	`def total(*nums):` `  return sum(nums)` `print(total(1, 2, 3))` `print(total(10, 20, 30, 40))` `6` `100`
CASE 2 **10** VALUES	`def total(a, b, c, d, e, f, g, h, i, j):` `  return a + b + c + d + e + f + g + h + i + j` `print(total(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))` Ten values? **Ten parameters** - and a ten-term return line. Eleven? Rewrite both, again. `55  only after rewriting the function`	`def total(*nums):` `  return sum(nums)` `print(total(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))` The very same def as Case 1 - not one character changed. It would take 40 just as happily. `55  nothing to rewrite`

© Kalyan Reddy Daida

Functions: **kwargs - the problem it solves

One caller wants gift wrap. The next wants a note. The third wants both, and express.
A normal `def` locks in **which options exist and in what order**: a new one means editing the function, and leaving one out simply **crashes**.

THE CALLER WANTS

THE PROBLEM - A FIXED SET

`def make_order(item, gift_wrap, note)`

Every option is typed by hand, so the set and the order are locked in.

CASE 1 — 2 OPTIONS

```
def make_order(item, gift_wrap, note):
    print("options:", gift_wrap, note)

make_order("pen", True, "Thanks")
make_order("pen", True)
```

```
options: True Thanks
TypeError: make_order() missing 1 required
positional argument: 'note'
```

CASE 2 — 4 OPTIONS

```
def make_order(item, gift_wrap, note,
               express, insured):
    print("options:", gift_wrap, note, express, insured)

make_order("pen", True, "Hi", False, True)
```

The signature no longer fits one line, and the call is now four bare values. Which one is insured?

```
options: True Hi False True
```

THE FIX - ANY OPTIONS

`def make_order(item, **options)`

`**kwargs` collects any number of named values into ONE dict.

The name is yours - `**kwargs` is just the convention; `**options` works the same.

```
def make_order(item, **options):
    print("options:", options)

make_order("pen", gift_wrap=True, note="Thanks")
make_order("pen", gift_wrap=True)
```

```
options: {'gift_wrap': True, 'note': 'Thanks'}
options: {'gift_wrap': True}
```

```
def make_order(item, **options):
    print("options:", options)

make_order("pen", gift_wrap=True, note="Hi",
           express=False, insured=True)
```

The very same def as Case 1 - and every value arrives wearing its own name.

```
options: {'gift_wrap': True, 'note': 'Hi', 'express': False,
'insured': True}
```

© Kalyan Reddy Daida

Functions: *args and **kwargs - in the def

Two ways to say "however many you like"

`*args` gathers any number of **positional** values into a **tuple**. `**kwargs` gathers any number of **named** values into a **dict**. Both live in the **def** line, and both mean the caller is not limited to a fixed count.

*args

any number of positional values
arrive as a tuple

```
# Define Function
def total(*nums):
    return sum(nums)

# Calling Function
print(total(1, 2, 3))
print(total(10, 20, 30, 40))
```

OUTPUT

```
6
100
```

**kwargs

any number of named values
arrive as a dict

```
# Define Function
def make_order(item, **options):
    print("options:", options)

# Calling Function
make_order("pen", gift_wrap=True, note="Thanks")
make_order("pen", gift_wrap=True)
```

OUTPUT

```
options: {'gift_wrap': True, 'note': 'Thanks'}
options: {'gift_wrap': True}
```

The names are yours - `*args` and `**kwargs` are only a convention. `*nums` and `**info` work exactly the same. It is the star that does the work, not the word.

© Kalyan Reddy Daida

Functions: * and ** - at the call

Same symbols, opposite direction

In the **def** they GATHER many values into one object. At the **call** they do the reverse: they **SPREAD one object you already have back into separate arguments**. Where the symbol sits is what decides which.

*list

spread a list into positional arguments

```
# Define Function
def total(*nums):
    return sum(nums)

# Calling Function (unpack the list with *)
scores = [5, 10, 15]
print(total(*scores))
```

`total(5, 10, 15)` is what actually runs

OUTPUT

```
30
```

**dict

spread a dict into named arguments

```
# Define Function
def make_order(item, **options):
    print("options:", options)

# Calling Function (unpack the dict with **)
extras = {"gift_wrap": True, "note": "Thanks"}
make_order("pen", **extras)
```

`make_order("pen", gift_wrap=True, note="Thanks")` is what actually runs

OUTPUT

```
options: {'gift_wrap': True, 'note': 'Thanks'}
```

You already have the values in a list or a dict - * and ** save you unpacking them by hand. The def side gathers; the call side spreads.

© Kalyan Reddy Daida

Functions: keyword-only * and both together

Two rules about the def line itself

A **bare** `*` forces everything after it to be passed **by name**. And one function can take both - but `*args` must come **before** `**kwargs`.

def f(*, ...)

a bare star makes what follows keyword-only

```
# Define Function
def make_box(*, width, height):
    return width * height

# Calling Function
print(make_box(width=3, height=4))
```

```
print(make_box(3, 4))
TypeError: make_box() takes 0 positional
arguments but 2 were given
```

OUTPUT

```
12
```

*args, **kwargs

one function can take both - a tuple and a dict

```
# Define Function
def make_order(*items, **options):
    print("items:", items)
    print("options:", options)

# Calling Function
make_order("pen", "book", gift_wrap=True, note="Thanks")
```

OUTPUT

```
items: ('pen', 'book')
options: {'gift_wrap': True, 'note': 'Thanks'}
```

Order is not a style choice - it is a rule. `make_order(gift_wrap=True, "pen")` **is a SyntaxError: a keyword argument can never come before a positional one.**

© Kalyan Reddy Daida

Functions: *args and **kwargs

***args** and ****kwargs** let one function accept any number of positional and named arguments, gathering the extras into a tuple and a dict.

#	CONCEPT	SAMPLE CODE
1	**Collect positional values** `*args` Gathers any number of values into a tuple.	`def total(*nums):`
2	**Collect named values** `**kwargs` Gathers name=value inputs into a dict.	`def make_order(item, **options):`
3	**Unpack into a call** `*` `**` Spread a list or dict straight into a call.	`print(total(*scores))`
4	**Keyword-only arguments** `*` A bare * forces callers to pass by name.	`def make_box(*, width, height):`
5	**Take both at once** `*args` `**kwargs` *args (tuple) must come before **kwargs (dict).	`def make_order(*items, **options):`

© Kalyan Reddy Daida

Functions: Variable Scope (LEGB)

Scope is the region of code where a name is visible, and the LEGB rule sets the order Python searches for it - Local, Enclosing, Global, Built-in.

#	CONCEPT	SAMPLE CODE
1	**Local scope** A name made inside a function lives only there.	`course = "Python"`
2	**Global scope** A top-level name any function can read.	`platform = "StackSimplify"`
3	**Enclosing scope** A nested function sees the outer function's names.	`print(course, "on", platform)`
4	**LEGB lookup order** Python searches Local, Enclosing, Global, Built-in.	`print(x)  # Local wins here`
5	**The global statement** `global` Lets a function change a global value.	`global score`
6	**The nonlocal statement** `nonlocal` Lets an inner function change the enclosing value.	`nonlocal score`

© Kalyan Reddy Daida

Functions: scope - where a name lives

Every variable has a SCOPE

A scope is the region of your program where a name exists and can be seen. You write **three** of them: **Local** inside a function, **Enclosing** in the function around it, and **Global** at the top level of the file. One course page, one name at each depth.

L LOCAL

made **inside** a function
gone the moment it ends

```
def show_course():
    course = "Python"
    print("Course:", course)

show_course()
```

OUTPUT

```
Course: Python
```

G GLOBAL

made at the **top level** of the file
any function can READ it

```
platform = "StackSimplify"

def show_platform():
    print("Platform:", platform)

show_platform()
```

OUTPUT

```
Platform: StackSimplify
```

E ENCLOSING

the **outer function's** variable
read by a function nested inside

```
def show_course_page():
    course = "Python"

    def show_title():
        print(course, "on", platform)

    show_title()

show_course_page()
```

OUTPUT

```
Python on StackSimplify
```

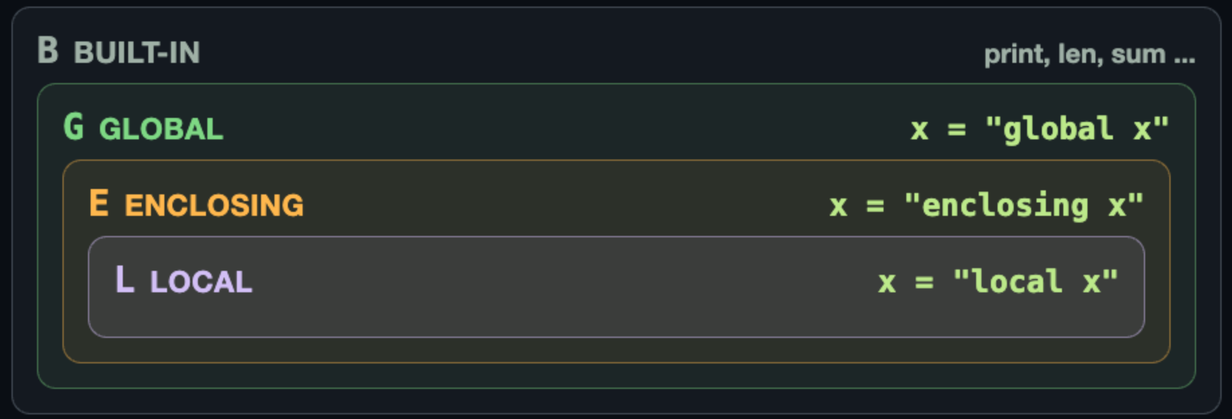
Look at the last card: `show_title` reaches the enclosing `course` AND the global `platform` in one line. There is a fourth scope you never write - Built-in, the names Python ships with like `print` and `len`. So which one wins when the SAME name exists in several? That is LEGB, next.

© Kalyan Reddy Daida

Functions: the LEGB rule - which x does Python use?

Reading a name searches OUTWARD

Python looks for a name Local, then Enclosing, then Global, then Built-in - **the first match wins** and the search stops right there. That is why the very same **print(x)** gives a different answer at each depth.



start where you stand → search outward, first match wins

AND WHEN IT IS IN NONE OF THEM - THE B

```
def show_length():
    print(len("Python"))
show_length()
```

Nothing we wrote is called **len**, so the search runs past L, E and G and finds it in Built-in. Prints **6**.

Never shadow one: `len = 5` hides the built-in, and len("hi") is broken for the rest of the file.

```
x = "global x"

def legb_outer():
    x = "enclosing x"
    def legb_inner():
        x = "local x"
        print(x)   1
    legb_inner()
    print(x)       2
legb_outer()
print(x)           3
```

1 local x stops at L
2 enclosing x legb_outer's own x wins
3 global x outside both, finds G

That is READING. Assigning is the opposite - it always creates a LOCAL name, which is exactly why global and nonlocal exist. Next slide.

© Kalyan Reddy Daida

Functions: `global` and `nonlocal` - writing further out

Assigning inside a function ALWAYS makes a local - unless you say otherwise

So `score = score + 10` reads a name that is already local and still empty: **UnboundLocalError**. One word fixes it, and which word depends only on **where the real score lives.**

`global`

the score lives at the **top level of the file**

WITHOUT global

```
score = 20

def add_points_bad():
    score = score + 10
```

`add_points_bad()` raises
`UnboundLocalError: cannot access local variable 'score' where it is not associated with a value`

WITH global

```
score = 20

def add_points():
    global score
    score = score + 10
    print("Inside the function score:", score)
```

add_points() twice

```
Inside the function score: 30
Inside the function score: 40
Global score: 40
```

`nonlocal`

the score lives in the **function around it**

WITHOUT nonlocal

```
def create_score_bad():
    score = 20
    def add_points():
        score = score + 10
```

`add_points()` raises
`UnboundLocalError: cannot access local variable 'score' where it is not associated with a value`

WITH nonlocal

```
def create_score():
    score = 20
    def add_points():
        nonlocal score
        score = score + 10
        print("Inner Function score:", score)
```

add_points() twice

```
Inner Function score: 30
Inner Function score: 40
Enclosing score: 40
```

Use these sparingly - returning a value is usually cleaner. And that inner `add_points` remembering score is a CLOSURE: the idea behind decorators.

© Kalyan Reddy Daida

Functions: First-class and Higher-order

A function in Python is itself a **value** you can store and pass around, so a **higher-order function** is simply one that takes or returns another function.

#	CONCEPT	SAMPLE CODE
1	**A function is a value** Store it in a variable with no parentheses.	`yeller = shout  # no ()`
2	**Function as an argument** A higher-order function takes another function.	`apply_twice(add_three, 10)`
3	**Transform every item** `map()` Apply one function across a whole list.	`list(map(add_three, prices))`
4	**Keep matching items** `filter()` Keep only the items that pass a test.	`list(filter(is_big, prices))`
5	**Dict dispatch** `dict` Store functions in a dict, pick one by key.	`operations["tax"](1000)`

© Kalyan Reddy Daida

Functions: a function is a VALUE

One character decides it: the parentheses

The name **without** `()` is **the function itself** - a value you can store, like a number or a string. Add `()` and you get **the result of running it**.

THE FUNCTION WE ARE STORING

```
def shout(text):
    return text.upper() + "!"
```

NO parentheses `shout`

```
yeller = shout
```

yeller holds `the function`

type `function`

so you can still call it, through the new name

```
yeller("hello") HELLO!
```

WITH parentheses `shout()`

```
result = shout("hello")
```

result holds `'HELLO!'`

type `str`

the function is gone - only its answer is left

```
print(result("x")) TypeError: 'str' object is not callable
```

A variable can hold a function exactly like it holds anything else:

`total = 5`	total holds a number
`name = "Sam"`	name holds a string
`yeller = shout`	yeller holds a function - same idea

© Kalyan Reddy Daida

Functions: a function that takes a function

The function you pass in becomes `func` inside

A **higher-order** function takes another function as a parameter and calls it. Pass the **name with no `()`**, and inside the body `func` is just another name for the function you handed in - so `func(value)` runs it. These three differ in **one thing only**: how many times they call it.

THE FUNCTION WE HAND IN

```
def add_three(n):
    return n + 3
```

apply_once

```
def apply_once(func, value):
    return func(value)
```

`func` is `add_three`
`value` is `10`

```
apply_once(add_three, 10)
add_three(10)
```

13 10 TO 13

apply_twice

```
def apply_twice(func, value):
    return func(func(value))
```

`func` is `add_three`
`value` is `10`

```
apply_twice(add_three, 10)
add_three(add_three(10))
```

16 10 TO 13 TO 16

apply_thrice

```
def apply_thrice(func, value):
    return func(func(func(value)))
```

`func` is `add_three`
`value` is `10`

```
apply_thrice(add_three, 10)
add_three(add_three(add_three(10)))
```

19 10 TO 13 TO 16 TO 19

`add_three` never changed, and neither did the `10` we passed in. Only the wrapper did. That is the point of a higher-order function: it does not care which function it receives - pass `shout` from Concept-01 instead and these same three wrappers still work.

© Kalyan Reddy Daida

Functions: The lambda Expression

A **lambda** is a small function with no name, written on one line as **lambda args: expression** and passed straight into another function.

#	CONCEPT	SAMPLE CODE
1	**Anonymous function** `lambda` A lambda has no name; to name a function, use def.	`(lambda x: x * x)(5)`
2	**Lambda as a key** `key=` Sort words by length, not alphabetically.	`sorted(words, key=lambda word: len(word))`
3	**Sort by a field** `pair[1]` Reach inside each item to sort by a field.	`sorted(pairs, key=lambda pair: pair[1])`
4	**Transform every item** `map()` A lambda works anywhere a function is expected.	`list(map(lambda n: n + 3, prices))`
5	**Keep matching items** `filter()` A one-line test, written where it is used.	`list(filter(lambda n: n > 100, prices))`

© Kalyan Reddy Daida

Functions: `lambda` - a function with no name

One expression, and that expression IS the answer

A **lambda** is a tiny **anonymous** function: no name, no `return` keyword, exactly one expression. Use **def** when the function needs a **name**; use a lambda when you are handing it straight to something, right here.

THE FORM `lambda arguments: expression`

ON THIS SLIDE `lambda x: x * x` one argument `x`, and the expression `x * x` IS what comes back

no name | no return | one expression

def

a **NAMED** function you can call again by that name

```
def square_number(x):
    return x * x

print(square_number(5))
```

OUTPUT

```
25
```

lambda

the **SAME** rule, built and called on the spot

```
print((lambda x: x * x)(5))
```

OUTPUT

```
25
```

Same answer, 25. The only difference is what is left behind: def leaves a name you can call again, the lambda leaves nothing. A lambda can take several arguments too: `(lambda x, y: x + y)(2, 3)` is 5.

© Kalyan Reddy Daida

Functions: lambda - as a sort key

The key decides WHAT to compare

`sorted(items, key=...)` does not compare the items themselves - it compares whatever the key function returns for each one. Any function will do, named or anonymous.

no key at all `sorted(words)` gives `['apple', 'banana', 'fig', 'kiwi']` alphabetical

def

the key is a **NAMED** function, passed by name

```
words = ["banana", "kiwi", "apple", "fig"]

def get_length(word):
    return len(word)

by_length = sorted(words, key=get_length)
print(by_length)
```

OUTPUT

```
['fig', 'kiwi', 'apple', 'banana']
```

lambda

the **SAME** key, written inside the call

```
words = ["banana", "kiwi", "apple", "fig"]
by_length = sorted(words, key=lambda word: len(word))
print(by_length)
```

OUTPUT

```
['fig', 'kiwi', 'apple', 'banana']
```

Identical output, shortest first - so pick by whether the key deserves a name. def when it is worth naming or reusing; lambda when it is this small and used once.

© Kalyan Reddy Daida

Functions: a key that reaches inside each item

Sort records by a field

When each item is a small record, the key returns the ONE field you want to order by. This is the shape you meet most often in real code.

each item is a pair `("Sam", 70)` so `pair[0]` is the name, `pair[1]` is the score

def

a **NAMED** key that picks out the score

```
pairs = [("Sam", 70), ("Tom", 60), ("Ben", 95)]

def get_score(pair):
    return pair[1]

by_score = sorted(pairs, key=get_score)
print(by_score)
```

OUTPUT

```
[('Tom', 60), ('Sam', 70), ('Ben', 95)]
```

lambda

the **SAME** pick, short enough to read inline

```
pairs = [("Sam", 70), ("Tom", 60), ("Ben", 95)]
by_score = sorted(pairs, key=lambda pair: pair[1])
print(by_score)
```

OUTPUT

```
[('Tom', 60), ('Sam', 70), ('Ben', 95)]
```

Sorted by SCORE, not by name - 60, 70, 95. Change `pair[1]` to `pair[0]` and the same line sorts by name instead. The key is the whole control.

© Kalyan Reddy Daida

Functions: when to use a `lambda` - and when not to

The rule of thumb: a lambda is for a function you do not want to name

Both write the **same** rule and both give the **same** answer, so this is never about what is possible; it is about what **reads** better. Ask one question: **does this function deserve a name?**

Reach for `lambda`

- You are handing it straight to another function and nowhere else, like `sorted(words, key=lambda word: len(word))`
- The whole rule is one short expression you can read at a glance, like `lambda pair: pair[1]`
- Naming it would add a line and say nothing that the expression does not already say

Reach for `def`

- You will call it again; a name is the only way to reach it a second time
- It needs more than one expression, or any statement at all, and a lambda cannot hold one
- It is long enough that a good name would explain it, the way `get_score` tells the reader more than the code does

A LAMBDA CANNOT HOLD `if / elif / else` `for` `while` `try / except` `x = 5` `return` `a docstring`

ALL THREE TAKE A FUNCTION `sorted(key=...)` `map(...)` `filter(...)` you used all three in this step; a lambda fits every one of them

One thing never to do: give a lambda a name. `square_number = lambda x: x * x` takes the one advantage a lambda has (being anonymous) and throws it away, and the error messages get worse too. If it needs a name, it needs `def`.

© Kalyan Reddy Daida

Functions: Modules and Imports

A **module** is a .py file full of functions, and an **import** pulls that code into your program so you can reuse it.

#	CONCEPT	SAMPLE CODE
1	**Import a module** `import` Load a whole module, then call module.function().	`import math`
2	**Import specific names** `from` Pull only the names you need out of a module.	`from math import pi, ceil`
3	**Short alias** `as` Give a long module name a short handle.	`import math as m`
4	**Import your own file** `.py` Reuse functions you wrote in another .py file.	`from texttools import make_upper`

© Kalyan Reddy Daida

Functions: what name does import put in your file?

The import form decides how you must write the call

`import math` lands the **module**, so you write `math.sqrt()`. `from math import ceil` lands the **name itself**, so you write `ceil()` with no prefix. That is the whole difference.

YOU WRITE	WHAT LANDS IN YOUR FILE	SO YOU CALL IT	GIVES
`import math`	`math` the module	`math.sqrt(144)`	`12.0`
`from math import pi, ceil`	`pi, ceil` the names	`ceil(4.1)`	`5`
`import math as m`	`m` renamed module	`m.sqrt(49)`	`7.0`
`from math import factorial as fact`	`fact` renamed name	`fact(5)`	`120`

Your own files work exactly the same way: `from texttools import make_upper` lands `make_upper` so `make_upper("welcome")` gives `WELCOME`

as is just a rename; nothing else changes. It is why you will see import numpy as np everywhere later. Python finds your own module because it sits in the same folder.

© Kalyan Reddy Daida

Functions: recursion - a function that calls itself

Rule 1 - call yourself on something SMALLER
Every call hands on a smaller job: `countdown(n - 1)`

Rule 2 - a BASE CASE that stops
One value that answers without calling again: `n == 0`

countdown(3)

the work happens on the way **DOWN**
nothing comes back

CALLS GO DOWN

```
countdown(3)          prints 3
  countdown(2)        prints 2
    countdown(1)      prints 1
      countdown(0)    prints Liftoff!
```

BASE CASE - STOPS HERE

OUTPUT `3 2 1 Liftoff!`

factorial(5)

the answer is built on the way **BACK UP**
each return feeds the line above it

VALUES COME BACK UP

```
factorial(5) = 5 * factorial(4)          5 * 24 = 120
  factorial(4) = 4 * factorial(3)          4 * 6 = 24
    factorial(3) = 3 * factorial(2)          3 * 2 = 6
      factorial(2) = 2 * factorial(1)          2 * 1 = 2
        factorial(1) = 1                             1
```

BASE CASE - THE RETURNS START HERE

OUTPUT `120`

No base case, no stop. A recursion that never reaches its base case keeps calling until Python stops it with `RecursionError`, after about 1000 calls. Most everyday repeating is still a loop - recursion is for problems shaped like a smaller version of themselves.

© Kalyan Reddy Daida

Functions: Recursion

Recursion is when a function **calls itself** to solve a smaller version of the same problem, and a base case stops it.

#	CONCEPT	SAMPLE CODE
1	**A function calls itself** It repeats by calling itself; a base case stops it.	`countdown(n - 1)`
2	**Factorial, the classic example** n! = n * factorial(n-1), and factorial(1) is 1.	`return n * factorial(n - 1)`

© Kalyan Reddy Daida

Section 08: Type Hints Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Type Hints: the colon and the arrow

A hint is a LABEL, not a rule

It tells you and your editor what a function **expects** and what it **gives back**. It does not change how the code runs.

THE SAME FUNCTION WITHOUT HINTS `def average(a, b):` Perfectly valid Python. It just never says **what a and b should be**, or **what you get back**.

```
def average(a: float, b: float) -> float:
```

after each colon, what goes IN

`a: float` and `b: float` label the two parameters

after the arrow, what comes OUT

`-> float` labels the returned value

THE HINTED DEF	THE CALL	GIVES
`def greet(name: str) -> str:`	`greet("Kalyan")`	`Hello, Kalyan!`
`def area(width: int, height: int) -> int:`	`area(4, 5)`	`20`
`def average(a: float, b: float) -> float:`	`average(10.0, 20.0)`	`15.0`

The shape never changes. Every parameter gets a type after its colon, and the whole function gets one type after the arrow.

© Kalyan Reddy Daida

Type Hints: Function Parameter and Return Hints

A **type hint** labels each parameter after a colon and the return value after **->**, so editors and readers know the types a function expects and gives back.

#	CONCEPT	SAMPLE CODE
1	**Parameter and return hint** `: str` `->` Colon marks the input type, -> marks the return type.	`def greet(name: str) -> str:`
2	**Several parameters** `int` Each parameter carries its own type hint.	`def area(width: int, height: int) -> int:`
3	**Decimal number hints** `float` Use float when values can carry decimals.	`def average(a: float, b: float) -> float:`

© Kalyan Reddy Daida

Type Hints: Variables and Containers

A container hint has two halves

The word outside the brackets is the **container**; the part inside is **what it holds**. `list[int]` is not just "a list", it is a list of whole numbers.

A plain variable

name : type, one type, nothing inside

```
name: str = "Kalyan"
age: int = 30
price: float = 19.99
is_active: bool = True
```

PRINTS `Kalyan 30 19.99 True`

A container

the brackets say what is held inside

```
scores: list[int] = [70, 95, 60, 88]
stock: dict[str, int] = {"pens": 3, "books": 1}
```

In `dict[str, int]` the first type is the key, the second is the value, so this maps text keys to whole numbers.

sum(scores)	313
stock["pens"]	3

It works the same on a function: `def total(prices: list[int]) -> int:` takes a list of numbers and gives one number back. `total([10, 20, 30])` is `60`.

© Kalyan Reddy Daida

Type Hints: Variable and Container Hints

A type hint labels a variable with its type, and a **container hint** like **list[int]** says exactly what type sits inside.

#	CONCEPT	SAMPLE CODE
1	**Variable hint** `: type` Put a colon and the type after the name.	`name: str = "Kalyan"`
2	**List hint** `list[int]` Say a list holds whole numbers inside.	`scores: list[int] = [70, 95, 60, 88]`
3	**Dict hint** `dict[str, int]` Map text keys to whole-number values.	`stock: dict[str, int] = {"pens": 3}`
4	**Function hint** `-> int` Hint the input type and the return type.	`def total(prices: list[int]) -> int:`

© Kalyan Reddy Daida

Type Hints: what a hint promises, and what it does not

A hint is a message to the reader, not a guard on the door

It can **tell** you that a value might be missing. It cannot stop anyone passing the wrong thing. Nothing does, until you reach for a tool that checks.

What it DOES tell you

"or None" tells you the answer may be missing, so you handle it

```
def find_price(name: str) -> int | None:
```

```
find_price("pen")    2
find_price("eraser") None
```

The hint told you None was possible

so you check before you use the value

What it does NOT do

Python does not check hints while the code runs

```
def greet(name: str) -> str:
```

```
greet("Kalyan") Hello, Kalyan!
greet(123)      Hello, 123!
```

A number was passed and it still ran

Python ran it exactly as written

So why bother? Because other tools read what Python ignores. The next slide shows exactly who reads a hint, and which of them will actually stop bad data.

© Kalyan Reddy Daida

Type Hints: who actually enforces them

Python never checks a hint. Other tools choose to.
The hint `name: str` is the **same text** in every row below. What changes is **who reads it, and when**.

WHO READS THE HINT	WHEN	ENFORCES?	WHAT YOU ACTUALLY SEE
`Python itself`	while it runs	NO	`greet(123)` runs and prints `Hello, 123!`
`@dataclass`	while it runs	NO	without **type hints** there is no `__init__` constructor; even then, the values are never checked
`Pydantic`	while it runs	YES	checks every value against its hint, and converts what it safely can
`FastAPI`	on every request	YES	bad input never reaches your function, because Pydantic checks it first

IS IT A BEST PRACTICE? **Yes. They cost nothing while your code runs, they tell the next reader what you meant, and Pydantic and FastAPI cannot check anything without them.**

So why write hints at all? Python ignores them, but the tools built on top of Python do not. You write the hint once, and Pydantic and FastAPI read it and put it to work.

© Kalyan Reddy Daida

Type Hints: Optional Values and Not Enforced

The **X | None** hint marks a value that might be missing, and Python never enforces hints when the code runs; they guide people and tools only.

#	CONCEPT	SAMPLE CODE
1	**Optional value** `X | None` Says a value is that type, or None when missing.	`def find_price(name: str) -> int | None:`
2	**Hints not enforced** Python runs the code even when a hint is wrong.	`print(greet(123))  # still runs`

© Kalyan Reddy Daida

Section 09: Error Handling and Debugging Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Errors: Keywords Covered

Section 09: Error Handling and Debugging - the keywords in bright green are the new ones we cover here.

Values
`True` `False` `None`

Logic & Membership
`and` `or` `not` `in` `is`

Conditionals
`if` `elif` `else`

Loops
`for` `while` `break` `continue`

Functions
`def` `return` `lambda`

Classes
`class`

Error Handling
`try` `except` `finally` `raise` `assert`

Modules & Imports
`import` `from` `as`

Scope
`global` `nonlocal`

Context Managers
`with`

Cleanup & Flow
`pass` `del`

Generators & Async
`yield` `async` `await`

Soft Keywords
`match` `case` `_` `type`

`keyword` New in this section `keyword` Covered earlier `keyword` Not covered yet

© Kalyan Reddy Daida

Errors: crash, or keep going

try / except is not about the message - it is about what happens NEXT

An unhandled error stops the whole program where it stands. Put the risky lines in `try` and the program **survives the bad value and carries on.**

Without try / except

```
def calculate_books_per_student(b, s):
    books = int(b)
    students = int(s)
    return books / students
```

RUNNING

- ("100", "20") 5.0
- ("abc", "20") ValueError
- ("100", "0") never reached
- everything after never reached

One bad value ends the program

With try / except

```
def calculate_books_per_student(b, s):
    try:
        books = int(b)
        students = int(s)
        return books / students
    except Exception:
        return "Something went wrong with the input."
```

RUNNING

- ("100", "20") 5.0
- ("abc", "20") Something went wrong...
- ("100", "0") Something went wrong...
- The program kept running.

Every call answers; nothing stops

try holds the lines that might fail. except is what to do instead of crashing. The lines after the call still run - that is the whole win.

© Kalyan Reddy Daida

Errors: The try / except Block

try / **except** lets your program run risky code and catch any error, so one bad value cannot crash the whole program.

#	CONCEPT	SAMPLE CODE
1	**Bad input crashes everything** One bad value ends the program; nothing after it runs.	`int("abc")  # ValueError`
2	**Catch it with try / except** `try` Wrap the risky lines; except runs instead of crashing.	`try: ... except Exception:`

© Kalyan Reddy Daida

Errors: one try, many exits

The kind of error decides which except block runs

Name the error you expect and each problem gets **its own honest reply** instead of one vague message.

```
try:
    books = int(total_books)
    students = int(total_students)
    result = books / students
except ValueError:
    return "Books and students must be whole numbers."
except ZeroDivisionError:
    return "There must be at least one student."
else:
    # only when NOTHING was raised
    print("Input looked good, sharing the books.")
    return result
```

```
("100", "20") ->   NO ERROR - else RUNS
                   Input looked good, sharing the books.
                   5.0
```

```
("abc", "20") ->   except ValueError
                   Books and students must be whole numbers.
```

```
("100", "0") ->    except ZeroDivisionError
                   There must be at least one student.
```

When two errors deserve the SAME reply, catch them together

`except (ValueError, ZeroDivisionError):` answers both with one line - `"Please enter valid numbers (students above zero)."`

Read it top to bottom: Python takes the first except that matches, and else is the lane that opens only when no error was raised at all.

© Kalyan Reddy Daida

Errors: Catching Specific Exceptions

A specific **except** block catches one named error type, so each kind of problem gets its own clear message.

#	CONCEPT	SAMPLE CODE
1	**Separate except blocks** `except` Each error type gets its own reply.	`except ValueError:`
2	**Success-only step** `else` else runs only when try had no error.	`else:`
3	**Catch two together** `except (A, B)` One message for two error types.	`except (ValueError, ZeroDivisionError):`

© Kalyan Reddy Daida

Errors: finally always runs

Cleanup you cannot forget

A `finally` block runs whichever way the call ended - success, caught error, **even after a return**. It is where you close things and tidy up.

```
try:
    books = int(total_books)
    students = int(total_students)
    return books / students
except (ValueError, ZeroDivisionError):
    return "Please enter valid numbers..."
finally:
    print("Finished sharing the books.")
```

The good call - ("100", "20")

```
Finished sharing the books.
5.0
```

finally printed BEFORE the value came back

The bad call - ("100", "0")

```
Finished sharing the books.
Please enter valid numbers...
```

same line, same place, after the error too

`try` succeeded
finally runs

`except` handled it
finally runs

`return` already happened
finally still runs

Look at the order in both outputs: "Finished sharing the books." appears before the returned value. The `return` value waits while `finally` does its job.

© Kalyan Reddy Daida

Errors: finally, Cleanup That Always Runs

A **finally** block always runs after the try and except, whether the code worked or failed, so it is where you put cleanup that must never be skipped.

#	CONCEPT	SAMPLE CODE
1	**The finally block** `finally` **Runs whether the try worked or failed.**	`finally:`

Errors: raise one, catch it, read it

Some bad input is not a Python error; it is YOUR rule

Nothing crashes on minus five books; Python is happy. `raise` is how you say **this is not allowed here**, and the sentence you write is the sentence the caller reads.

INSIDE THE FUNCTION, you raise it

```
if students <= 0:
    raise ValueError("There must be at least one student")
```

THE SAME SENTENCE

WHERE YOU CALL IT, you catch it

```
try:
    calculate_books_per_student("100", "0")
except ValueError as err:
    print("Caught a problem:", err)
```

WHAT PRINTS

```
5.0  the good call still returns
Caught a problem: There must be at least one student

CONTINUE PROGRAM – The program kept running
```

raise
Fires only when your rule is broken. Good input returns exactly as it always did.

as err
Hands you the error object, so you can print what went wrong.

your sentence
Is the error screen your user reads. Write one they can act on.

© Kalyan Reddy Daida

Errors: raise to Reject Bad Input

raise stops a function on purpose when the input breaks a rule, and hands the caller a clear error message to catch and read.

#	CONCEPT	SAMPLE CODE
1	**Signal an error on purpose** `raise` Stop the function and send a clear message.	`raise ValueError("total books cannot be negative")`
2	**Catch it where you call** `except` A raised error is caught like any other.	`except ValueError as err:`

© Kalyan Reddy Daida

Errors: three kinds of bug, and when each one bites

The three bugs are told apart by WHEN they hit you

One stops the file before line 1. One stops it halfway. One never stops anything - it just **gives you the wrong answer**, which is why it is the dangerous one.

BEFORE ANYTHING RUNS — MID-RUN — NEVER CRASHES

BEFORE LINE 1

Syntax error

```
def calculate(books, students)
# the ":" is missing
```

```
SyntaxError – nothing runs, not even line 1
```

Python reads the WHOLE file first. Fix the spot the caret points at.

MID-RUN

Runtime error

```
    return books / studphts
# typo: students
```

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

Earlier lines already ran. Read the traceback bottom-up - the last line names the error.

NEVER

Logic bug

```
    return books * students
# should be /
```

```
Buggy result: 2000   (expected 5.0)
```

No error at all. Print the values to see them: debug: books=100 students=20

`print(f"{books=}")` prints the name and the value together - the fastest way to see what your code actually has, when nothing has crashed to tell you.

© Kalyan Reddy Daida

Errors: Debugging, Find and Fix Bugs

Debugging is finding why code misbehaves and fixing it, by reading error messages and printing values to see what really happened.

#	CONCEPT	SAMPLE CODE
1	**Logic bug** `{books=}` Wrong answer, no crash. Print the values to spot it.	`print(f"{books=} {students=}")`
2	**Runtime error** `NameError` Crashes mid-run. Let it crash, then read the traceback from the bottom up.	`NameError: name 'studphts' is not defined`

© Kalyan Reddy Daida

Errors: Debug in VS Code, Pause and Step Through Your Code

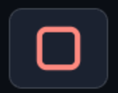
A **breakpoint** pauses your program on a line so you can look at every variable, then walk forward one line at a time instead of guessing.

#	CONCEPT	SAMPLE CODE
1	**A call chain** `Step Into` One function calls another, so Step Into goes down and Step Out comes back.	`each = calculate_books_per_student("100", "20")`
2	**Breakpoint in a loop** `Continue` It stops once per pass, so Continue walks you through the rows.	`for total_books, total_students in orders:`

© Kalyan Reddy Daida

Debugging in VS Code: what each toolbar button does

> The toolbar appears at the top the moment your program **pauses at a breakpoint**. Left to right, these are all six buttons. Five keys are the same everywhere; only **Restart** differs on Windows.

#	BUTTON	NAME	MAC	WINDOWS	WHAT IT DOES
1		**Continue**	`F5`	`F5`	Runs at full speed to the **next breakpoint**, or to the end. While running, this button becomes Pause.
2		**Step Over**	`F10`	`F10`	Runs this line, stops on the next. A function on the line runs fully, but you never go inside. **The one you press most.**
3		**Step Into**	`F11`	`F11`	Goes **inside** the function called on this line, so the next stop is that function's first line.
4		**Step Out**	`Shift + F11`	`Shift + F11`	Finishes the function you are inside and drops you back on the line that called it.
5		**Restart**	`Shift + Cmd + F5`	`Ctrl + Shift + F5`	**The green one.** Starts the same session again from line 1, keeping every breakpoint.
6		**Stop**	`Shift + F5`	`Shift + F5`	**The red one.** Ends the session. Your breakpoints stay set, so F5 picks up again later.

The dot in buttons 2, 3 and 4 is the function call: the arrow hops over it, goes down into it, or comes up out of it. Most bugs need only Step Over and the red square.

© Kalyan Reddy Daida

Section 10: Files, JSON, and Paths Concept Map

The full path through this section - we open here before each concept, then dive in.

16
CONCEPT DEMOS

FILES

01	Write and Read a Text File	C1 to C3	`with open()` `mode "w" / "r"` `f.read()` `\n newlines` `encoding="utf-8"`
02	Append and Read Lines	C1 to C4	`mode "a"` `for line in f` `.strip()` `readlines()` `seek(0)`
03	JSON Save and Load	C1 to C3	`json.dump` `json.load` `indent=2` `dict to file`
04	Paths with pathlib	C1 to C4	`Path() / join` `.name / .suffix` `.stem / .parent` `.exists()` `read_text() / write_text()`
05	Save Next to Your Script	C1 and C2	`__file__` `.parent` `mkdir()` `rmdir()`

© Kalyan Reddy Daida

Files: Keywords Covered

Section 10: Files, JSON and Paths - the keywords in bright green are the new ones we cover here.

Values

`True` `False` `None`

Logic & Membership

`and` `or` `not` `in` `is`

Conditionals

`if` `elif` `else`

Loops

`for` `while` `break` `continue`

Functions

`def` `return` `lambda`

Classes

`class`

Error Handling

`try` `except` `finally` `raise` `assert`

Modules & Imports

`import` `from` `as`

Scope

`global` `nonlocal`

Context Managers

`with`

Cleanup & Flow

`pass` `del`

Generators & Async

`yield` `async` `await`

Soft Keywords

`match` `case` `_` `type`

`keyword` New in this section `keyword` Covered earlier `keyword` Not covered yet

© Kalyan Reddy Daida

Files: the mode decides everything

One letter says what you are about to do to the file

You pass it as the second argument to **open**. Reach for **"w"** on a file you meant to keep and **the old contents are gone** - it does not warn you.

"w" write - creates it, or REPLACES it
everything already in the file is thrown away

"r" read - the file must already exist
this is the default if you pass no mode

"a" append - adds to the end, keeps the rest
coming up in Step-02

```
with open("demo_notes.txt", "w") as f:
    f.write("first line\n")
    f.write("second line\n")

with open("demo_notes.txt", "r") as f:
    content = f.read()
```

PRINTING content

```
first line
second line
```

with is why there is no `f.close()` here. It closes the file when the block ends - even if an error happens partway through.

Two habits worth forming now: always pass encoding="utf-8" so text reads the same on every computer, and tidy up a demo file with Path("demo_notes.txt").unlink().

© Kalyan Reddy Daida

Files: Write a Text File, Then Read It Back

A text file keeps your data on disk after the program ends; **open()** saves text to it and reads it back into your program.

#	CONCEPT	SAMPLE CODE
1	**Write a file** `with open` Mode "w" creates or replaces; with closes it for you.	`f.write("first line\n")`
2	**Read it back with mode** `"r"` Load the whole file as one string.	`content = f.read()`
3	**Clean up the file** `.unlink()` Last step: remove the demo file; missing_ok=True if it is already gone.	`Path("notes.txt").unlink()`

© Kalyan Reddy Daida

Files: append, and three ways to read lines

A log grows; it does not get rewritten

Mode `"a"` adds to the end and keeps what was there. That one letter is the difference between a log and a file you just destroyed.

`open(path, "a")`

keeps every old line, adds yours at the end - what a log needs

`open(path, "w")`

throws the old lines away, starts empty - only when you mean it

ONE LINE AT A TIME

```
for line in f:
    print(line.strip())
```

```
Kalyan logged in
Kalyan opened the report
Kalyan logged out
```

Never loads the whole file - the right choice for a big one. strip() drops the newline.

EVERY LINE AS A LIST

```
f.readlines()
```

```
['alpha\n', 'beta\n', 'gamma\n']
```

Watch the \n on every item. This is why line == "alpha" quietly fails.

CLEAN LINES AS A LIST

```
f.read().splitlines()
```

```
['alpha', 'beta', 'gamma']
```

Same list, no newlines. Usually the one you actually wanted.

Tidying up: Path("demo_log.txt").unlink() removes the file, and unlink(missing_ok=True) stays quiet when it is already gone instead of raising FileNotFoundError.

© Kalyan Reddy Daida

Files: Append Lines, Then Read Line by Line

Append mode adds new lines to the end of a file without erasing the old ones, then you read them back one line at a time.

#	CONCEPT	SAMPLE CODE
1	**Append mode** `"a"` Adds new lines to the end and keeps the old entries.	`open("demo_log.txt", "a")`
2	**Read line by line** `for line in f` One line at a time, so the whole file never loads.	`print("line:", line.strip())`
3	**Read every line at once** `.readlines()` The whole file as a list: costs memory, buys len() and indexing.	`all_lines = f.readlines()`
4	**Rewind the cursor** `seek(0)` A second read returns nothing until you rewind to the start.	`f.seek(0)`

© Kalyan Reddy Daida

Files: json - the s means string

Four names, one letter apart - here is the rule that settles it

The **s** stands for **string**. No **s** means it works on a **file**. That is the only thing separating the four.

	PYTHON DICT -> JSON	JSON -> PYTHON DICT
to a STRING with s	**`json.dumps`** gives you the JSON as text, no file `{"name": "Kalyan", "age": 30,` `"languages": ["Python", "Bash"]}`	**`json.loads`** turns that text back into a dict `back["languages"][0] -> Python`
to a FILE no s	**`json.dump`** writes the dict into an open file `json.dump(person, f, indent=2)`	**`json.load`** reads the file back as a real dict `{'name': 'Kalyan', 'age': 30,` `'languages': ['Python', 'Bash']}`

Look at the quotes

JSON text uses "double quotes"; a Python dict prints with 'single' ones

type(text) is str

dumps gives you a string you can send or log, not a saved file

Use dump / load when the data belongs on disk - settings, a saved record. Use dumps / loads when it is going somewhere else, like an API request or a log line.

© Kalyan Reddy Daida

Files: Save and Load Data with JSON

JSON is a simple text format for structured data - dicts, lists, strings, and numbers - so you can save it to a file and read it back later.

#	CONCEPT	SAMPLE CODE
1	**Dict to string and back** `json.dumps()` `json.loads()` Turn a dict into a JSON string for sending or logging.	`text = json.dumps(person)`
2	**Save a dict to a file** `json.dump()` Write a Python dict to a file as readable JSON.	`json.dump(person, f, indent=2)`
3	**Load a file into a dict** `json.load()` Read the JSON file back into a Python dict.	`loaded = json.load(f)`

© Kalyan Reddy Daida

Files: one path, four parts

Build paths with / and never type a slash again

A `Path` knows it is a path, so joining with `/` makes the right separator on Windows, Mac and Linux. **Never glue path strings together by hand.**

```
from pathlib import Path
p = Path("reports") / "app_files" / "demo_note.txt"      # reports/app_files/demo_note.txt
```

reports/app_files/demo_note.txt

p.parent	p.name	p.stem	p.suffix
reports/app_files	demo_note.txt	demo_note	.txt

p.exists()
asks the operating system if it is really there - this one gives `False`, it was never created

write_text() / read_text()
a whole small file in one line, no `with` block needed

A path object is not a string - it knows its own parts. p.name gives you the filename without any splitting, searching or slicing.

© Kalyan Reddy Daida

Files: Paths with pathlib

pathlib represents a file or folder path as an object you can build, inspect, and use the same way on Windows, Mac, and Linux.

#	CONCEPT	SAMPLE CODE
1	**Build a path** `/` Join folder and file parts with the / operator.	`p = Path("reports") / "app_files" / "demo_note.txt"`
2	**Parts of a path** `.name` `.stem` Read each piece: .name, .stem, .suffix, .parent.	`print("name:  ", p.name)`
3	**Write and read** `.write_text()` `.read_text()` Save or read a whole small file in one line.	`note.write_text("saved with one line\n")`
4	**Does it exist?** `.exists()` False before you write it, True after, False once removed.	`print(p.exists())`

© Kalyan Reddy Daida

Files: Save Next to Your Script

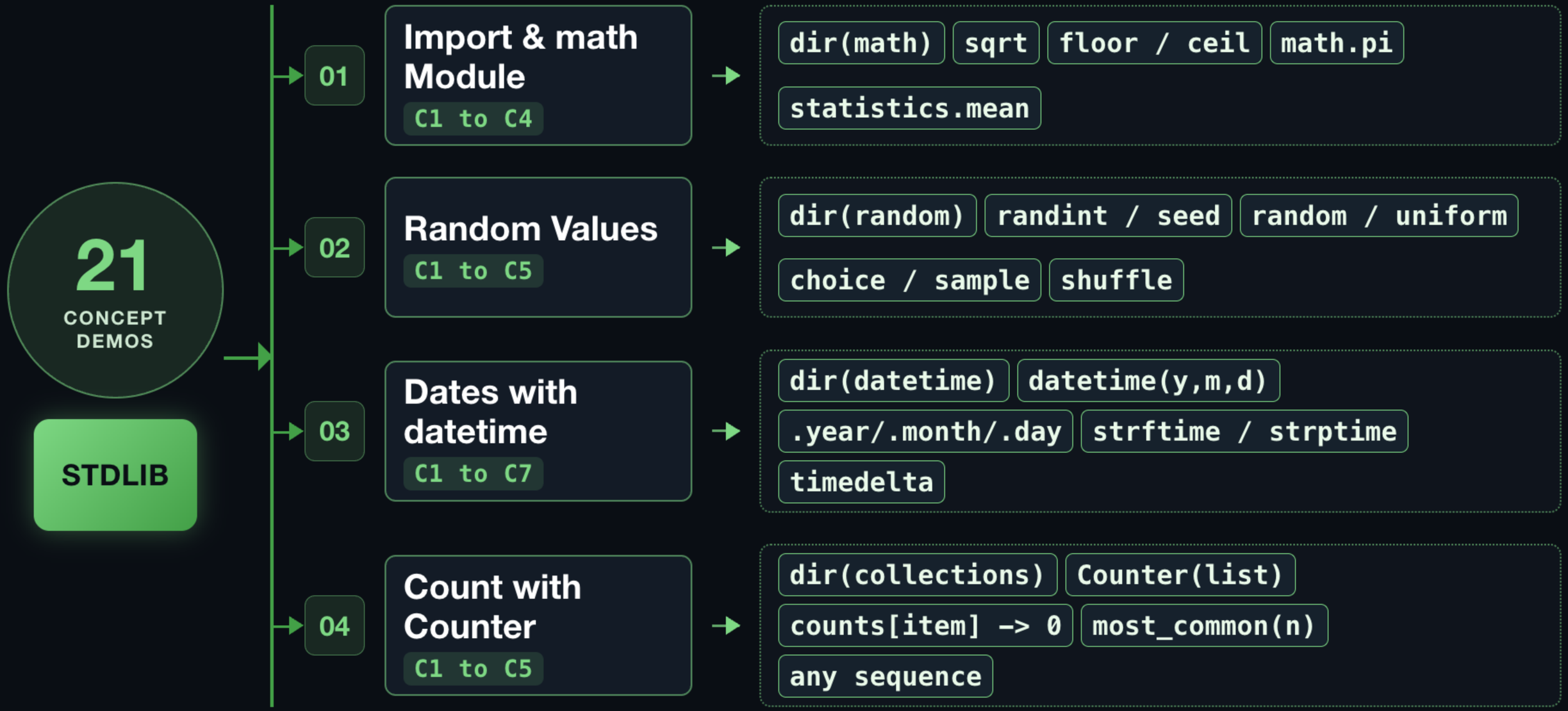
A bare file name lands wherever you RAN the program. **__file__** is the path of the running script, so a path built from it always points to the same place.

#	CONCEPT	SAMPLE CODE
1	**Pin a path to your code** `__file__` A bare name follows the terminal; this follows your script.	`Path(__file__).parent / "notes.txt"`
2	**A folder beside your code** `mkdir()` Make the folder, then write the file into it.	`folder.mkdir(parents=True, exist_ok=True)`

© Kalyan Reddy Daida

Section 11: Standard Library Essentials Concept Map

The full path through this section - we open here before each concept, then dive in.



© Kalyan Reddy Daida

Std Library: Import a Module and Use math

The **standard library** is a set of ready-made modules that come with Python; **import** brings a module like **math** into your program so you can use its tools.

#	CONCEPT	SAMPLE CODE
1	**Square root** `math.sqrt()` Get the square root of a number.	`math.sqrt(144)`
2	**Round down and up** `floor()` `ceil()` Round to the nearest whole number, down or up.	`math.floor(4.7), math.ceil(4.2)`
3	**The pi constant** `math.pi` Use pi without typing the digits yourself.	`math.pi`
4	**Average of a list** `statistics.mean()` Get the mean of a list of numbers.	`statistics.mean(scores)`

© Kalyan Reddy Daida

Std Library: Random Values with random

The **random** module produces random values - whole numbers, decimals, a random pick from a list, or a shuffled order.

#	CONCEPT	SAMPLE CODE
1	**Repeatable runs** `seed()` Fix the sequence so every run matches.	`random.seed(42)`
2	**Whole number in a range** `randint()` An integer from a to b, both ends included.	`random.randint(1, 6)`
3	**Random decimal** `random()` A float from 0.0 up to but not 1.0.	`random.random()`
4	**Pick from a list** `choice()` Grab one random item, no index needed.	`random.choice(names)`
5	**Shuffle in place** `shuffle()` Reorder the list itself, not a copy.	`random.shuffle(deck)`

© Kalyan Reddy Daida

Std Library: Dates with datetime

The **datetime** module stores a specific date and time as an object you can read, format, and do date math on.

#	CONCEPT	SAMPLE CODE
1	**Build a date** `datetime()` Store a specific date you can use in code.	`event = datetime(2026, 6, 17)`
2	**Read date parts** `.year` Pull the year, month, or day number out.	`print("year:", event.year)`
3	**Format as text** `strftime()` Show a date as friendly, readable text.	`event.strftime("%d %B %Y")`
4	**Add or subtract time** `timedelta()` Add days to a date to jump forward in time.	`event + timedelta(days=7)`
5	**Parse text to date** `strptime()` Read a date out of a text string.	`datetime.strptime("2026-06-17", "%Y-%m-%d")`
6	**Days between dates** Subtract two dates; read .days for the gap.	`(event - today).days`

© Kalyan Reddy Daida

Std Library: Count Items with Counter

Counter is a collections type that counts how many times each item appears in a sequence - all in one step.

#	CONCEPT	SAMPLE CODE
1	**Count every item** `Counter()` One step tallies each item in a sequence.	`counts = Counter(votes)`
2	**Look up a count** `counts[key]` A missing key returns 0, not an error.	`print("votes for Joe:", counts["Joe"])`
3	**Top items** `most_common(n)` Returns the n highest counts, biggest first.	`counts.most_common(2)`
4	**Count any sequence** `Counter("...")` Even counts each letter in a word.	`letter_counts = Counter("banana")`

© Kalyan Reddy Daida

Section 12: Virtual Environments and Packaging Concept Map

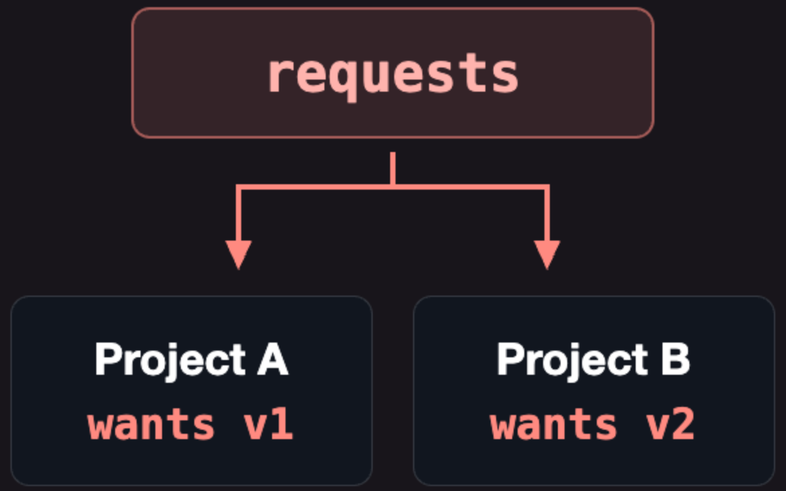
The path through Section 12: the habits that turn loose scripts into an isolated, installable project.



© Kalyan Reddy Daida

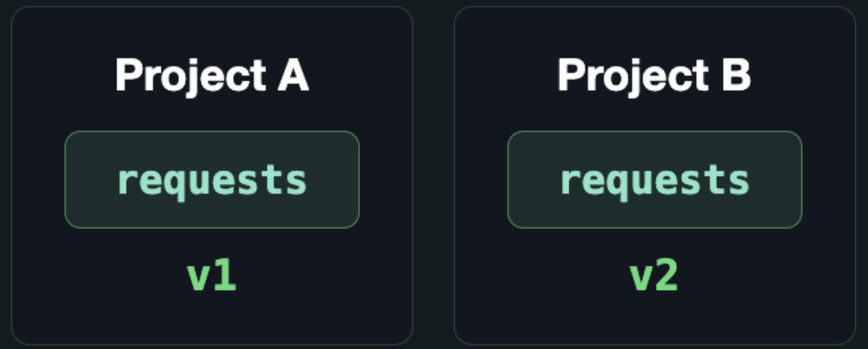
Why each project needs its own venv (virtual environment)

One shared Python



Only one can win

One Python each



Both work

© Kalyan Reddy Daida

What a venv actually is

A folder inside your project

```
my_project/
    .venv/
    its own python and packages
    app.py
    number_utils/
```

That is all it is. Delete the folder and it is gone.

What activating changes

```
source .venv/bin/activate
```

points your terminal at that folder

before `python` → system Python

after `python` → `.venv/`

So `pip install` lands inside the project, not on your whole machine.

© Kalyan Reddy Daida

Virtual Environments: what you will do in this step

Build one project folder that carries its own private Python, prove to yourself that it changed, then tidy up after. Every line below is typed in the **terminal**, not in a **.py** file.

#	WHAT YOU DO	COMMAND
1	**Create the project and its venv** `venv` A folder for the project, and a private Python inside it.	`python3 -m venv .venv`
2	**Check which Python you are on** Look BEFORE you activate, so you have something to compare.	`which python3`
3	**Activate it** `activate` Your prompt then carries the name in front of it.	`source .venv/bin/activate`
4	**See what changed** The same question now answers with a path inside your project.	`which python`
5	**Look inside a brand new venv** Nearly empty, which is the win. deactivate steps back out.	`pip list`
6	**The name is your choice** .venv is only a convention, not a rule.	`python3 -m venv my_environment`
7	**A second venv, then clean up** One folder can hold two. Leave one, enter the other, delete the spare.	`rm -rf .venv2`

© Kalyan Reddy Daida

Installing Packages: what you will do in this step

Add a package someone else wrote, record exactly what you installed, then DELETE your environment and rebuild it from that record. Every line is typed in the **terminal**.

#	WHAT YOU DO	COMMAND
1	**Fresh project, activate its venv** pip installs into whichever Python is active. It will not warn you.	`mkdir step02_my_project`
2	**Install a third-party package** `requests` The first command in the course that needs the internet.	`pip install requests`
3	**See what you got** You asked for one package and five arrived. The extras are its dependencies.	`pip list`
4	**Record what is installed** Every package pinned to its exact version, written to a file.	`pip freeze > requirements.txt`
5	**Rebuild in a fresh environment** The proof: an empty venv becomes the same five packages.	`pip install -r requirements.txt`
6	**Clean up** Delete the spare and go back to the project venv.	`rm -rf .venv2`

© Kalyan Reddy Daida

The Main Guard: what you will do in this step

See what goes wrong when one file imports another, then fix it with **if __name__ == "__main__"**. It gets a small demo project of its own, so nothing else is in the way.

#	WHAT YOU DO	SAMPLE CODE
1	**Meet Python's special names** A fresh demo folder and venv, then print what Python sets.	`print(__name__)`
2	**Write app.py, without the guard** Prints its __name__, then a function and a bare main().	`main()`
3	**Write runner.py** A second file that only wants one function.	`from app import double`
4	**Run it and watch it break** main() runs, even though runner.py never asked for it.	`python runner.py`
5	**Add the guard** Change only the last line of app.py.	`if __name__ == "__main__":`
6	**Run it both ways** Imported, main() stays quiet. Run directly, it still runs.	`python app.py`

© Kalyan Reddy Daida

Your Own Package: what you will do in this step

Put your own functions in their own file, group them into a package, and import them from another file. The package files you write in **VS Code**.

#	WHAT YOU DO	SAMPLE CODE
1	**Fresh project and its venv** A new folder beside the last one, with its own venv.	`mkdir step04_my_project`
2	**Make the package and write the module** A folder, an empty __init__.py, and the functions.	`def double(n):`
3	**Create app.py and watch it break** ImportError: the front door is still empty.	`from number_utils import double`
4	**Open the front door** One import line in __init__.py hands the functions out.	`from number_utils.calculations import double`
5	**Run it again** Same app.py, now it works.	`python app.py`
6	**Import everything with *** Grabs every name the package hands out.	`from number_utils import *`
7	**Limit * with __all__** Named imports are not affected.	`__all__ = ["double"]`

© Kalyan Reddy Daida

Module vs Package: one file, and a folder of files

The mental picture

PACKAGE `number_utils/`
a folder that organizes related modules

`__init__.py` the front door: what the package hands out

MODULE `calculations.py`
`double()` `percentage()`

MODULE `another_module.py`
a package can hold as many related modules as you need

PACKAGE holds **MODULES** hold **FUNCTIONS**

In one line each

MODULE one reusable `.py` file
it holds the functions: `calculations.py`

PACKAGE a folder of related modules
it groups the modules: `number_utils/`

FUNCTION the code you actually call
it lives inside a module: `double()`

Two ways to get double()

```
from number_utils.calculations import double
```
reach into the module

```
from number_utils import double
```
through the front door, once __init__.py hands it out

A module is one reusable .py file. A package is a folder that organizes related modules.

© Kalyan Reddy Daida

Environments: what the -e in pip install -e . does

It does not copy your code anywhere

`pyproject.toml` describes your project, then `-e` (editable) drops a **pointer** into the venv. Your package becomes importable from any folder, and it still runs from **your own source files**.

FROM ANOTHER FOLDER, BEFORE

```
from number_utils import double
ModuleNotFoundError:
No module named 'number_utils'
```

`pip install -e .` →

FROM ANOTHER FOLDER, NOW IT WORKS

```
from number_utils import double
print(double(25))
50
```

What actually landed in the venv

pip wrote a small pointer file, not your code:

```
site-packages/__editable__.number_utils-0.1.0.pth
```

and your package still runs from where you wrote it:

```
number_utils.__file__
  /step05_my_project/number_utils/__init__.py
```

So your edits are live, no reinstall

```
edit number_utils/calculations.py
(double: n * 2 becomes n * 3), run again:
75
pip was never run a second time
```

Drop the `-e` and pip copies your code into the venv instead, so every edit needs a reinstall before it shows up. **Use -e while you are building the package.**

© Kalyan Reddy Daida

Packaging Your Project: what you will do in this step

Make your own package importable from ANY folder, the way **requests** is, by describing the project in a **pyproject.toml** and installing it into the venv.

#	WHAT YOU DO	SAMPLE CODE
1	**Start from Step-04's code** A fresh folder and venv, with the package copied in.	`mkdir step05_my_project`
2	**Watch it break** Step out of the folder and the import fails.	`python -c "from number_utils import double"`
3	**Describe your project** `pyproject.toml` Its name, its version, and how to build it, in one file.	`name = "number_utils"`
4	**Install it into the venv** The dot means the project in the folder you are standing in.	`pip install -e .`
5	**Use it from a different project** A brand new folder with one file imports your package.	`python report.py`
6	**Change the code, run again, install nothing** Edit the module and the other project changes at once.	`return n * 3`
7	**See what pip actually did** A pointer file, not a copy. Your own source folder still runs.	`pip show number_utils`

© Kalyan Reddy Daida

Section 13: Classes and OOP Basics Concept Map

The full path through this section: we open here before each concept, then dive in.

25
CONCEPT DEMOS

CLASSES

01	Why Objects, First Class C1 to C4	loose variables	class blueprint	p1 = Product()	0x... address
02	__init__, self, Attributes C1 to C4	__init__ auto-runs	self IS the object	parameters	self.name
03	Class vs Instance Attrs C1 to C4	shared class attr	Product.store_name	per-object data	instance wins
04	Methods C1 to C4	def inside class	method changes data	return vs act	missing () gotcha
05	Readable Objects (__str__, __repr__) C1 to C6	address is back	plain method ignored	repr default too	__str__ + __repr__
06	Class & Static Methods C1 to C3	@classmethod (cls)	from_dict	@staticmethod	is_valid_price

© Kalyan Reddy Daida

OOP: Keywords Covered

Section 13: Classes and OOP Basics. The keywords in bright green are the new ones we cover here.

Values

`True` `False` `None`

Logic & Membership

`and` `or` `not` `in` `is`

Conditionals

`if` `elif` `else`

Loops

`for` `while` `break` `continue`

Functions

`def` `return` `lambda`

Classes

`class`

Error Handling

`try` `except` `finally` `raise` `assert`

Modules & Imports

`import` `from` `as`

Scope

`global` `nonlocal`

Context Managers

`with`

Cleanup & Flow

`pass` `del`

Generators & Async

`yield` `async` `await`

Soft Keywords

`match` `case` `_` `type`

`keyword` New in this section `keyword` Covered earlier `keyword` Not covered yet

© Kalyan Reddy Daida

Classes: why objects at all?

Count the variables before you meet the syntax

Two products already need **six** loose variables that must be kept in step by hand. Ten products need thirty. A class is how you stop counting.

Loose variables

```
product1_name = "Laptop"
product1_price = 1000
product1_stock = 5
product2_name = "Mouse"
product2_price = 25
product2_stock = 200
```

Nothing ties these together. Delete one, misspell one, or pass the wrong pair to a function and Python will not notice.

One blueprint

```
class Product:
    store_name = "Pyzon Store"

p1 = Product()
p2 = Product()
print(p1.store_name)   # Pyzon Store
```

Describe a product once, then make as many as you like. Calling the class name builds an object.

6 variables for 2 products

30 variables for 10 products

1 class, for any number

Honest about what is still missing

No product has its OWN name or price yet -> Step-02

print(p1) shows <__main__.Product object at 0x...> -> Step-05

© Kalyan Reddy Daida

OOP: Why Objects? Your First Class

A **class** is a blueprint that describes one kind of thing, and an **object** is one real item built from that blueprint, like a **Product** blueprint and the two products Laptop and Mouse made from it.

#	CONCEPT	SAMPLE CODE
1	**The problem** `loose variables` Two products already need six separate variables.	`product1_name = "Laptop"` `product1_price = 1000`
2	**Write a blueprint** `class` Describe "a product" once and reuse it for every product.	`class Product:` `    store_name = "Pyzon Store"`
3	**Create an object** `Product()` Call the class name like a function to build one instance.	`p1 = Product()` `print(p1.store_name)`
4	**What is still missing** `print(p1)` Both share the store name, but neither has its own name yet.	`print(p1)` `print(p2)`

© Kalyan Reddy Daida

Classes: the __init__ Method and the self Object

Not a keyword, not magic: just a name for the object being worked on

`__init__` runs by itself the moment you create an object, and **self is that object**.

Print it from inside, print it from outside

```
class Product:
    def __init__(self):
        print("inside  __init__:", self)
inside  __init__: <__main__.Product object at 0x1043b46e0>
```

```
p1 = Product()
print("outside __init__:", p1)
outside __init__: <__main__.Product object at 0x1043b46e0>
```

The same address, both times: one object, two names

the digits differ on every run; what matters is that they MATCH

```
def __init__(self, name, price, stock):
    self.name = name
    self.price = price
    self.stock = stock
```

```
now each object owns its data
p1 = Product("Laptop", 1000, 5)
p2 = Product("Mouse", 25, 200)
print(p1.name, p1.price, p1.stock)
print(p2.name, p2.price, p2.stock)
Laptop 1000 5
Mouse 25 200
```

self.name = name reads oddly until you see it: the right side is the parameter that was passed in, the left side is where it gets stored on this object.

© Kalyan Reddy Daida

OOP: Give Each Object Its Own Data

__init__ is the setup method Python runs for you when an object is created, **self** is that object, and an **instance attribute** is data stored on it, so **Product("Laptop", 1000, 5)** makes a product with its own name, price, and stock.

#	CONCEPT	SAMPLE CODE
1	**Setup method** `__init__` Python runs it for you the moment an object is created.	`class Product:` `  def __init__(self):`
2	**The object itself** `self` Inside and outside print the SAME object.	`print("inside  __init__:", self)` `print("outside __init__:", p1)`
3	**Start with own data** `__init__(self, name)` Add a parameter, then pass a value when you create the object.	`p1 = Product("Laptop")` `p2 = Product("Mouse")`
4	**Each object's own data** `self.name = name` Data stored on self belongs to THIS object; each object keeps its own copy.	`self.name = name` `self.price = price`

© Kalyan Reddy Daida

Classes: Class Attributes vs Instance Attributes

Ask where the value should LIVE

Every product is sold by the same store, so the store name goes on the **class**, written once. A name, a price and a stock differ per product, so they go on **each object**.

CLASS ATTRIBUTE: one copy, shared by every product

```
class Product:
    store_name = "Pyzon Store"
```

```
print(Product.store_name)   # Pyzon Store
print(p1.store_name)        # Pyzon Store
print(p2.store_name)        # Pyzon Store
```

from the class or any object: the same shared value

INSTANCE ATTRIBUTES: each object keeps its own name, price and stock

```
p1 = Product("Laptop", 1000, 5)
p1.price = 900
print(p1.name, p1.price)
```

```
Laptop 900
```

only the Laptop's price changed

```
p2 = Product("Mouse", 25, 200)

print(p2.name, p2.price)
```

```
Mouse 25
```

untouched: its price lives on p2

SAME NAME ON BOTH? The object's own value wins, for that object only

```
p2.store_name = "ABC Sellers"
```

```
p2.store_name
ABC Sellers
```

```
p1.store_name
Pyzon Store
```

```
Product.store_name
Pyzon Store
```

© Kalyan Reddy Daida

OOP: Class vs Instance Attributes

A **class attribute** is written in the class body and is shared by every object, while an **instance attribute** is set on **self** and belongs to one object only: every Product shares the same **store_name**, but Laptop and Mouse each keep their own **price** and **stock**.

#	CONCEPT	SAMPLE CODE
1	**Class attribute** `store_name` Write it once in the class body; every object shares it.	`class Product:` `  store_name = "Pyzon Store"`
2	**Read it three ways** `Product.store_name` The class or any object gives the same shared value.	`print(Product.store_name)` `print(p1.store_name)` `print(p2.store_name)`
3	**Instance attributes stay separate** `p1.price` Each object holds its own values; changing one leaves the rest alone.	`p1.price = 900` `print(p2.price)  # still 25`
4	**Instance attribute wins** `p2.store_name` Set the same name on an object and that object gets its own value.	`p2.store_name = "ABC Sellers"` `print(p2.store_name)  # instance attribute wins`

© Kalyan Reddy Daida

Classes: Adding Behavior with Methods

A method is just a function that gets the object handed to it

Its first parameter is `self`, so it can read **this** object's data and change it. You call it on an object with a dot, like `p1.describe()`.

Read the object's data

```
def describe(self):
    return f"{self.name} costs ${self.price}"
```

```
print(p1.describe())   # Laptop costs $1000
print(p2.describe())   # Mouse costs $25
```

Change the object's data

```
def restock(self, qty):
    self.stock += qty

def sell(self, qty):
    self.stock -= qty
```

```
print(p1.stock)   # 5
p1.restock(10)
p1.sell(2)
print(p1.stock)   # 13
```

The trap: without the parentheses, nothing runs and nothing complains

```
print(p1.describe)
<bound method Product.describe of ...>
```

you only named the method; you never ran it

```
print(p1.describe())
Laptop costs $1000
```

the () is what actually calls it

A method that RETURNS gives you something to keep working with, like `label.upper()` after `label = p1.describe()`. One that only ACTS, like `sell()`, hands back `None`, and that is correct.

© Kalyan Reddy Daida

OOP: Adding Behavior with Methods

A **method** is a function written inside a class, and its first parameter **self** is the object itself, so **describe()** and **sell()** can use and change that product's own data.

#	CONCEPT	SAMPLE CODE
1	**Method with self** `self` A function inside the class; self reaches that object's own data.	`def describe(self):` `  return f"{self.name} costs ${self.price}"`
2	**Change the data** `self.stock -= qty` A method can also update the object's own data.	`def sell(self, qty):` `  self.stock -= qty`
3	**Return or act** `return` `None` describe hands text back; sell only acts, so it returns None.	`label = p1.describe()` `print(p1.sell(2))  # returns None`
4	**The () runs it** `()` Without () you only name the method; with () you run it.	`print(p1.describe)  # not the text` `print(p1.describe())  # the text`

© Kalyan Reddy Daida

Classes: Readable Objects with __str__ and __repr__

print() and repr() each look for one exact method name

`print(p1)` uses **__str__** and `repr(p1)` uses **__repr__**. A method with any other name, like `describe()`, is never used by them.

A normal method: print() ignores it

```
def describe(self):
    return f"{self.name} - ${self.price} ({self.stock} in stock)"
```

```
print(p1)
<__main__.Product object at 0x...>
print(p1.describe())   works only when WE call it
Laptop - $1000 (5 in stock)
```

__str__: same body, exact name, the friendly form

```
def __str__(self):
    return f"{self.name} - ${self.price} ({self.stock} in stock)"
```

```
print(p1)
Laptop - $1000 (5 in stock)
```

__repr__: its own exact name, the exact form for developers

```
def __repr__(self):
    return f"Product(name={self.name!r}, price={self.price!r}, stock={self.stock!r})"
```

```
print(repr(p1))   before adding __repr__
<__main__.Product object at 0x...>
print(repr(p1))   after adding __repr__
Product(name='Laptop', price=1000, stock=5)
```

A list of products shows __repr__, one product shows __str__

```
print(products)     [Product(name='Laptop', price=1000, stock=5), Product(name='Mouse', price=25, stock=200)]
print(products[0])  Laptop - $1000 (5 in stock)
```

A list shows each item with __repr__, so printing a list gives you the exact form. That is what you want when you are debugging.

© Kalyan Reddy Daida

OOP: Readable Objects with __str__ and __repr__

Two more **dunder methods**, both about SHOWING an object. **print()** calls **__str__** for a friendly line; **repr()** and a list of objects use **__repr__** for the exact one. The name must match exactly; a normal method will not do.

#	CONCEPT	SAMPLE CODE
1	**Plain object prints an address** `print(p1)` With no __str__, Python shows the type and a memory address.	`p1 = Product("Laptop", 1000, 5)` `print(p1)`
2	**A normal method is ignored** `describe()` It works when YOU call it, but print() never looks for it.	`def describe(self):` `print(p1.describe())`
3	**Friendly form** `__str__` Same body, exact name, and now print() uses it.	`def __str__(self):` `  return f"{self.name} - ${self.price}"`
4	**repr() has a default too** `repr(p1)` On a plain object, str and repr fall back to the SAME address.	`print(p1)` `print(repr(p1))`
5	**Exact developer form** `__repr__` A precise line that names each field, for inspecting objects.	`return f"Product(name={self.name!r})"`
6	**A list shows __repr__** `not __str__` Print a list to debug and you get the exact form, not the friendly one.	`products = [p1, p2]` `print(products)`

© Kalyan Reddy Daida

Classes: what a decorator is, and why we need one

A decorator is a line that starts with @, written just above a def

It does not run your code. It tells Python **how that method should be used**. `@classmethod` and `@staticmethod` are two that Python already gives you.

The shape

```
@classmethod
def from_dict(cls, data):
    return cls(data["name"], ...)
```

The @ line decorates the method written under it. Nothing else changes.

What it decides: who arrives first

`no decorator`	`def describe(self):`	Python sends the object
`@classmethod`	`def from_dict(cls, data):`	Python sends the class
`@staticmethod`	`def is_valid_price(price):`	Python sends nothing

Why we need it: a helper on the class must work from the class AND from an object

```
def is_valid_price(price):     # no decorator
Product.is_valid_price(1000)   True
p1.is_valid_price(1000)        TypeError: takes 1 positional
argument but 2 were given
```

Called on an object, Python puts p1 in the first slot, so price gets 1000 as a second value.

```
@staticmethod
def is_valid_price(price):
Product.is_valid_price(1000)   True
p1.is_valid_price(1000)        True
```

The decorator tells Python to send nothing first, so both calls mean the same thing.

A decorator is a label Python reads before the method. In this step it decides one thing: what arrives first, the object, the class, or nothing.

© Kalyan Reddy Daida

Classes: Class Methods and Static Methods

Look at the decorator and the first parameter: they tell you what the method receives

A normal method receives **self**, one object. A `@classmethod` receives **cls**, the class itself. A `@staticmethod` receives **neither**.

cls @classmethod receives the class and builds the object

```
@classmethod
def from_dict(cls, data):
    print("cls is:", cls)
    return cls(data["name"], data["price"], data["stock"])
```

```
p1 = Product.from_dict(data)
cls is: <class '__main__.Product'>
print(p1.name, p1.price, p1.stock)
Laptop 1000 5
```

self Both ways give a real Product that normal methods work on

```
def describe(self):
    return f"{self.name} - ${self.price}"

p1 = Product("Mouse", 25, 200)
p2 = Product.from_dict({"name": "Laptop", "price": 1000, "stock": 5})
```

```
print(p1.describe())   made directly
Mouse - $25
print(p2.describe())   made by from_dict
Laptop - $1000
```

neither @staticmethod is a related helper: no self, no cls

```
@staticmethod
def is_valid_price(price):
    return price > 0
```

```
print(Product.is_valid_price(1000))
True
print(Product.is_valid_price(-5))
False
```

Which one do I write? Need the class itself, like building a new object? Use @classmethod, because it gets `cls`. Just a related helper that needs no object and no class? Use @staticmethod.

© Kalyan Reddy Daida

OOP: Class Methods and Static Methods

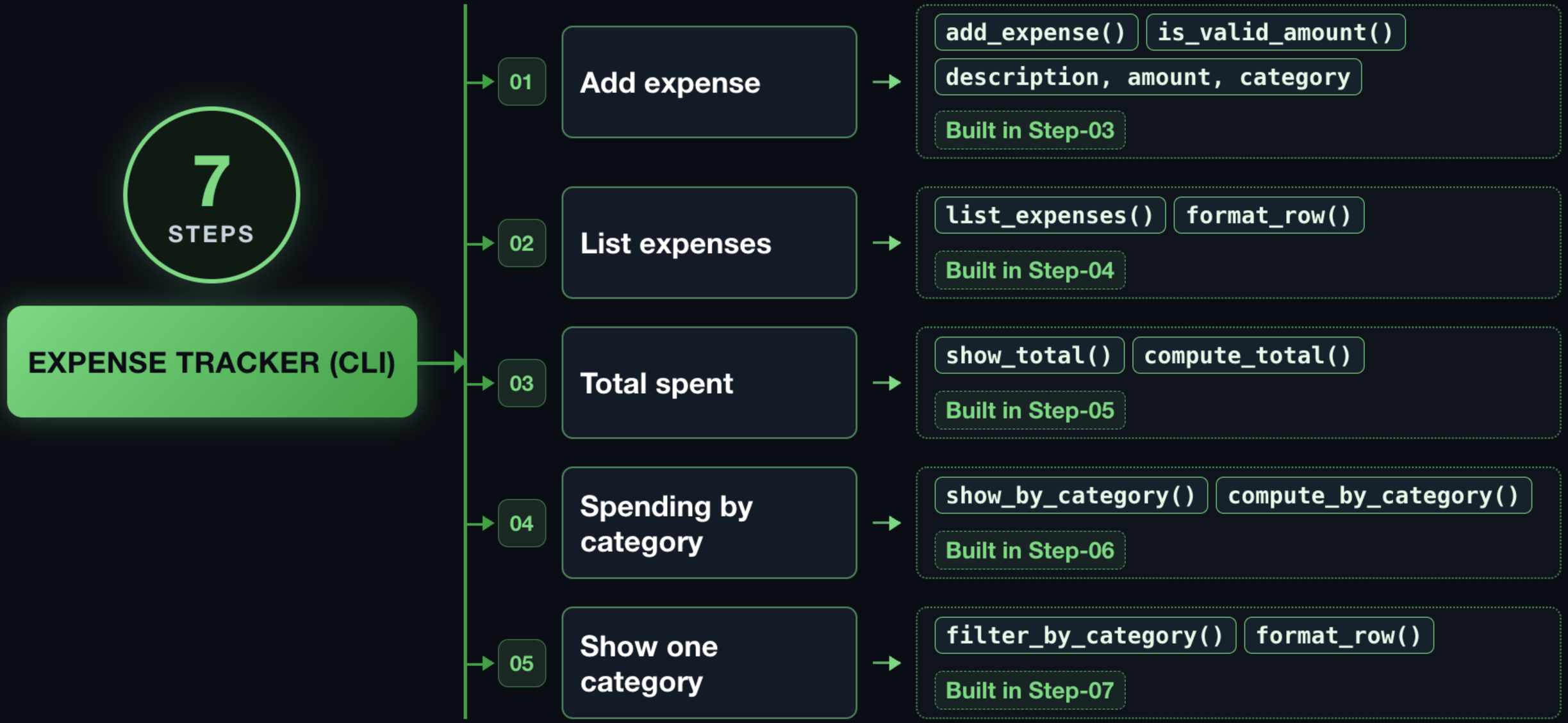
Most methods work on one object through **self**. A **@classmethod** receives the class as **cls** (great for an alternate constructor like **from_dict**); a **@staticmethod** takes neither **self** nor **cls**, making it a related helper like **is_valid_price**.

#	CONCEPT	SAMPLE CODE
1	**classmethod builds an object** `cls` It gets the CLASS as cls, so cls(...) makes a new Product.	`@classmethod` `def from_dict(cls, data):` `p1 = Product.from_dict(data1)`
2	**staticmethod, no self/cls** `@staticmethod` A related helper that lives on the class but needs no object.	`@staticmethod` `def is_valid_price(price):`
3	**Which one do I write?** `cls` `neither` Need the class itself -> classmethod; just a helper -> staticmethod.	`Product.from_dict(data)` `Product.is_valid_price(data["price"])`

© Kalyan Reddy Daida

Milestone Project-1: CLI Expense Tracker - What You Will Build?

The five menu actions, and the functions that make each one work.



© Kalyan Reddy Daida

Project 1: Concept Map

Almost everything in this project you have already learned. This map is the checklist we work through, feature by feature.

M1 CONCEPTS

43
CORE CONCEPTS REVISITED

01	Functions	`def` and call a function · parameter `expenses` · `return` a value
02	Lists & Dicts	`expenses = []` · one dict per expense · `expense["amount"]`
03	Loops	`while True` menu loop · `for` over `expenses`
04	Conditionals	`if / elif / else` · one branch per menu choice
05	f-strings	`:<20 :>10` aligned table · `:,.2f` format spec
06	input()	`input("Choose 1-6: ")` · `is_valid_amount()` checks the amount
07	Modules	`from ... import` · one helper per file, `f1` to `f4`

© Kalyan Reddy Daida

Project 1: Which Folder Is Which

Four folders in **milestone-projects/milestone-project-1/**. One of them is yours to type in.

`build-incremental`	THE ANSWER	The finished files, **Steps 01 to 06**. Your answer key.
`build-incremental-lecture`	YOU BUILD HERE	Six empty files. Build **Steps 01 to 06** here.
`solution`	ONE FILE	The whole program in `expense_tracker.py`. Compare at **Step-07**.
`warmup`	OPTIONAL	Six short drills, one per skill. An optional extra.

IN SHORT Everything you type goes in `build-incremental-lecture`. Compare with `build-incremental` as you go.

© Kalyan Reddy Daida

Step-01: 1.1 The Menu Shell in main.py

main.py is the menu driver: it shows the menu, reads a choice, and calls one of five actions.

CONCEPTS USED

- **Output: `print()`** — Section 02
- **User input: `input()` and `.strip()`** — Sections 02, 03
- **Conditionals: `if` / `elif` / `else`** — Section 04
- **Lists: `expenses = []`** — Section 05
- **Loops: `while True` and `break`** — Section 06
- **Functions: `def` and call** — Section 07
- **Modules: `from ... import`** — Section 07



WHAT WE BUILD

- **`show_menu()`** — prints the banner and six options
- **`main()`** — repeats the menu until you choose 6
- **`expenses`** — the one list every action uses
- **Choices 1 to 5** — call the five actions
- **Choice 6** — prints Goodbye! and stops
- **Any other input** — asks for a number from 1 to 6
- **Five imported names** — the actions from `f5_actions.py`

© Kalyan Reddy Daida

Step-01: 1.2 Five Stub Actions in f5_actions.py

A **stub** is a placeholder that only prints which option you picked and the current list, so the whole menu can run.

CONCEPTS USED

Functions: `def` and call
Section 07

Parameters: `(expenses)`
Section 07

Modules: your own `.py` file
Section 07

Output: `print(a, b)`
Section 02

Escape sequence: `\n`
Section 02



WHAT WE BUILD

`add_expense()`
stub for choice 1

`list_expenses()`
stub for choice 2

`show_total()`
stub for choice 3

`show_by_category()`
stub for choice 4

`filter_by_category()`
stub for choice 5

`Expenses: []`
each stub prints the shared list

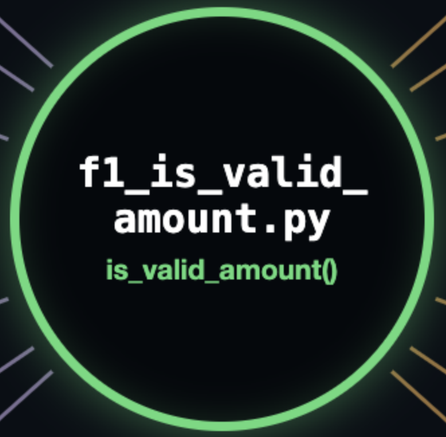
© Kalyan Reddy Daida

Step-02: 2.1 Validate the Amount in f1_is_valid_amount.py

is_valid_amount() checks the typed text with string methods first, then **float()**: True for a positive number, False for anything else.

CONCEPTS USED

- **String methods: `.strip()`, `.split()`**
  Sections 03, 05
- **String method: `.isdecimal()`**
  New in this project
- **Lists: `len()` and `[0]`**
  Section 05
- **Conditionals: `if` / `elif` / `else`**
  Section 04
- **Logical operators: `and`, `not`**
  Section 04
- **Type conversion: `float()`**
  Section 02
- **Return values: `return`**
  Section 07



WHAT WE BUILD

- **`is_valid_amount()`**
  returns `True` or `False`
- **Spaces trimmed**
  `" 12 "` is read as `12`
- **Whole number**
  `12` passes
- **One dot**
  `12.50` passes
- **Not digits**
  `-5`, `abc` and `1.2.3` fail
- **Not above zero**
  `0` and `0.00` fail
- **Safe `float()`**
  runs only after the digit check

© Kalyan Reddy Daida

Step-02: 2.2 Wire Up the Add Action in f5_actions.py

add_expense() asks for three values, checks the amount with **is_valid_amount()**, and adds one expense to the list.

CONCEPTS USED

- **Modules:** `from ... import` — Section 07
- **User input:** `input()` — Section 02
- **String methods:** `.strip()`, `.lower()` — Section 03
- **Conditionals:** `if`, `if not` — Section 04
- **Early return:** `return` — Section 07
- **Type conversion:** `float()` — Section 02
- **Dictionaries and lists:** `{...}`, `.append()` — Section 05



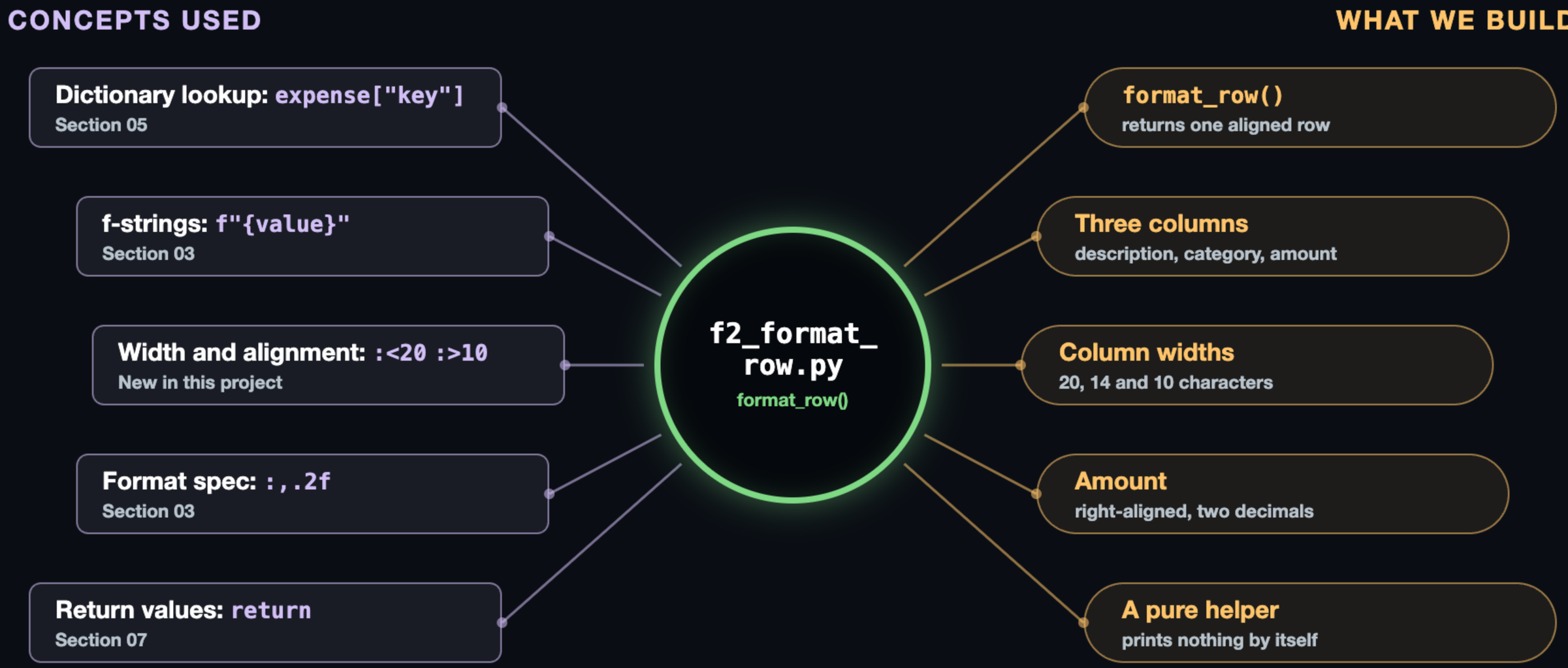
WHAT WE BUILD

- **add_expense()** — the real action for choice 1
- **The import** — `is_valid_amount` from `f1`
- **Empty description** — becomes `(no description)`
- **Bad amount** — prints a message, nothing is added
- **Category** — lowercased; empty becomes `uncategorized`
- **One expense** — a dict, appended to `expenses`
- **Choice 1 output** — Added: Coffee (3.50) [food]

© Kalyan Reddy Daida

Step-03: 3.1 One Aligned Row in f2_format_row.py

format_row() turns one expense into one line of text, every column the same width.



© Kalyan Reddy Daida

Step-03: 3.2 Wire Up the List Action in f5_actions.py

list_expenses() prints a header, then one **format_row()** line for each expense.

CONCEPTS USED

- **Modules: `from ... import`** — Section 07
- **Truthy and falsy: `if not`** — Section 04
- **Early return: `return`** — Section 07
- **Loops: `for` over a list** — Section 06
- **Width and alignment: `:<20 :>10`** — New in this project



WHAT WE BUILD

- **`list_expenses()`** — the real action for choice 2
- **The import** — `format_row` from `f2`
- **Empty list** — prints No expenses yet.
- **Header row** — the same widths as the rows
- **One row per expense** — `format_row(expense)`

© Kalyan Reddy Daida

Step-04: 4.1 Add Every Amount in f3_compute_total.py

compute_total() adds every amount in the list and returns one number.

CONCEPTS USED

- **Data types: `float`** — Section 02
- **Loops: `for` over a list** — Section 06
- **Dictionary lookup: `expense["key"]`** — Section 05
- **Augmented assignment: `+=`** — Section 04
- **Accumulator: a running total** — Section 06
- **Return values: `return`** — Section 07



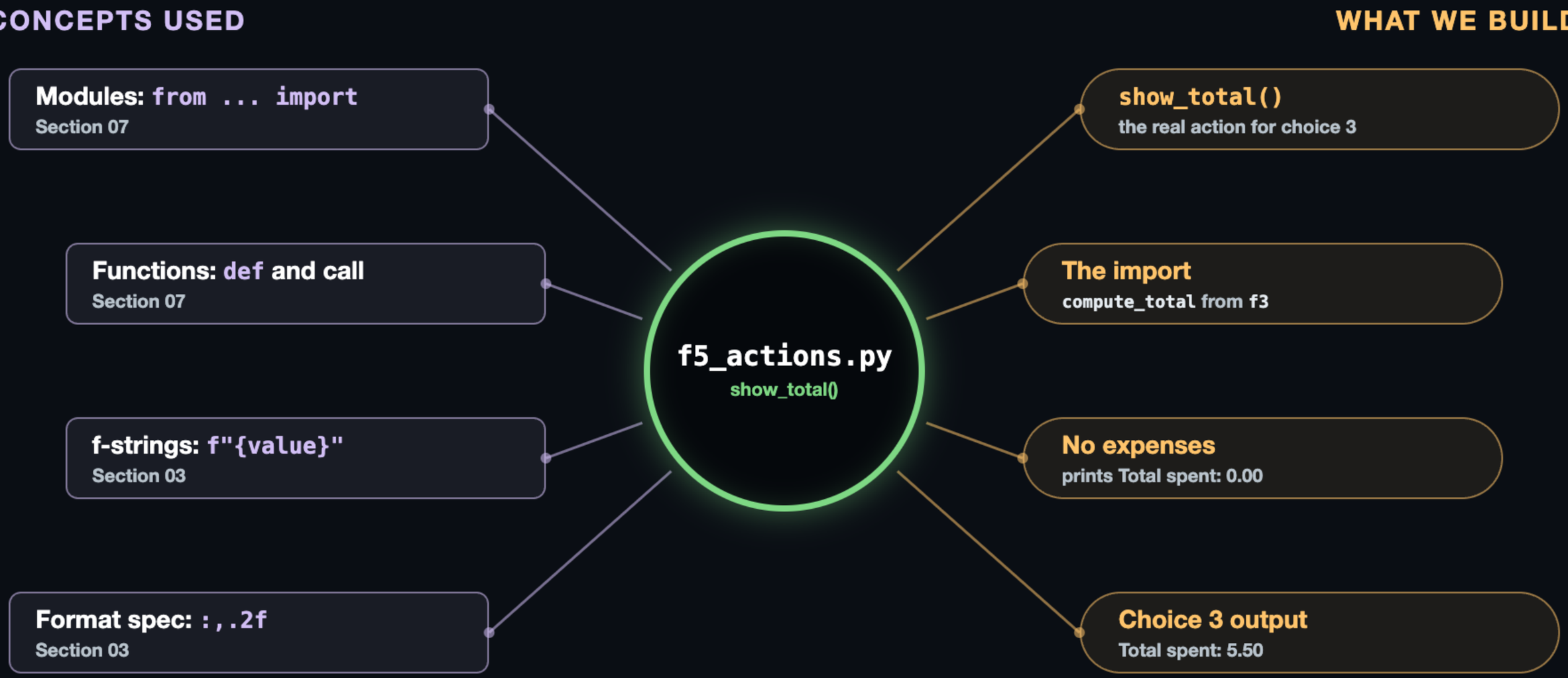
WHAT WE BUILD

- **`compute_total()`** — returns the sum of all amounts
- **`total`** — starts at `0.0`
- **Each expense** — adds its amount with `+=`
- **Empty list** — returns `0.0`
- **A pure helper** — prints nothing by itself

© Kalyan Reddy Daida

Step-04: 4.2 Wire Up the Total Action in f5_actions.py

show_total() calls **compute_total()** and prints the result as money.



© Kalyan Reddy Daida

Step-05: 5.1 Totals per Category in f4_compute_by_category.py

compute_by_category() keeps one running total per category and returns them in a dictionary.

CONCEPTS USED

- **Dictionaries: `totals = {}`** — Section 05
- **Loops: `for` over a list** — Section 06
- **Dictionary lookup: `expense["key"]`** — Section 05
- **Dictionary method: `.get()` with a default** — Section 05
- **Accumulator: a running total** — Section 06
- **Return values: `return`** — Section 07

f4_compute_by_category.py
compute_by_category()

WHAT WE BUILD

- **`compute_by_category()`** — returns a dict: category to total
- **New category** — starts at `0.0`, no `KeyError`
- **Same category** — adds to its own total
- **Empty list** — returns an empty dict
- **A pure helper** — prints nothing by itself

© Kalyan Reddy Daida

Step-05: 5.2 Wire Up the Category Action in f5_actions.py

show_by_category() asks **compute_by_category()** for the totals and prints one line per category.

CONCEPTS USED

- **Modules: `from ... import`** — Section 07
- **Truthy and falsy: `if not`** — Section 04
- **Early return: `return`** — Section 07
- **Loops: `for` with `.items()`** — Sections 05, 06
- **Width and alignment: `:<14 :>10`** — New in this project



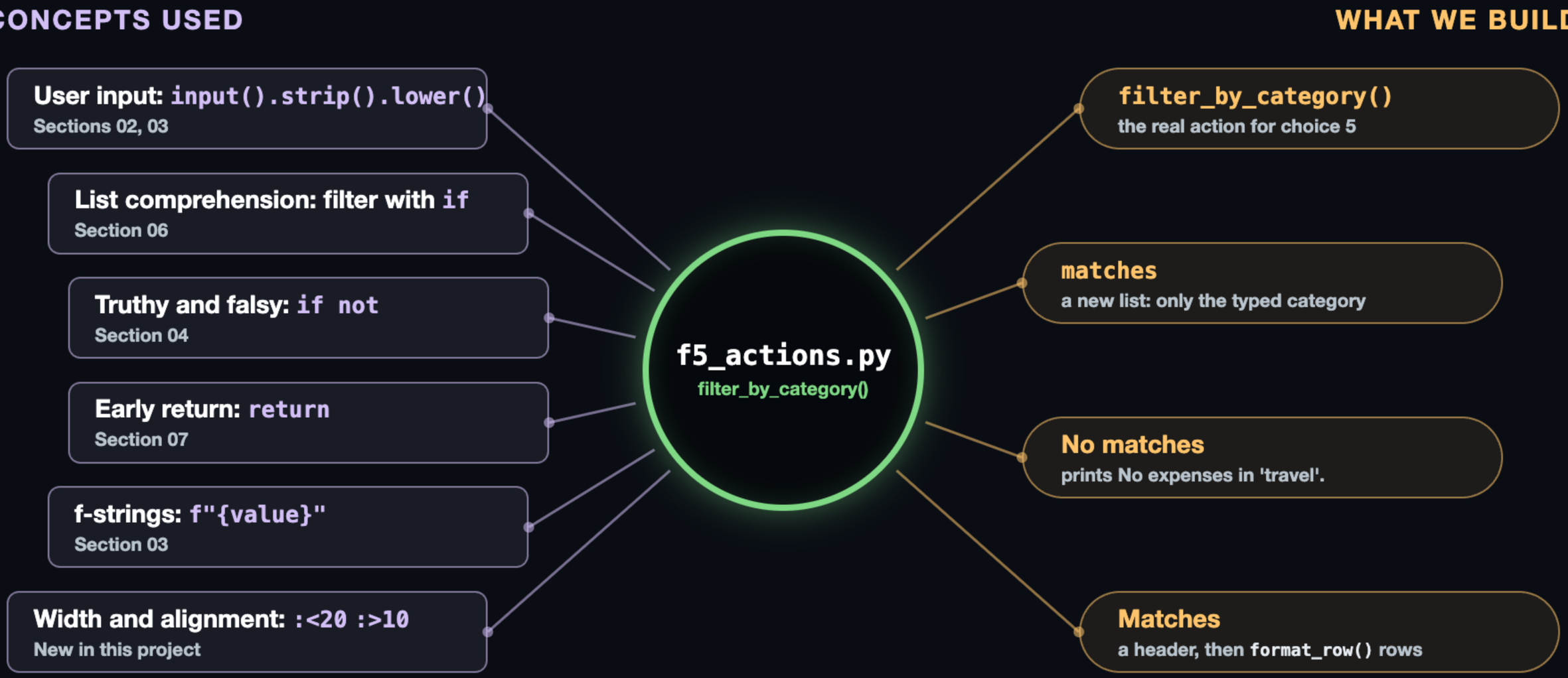
WHAT WE BUILD

- **`show_by_category()`** — the real action for choice 4
- **The import** — `compute_by_category` from `f4`
- **Empty dict** — prints No expenses yet.
- **One line per category** — category, then its total
- **Choice 4 output** — food 15.50 and transport 2.00

© Kalyan Reddy Daida

Step-06: 6.1 The Last Action in f5_actions.py

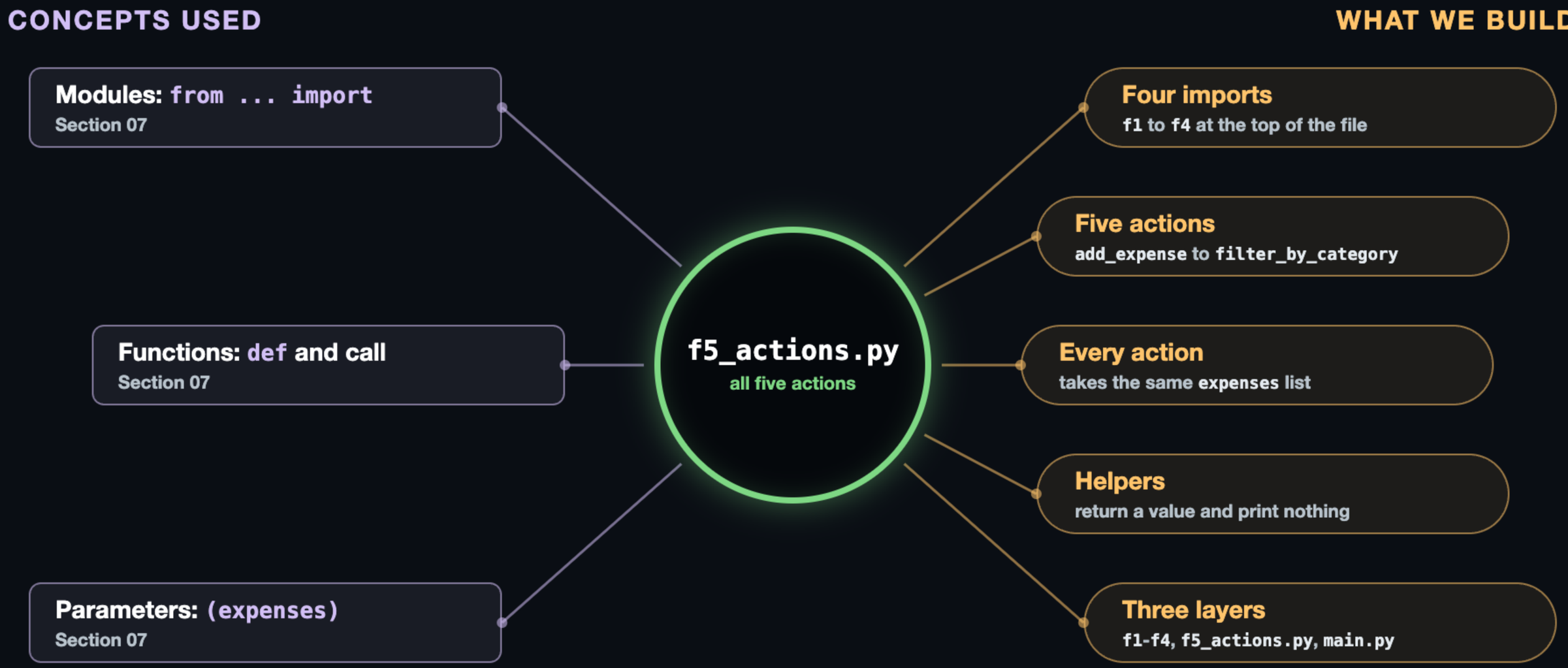
filter_by_category() asks for a category and shows only the expenses in it.



© Kalyan Reddy Daida

Step-06: 6.2 The Finished f5_actions.py

The finished **f5_actions.py**, top to bottom: four imports, then the five actions the menu calls.



© Kalyan Reddy Daida

Step-07: Run It All and Compare expense_tracker.py

Run every menu choice in your build, then read the same program in **ONE** file: same inputs, same output.

CONCEPTS USED

Run a program: `python3`
Section 02

Functions: `def` and call
Section 07

Modules: `from ... import`
Section 07

Entry point: `main()`
New in this project



WHAT WE BUILD

Choices 1 to 6
all work in one run of `main.py`

Six files
`f1` to `f4`, `f5_actions`, `main`

`expense_tracker.py`
all 11 functions in one file

No imports
nothing imported between files

Same output
`Total spent: 5.50` from both files

© Kalyan Reddy Daida

Milestone Project-2: Upgrade CLI Expense Tracker to Persistent Expense Tracker

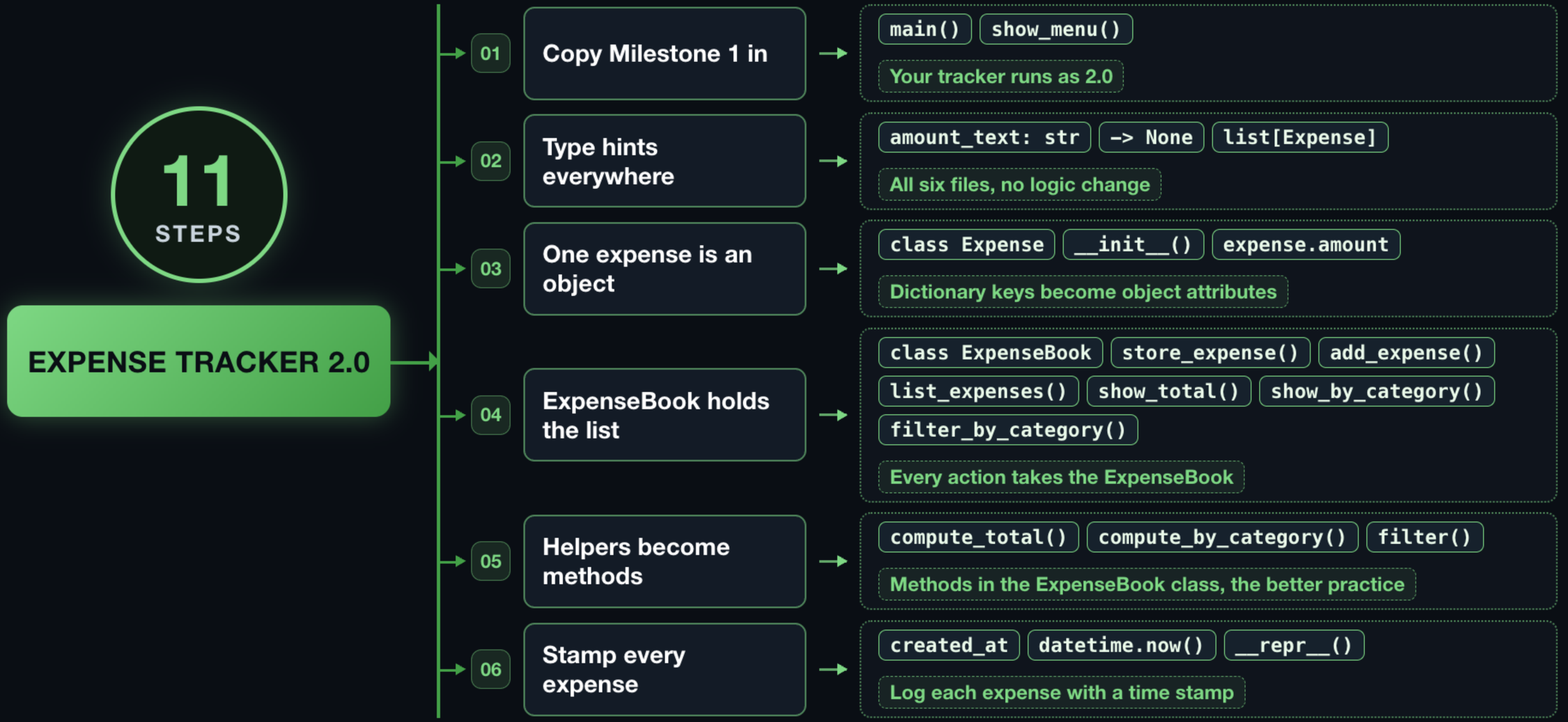
The same Expense Tracker and the same menu. Milestone 2 changes how it works inside, using Sections 08 to 13.

	MILESTONE 1: DICTIONARY-BASED	UPGRADE	MILESTONE 2: OBJECTS + JSON + ERROR HANDLING
One expense	a dictionary with three keys	Section 13 →	an **`Expense`** object with attributes
Who owns the list	**`main()`** holds **`expenses = []`**	Section 13 →	**`ExpenseBook`** owns it: **`book = ExpenseBook(...)`**
Read a value	**`expense["amount"]`**	Section 13 →	**`expense.amount`**
Total spent	**`compute_total(expenses)`**	Section 13 →	**`book.compute_total()`**
By category	**`compute_by_category(expenses)`**	Section 13 →	**`book.compute_by_category()`**
One category	a list comprehension inside **`filter_by_category()`**	Section 13 →	**`book.filter()`**, called by the same action
Types	*no type hints*	Section 08 →	**`amount_text: str`**, **`-> bool`**, **`list[Expense]`**
Quit and restart	everything is **gone**	Section 10 →	**`book.save()`** and **`book.load()`**: **`data/expenses.json`**
A broken file	*no file, so nothing to break*	Section 09 →	**`try`** / **`except`**: a warning, **no crash**
New answers	*not there yet*	Section 11 →	**`created_at`** time stamp on every expense
One package	*nothing but Python*	Section 12 →	**`.venv`** and **`pip install rich`**: a real table *(bonus)*

© Kalyan Reddy Daida

Milestone Project-2: CLI Persistent Expense Tracker - What You Will Build? 1 of 2

Eleven steps build it. Steps 01 to 06 turn your Milestone 1 code into objects and a book.



© Kalyan Reddy Daida

Milestone Project-2: CLI Persistent Expense Tracker - What You Will Build? 2 of 2

Steps 07 to 11 give it a file, a guard, a full run, and the optional rich bonus.



© Kalyan Reddy Daida

Project 2: How Your Data Survives

Project 1 forgot everything when you quit. Project 2 writes a file, so it comes back.

SESSION 1: TODAY

1. Add **Coffee 3.50 food**
2. It lives in **memory** only
3. Choose **6) Save and quit**
4. `save()` writes the file

program ends
memory wiped

SESSION 2: TOMORROW

1. Start the program again
2. `load()` reads the file
3. Objects are **rebuilt**
4. Coffee is **still there**

MEMORY: WHAT DIES

The `ExpenseBook` and its objects. Fast, but **gone the moment the program stops**.

Every new run starts with an empty list, until `load()` fills it again.

DISK: WHAT STAYS

`expenses.json` is plain text. It **stays until you delete it**.

Written by `save()` on choice 6, read back the next time you start.

WHERE IT LIVES

```
Path(__file__).parent / "data" / "expenses.json"
```

A data folder next to your code, not in the folder you ran from.

Saving

`Expense object` → `read attributes` → `plain dict` → `json.dumps()` → `JSON text` → `write_text()` → `file on disk`

Loading

`file on disk` → `read_text()` → `JSON text` → `json.loads()` → `plain dict` → `Expense(row[...])` → `Expense object`

© Kalyan Reddy Daida

Project 2: Which Folder Is Which

Five folders in **milestone-projects/milestone-project-2/**. One of them is yours to type in.

`01-orig-build-incremental`	THE ANSWER	The finished tracker, **Steps 01 to 10**. Your answer key.
`02-orig-build-incremental-with-rich`	BONUS ANSWER	After **Step-11**, with the `rich` table. Your bonus answer key.
`03-orig-warmup`	OPTIONAL	Six short drills, one new skill each. An optional extra.
`04-livecode-build-incremental`	YOU BUILD HERE	Your **Milestone 1** code. Build **Steps 01 to 10** here.
`05-livecode-build-incremental-with-rich`	BONUS	The finished **Step-11** code. Read, run, make the venv.

IN SHORT Everything you type goes in `04-livecode-build-incremental`. Compare with `01` when you are done.

© Kalyan Reddy Daida

Step-01: Copy Milestone 1 In, Retitle the Menu in main.py

Start from the Milestone 1 files in 04-livecode-build-incremental, and change one line so the menu says 2.0.

CONCEPTS USED

Run a program: `python3`
Section 02

Output: `print()`
Section 02



WHAT WE BUILD

`04-livecode-build-incremental/`
the six Milestone 1 files, no comments

Menu title
==== Expense Tracker 2.0 ====

Second run
prints No expenses yet.

© Kalyan Reddy Daida

Step-02: Add Type Hints, Starting with f1_is_valid_amount.py

Type hints on every function in all six files. No logic changes, and the output stays the same.

CONCEPTS USED

Type hints: parameters and ->
Section 08

Type hints: `list[...]`, `dict[...]`
Section 08

Type hints: not enforced
Section 08

Type hints: -> `None`
New in this project



WHAT WE BUILD

`is_valid_amount()`
takes `str`, returns `bool`

`format_row()`
takes `dict`, returns `str`

`compute_total()`
takes `list[dict]`, returns `float`

`compute_by_category()`
returns `dict[str, float]`

Five actions
each returns `None`

`show_menu()` and `main()`
both return `None`

Choice 3 output
Total spent: 3.50, as before

© Kalyan Reddy Daida

Step-03: One Expense Dictionary Becomes an Object

A dictionary with **three keys** becomes an Expense object with **three attributes**. Every square-bracket read becomes a dot.

BEFORE: A DICTIONARY

Milestone 1

One expense

```
expense = {
    "description": description,
    "amount": amount,
    "category": category,
}
```

Read a value

```
desc = expense["description"]
cat = expense["category"]
amt = expense["amount"]
```

The expense list

```
expenses: list[dict] = []
```

AFTER: AN EXPENSE OBJECT

Milestone 2, new file expense.py

One expense

```
class Expense:
    def __init__(self, description: str, amount: float, category: str) -> None:
        self.description = description
        self.amount = amount
        self.category = category

expense = Expense(description, amount, category)
```

Read a value

```
desc = expense.description
cat = expense.category
amt = expense.amount
```

The expense list

```
expenses: list[Expense] = []
```

The same hint changes in every function that takes the list: f3, f4 and all five actions in f5.

Step-03 touches six files `expense.py` new | `main.py` | `f5_actions.py` | `f2_format_row.py` | `f3_compute_total.py` | `f4_compute_by_category.py`

© Kalyan Reddy Daida

Step-03: 3.1 One Expense Becomes an Object in expense.py

Expense describes one expense: **__init__** stores its description, amount and category on the object.

CONCEPTS USED

- **Classes: a blueprint for objects**
  Section 13
- **Classes: `__init__` and `self`**
  Section 13
- **Type hints: parameters and `->`**
  Section 08



WHAT WE BUILD

- **`expense.py`**
  new file: home of the Expense class
- **`class Expense`**
  one expense, as an object
- **`__init__()`**
  stores `description`, `amount`, `category`
- **Read with a dot**
  `expense.amount`, later in f2 to f5

© Kalyan Reddy Daida

Step-03: 3.3 to 3.7 Build It, Then Fix Every Read in f5_actions.py

add_expense() builds an Expense, and the TypeError shows each read to change into a dot, until every choice prints the same output.

CONCEPTS USED

Modules: `from ... import`
Section 07

Classes: attributes with a dot
Section 13

Debugging: read the traceback
Section 09

`f5_actions.py`
then f2, f3 and f4

WHAT WE BUILD

`f5_actions.py`
`add_expense()` builds an `Expense`

`TypeError`
'Expense' object is not subscriptable

`f2_format_row.py`
three reads become dots

`f3_compute_total.py`
one read becomes a dot

`f4_compute_by_category.py`
two reads become dots

`f5_actions.py`
`filter_by_category()`, one read

© Kalyan Reddy Daida

Step-04: The Expense List Becomes an ExpenseBook

`main()` stops holding the list itself. An **ExpenseBook** owns the list, stores each new expense, and **every action takes the book**.

BEFORE: MAIN() HOLDS THE LIST

end of Step-03

Who holds the list `main.py` in `main()`

```
def main() -> None:
    expenses: list[Expense] = []
```

Store one expense `f5_actions.py` in `add_expense()`

```
expense = Expense(description, amount, category)
expenses.append(expense)
```

Hand it to the actions `main.py` in `main()`

```
add_expense(expenses)
list_expenses(expenses)
```

Inside an action `f5_actions.py` in `list_expenses()`

```
def list_expenses(expenses: list[Expense]) -> None:
    for expense in expenses:
```

AFTER: THE BOOK HOLDS THE LIST

new file expense_book.py

Who holds the list `expense_book.py` in `__init__()`

```
class ExpenseBook:
    def __init__(self) -> None:
        self.expenses: list[Expense] = []
```

Store one expense `expense_book.py` in `store_expense()`

```
    def store_expense(self, description: str, amount: float, category: str) -> Expense:
        expense = Expense(description, amount, category)
        self.expenses.append(expense)
        return expense
```

Hand it to the actions `main.py` in `main()`

```
book = ExpenseBook()
add_expense(book)
list_expenses(book)
```

Inside an action `f5_actions.py` in `list_expenses()`

```
def list_expenses(book: ExpenseBook) -> None:
    for expense in book.expenses:
```

What changes

- `expense_book.py` new — `class ExpenseBook` with `store_expense()`
- `main.py` — `main()` makes the book and passes it to every action
- `f5_actions.py` — all five actions take the book: `add_expense()` `list_expenses()` `show_total()` `show_by_category()` `filter_by_category()`

© Kalyan Reddy Daida

Step-04: 4.1 ExpenseBook Holds the List in expense_book.py

ExpenseBook keeps the list inside the class, and **store_expense()** builds and keeps each Expense.

CONCEPTS USED

Classes: `__init__` and `self`
Section 13

Classes: methods
Section 13

Type hints: `list[...]`, `dict[...]`
Section 08

WHAT WE BUILD

`expense_book.py`
the ExpenseBook class

`ExpenseBook`
holds `self.expenses`

`self.expenses`
starts as an empty list

`store_expense()`
builds, keeps and returns an Expense

© Kalyan Reddy Daida

Step-04: 4.2 to 4.7 Every Action Takes the Book in f5_actions.py

The five actions take the **book** instead of a list, and **main()** creates one book and passes it on.

CONCEPTS USED

Modules: `from ... import`
Section 07

Type hints: parameters and `->`
Section 08

Classes: create an object
Section 13

Classes: methods
Section 13

`f5_actions.py`
and main.py

WHAT WE BUILD

Five actions
each takes the **book**, not a list

`add_expense()`
calls `book.store_expense()`

Four actions
read `book.expenses`

`main()`
creates one book

Choice 1 output
Added: Coffee (3.50) [food], as before

© Kalyan Reddy Daida

Step-05: Why Move Two Helpers Into the ExpenseBook?

The **ExpenseBook** owns the expenses, so the **ExpenseBook does the work on them.**

BEFORE: THE WORK IS SPREAD OUT end of Step-04

receives the list `f3_compute_total.py`

```
def compute_total(expenses: list[Expense])
```

receives the list `f4_compute_by_category.py`

```
def compute_by_category(expenses: list[Expense])
```

searches the list on its own `f5_actions.py` `filter_by_category()`

```
for expense in book.expenses
```

move the work to the owner

AFTER: THE BOOK DOES ITS OWN WORK end of Step-05

the actions in `f5_actions.py` ask

```
book.compute_total()
book.compute_by_category()
book.filter(wanted_category)
```

asks →
← answers

class ExpenseBook `expense_book.py`

`self.expenses`

```
def compute_total(self)
def compute_by_category(self)
def filter(self, category: str)
```

STAY AS HELPERS `is_valid_amount()` in f1 checks the typed amount.
`format_row()` in f2 formats one table row. Neither one needs the book's list.

One menu choice: the action asks, the book answers

choice 5 (Show one category), you type food

ACTION	ACTION	EXPENSEBOOK	ACTION
1. Ask the user	**2. Ask the book**	**3. Find the matches**	**4. Show the answer**
`input("Category to show: ")`	`book.filter(wanted_category)`	`for expense in self.expenses`	`print(format_row(expense))`

The owner does the work

The book holds the expenses, so it answers the questions about them: the total, the spending by category, the matches.

A new screen, the same book

Now: the command line
`f5_actions.py` with `input()` and `print()`

Later: a web app
a web page with a form and a table

→ **ExpenseBook**
the same methods
no `input()` or `print()`

In Step-11, the `rich` package changes the screen, and `expense_book.py` stays the same.

One place for all three

They now live in `expense_book.py`:

- the total: `compute_total()`
- by category: `compute_by_category()`
- the matches: `filter()`

Before, they were in f3, f4 and f5.

© Kalyan Reddy Daida

Step-05: Two Helper Files Move Into the ExpenseBook

`compute_total()` and `compute_by_category()` become **methods**, the filter moves in too, and the actions **ask the book**.

BEFORE: HELPER FILES DO THE WORK

end of Step-04

Total spent `f3_compute_total.py` in `compute_total()`

```python
def compute_total(expenses: list[Expense]) -> float:
    total = 0.0
    for expense in expenses:
        total += expense.amount
```

By category `f5_actions.py` in `show_by_category()`

```python
totals = compute_by_category(book.expenses)
```

One category `f5_actions.py` in `filter_by_category()`

```python
matches = [
    expense
    for expense in book.expenses
    if expense.category == wanted_category
]
```

AFTER: THE BOOK DOES THE WORK

three methods in expense_book.py

Total spent `expense_book.py` in `compute_total()`

```python
    def compute_total(self) -> float:
        total = 0.0
        for expense in self.expenses:
            total += expense.amount
```

By category `f5_actions.py` in `show_by_category()`

```python
totals = book.compute_by_category()
```

One category `expense_book.py` in `filter()`

```python
    def filter(self, category: str) -> list[Expense]:
        return [expense for expense in self.expenses if expense.category == category]
```

What changes

- `expense_book.py` — three new methods: `compute_total()` `compute_by_category()` `filter()`
- `f5_actions.py` — the actions ask the book: `book.compute_total()` `book.compute_by_category()` `book.filter(wanted_category)`
- `f3_compute_total.py` deleted, `f4_compute_by_category.py` deleted — their code lives in the class now

© Kalyan Reddy Daida

Step-05: 5.1 Two Helpers Become Methods in expense_book.py

Your two helper functions move into the class as methods, with one change: **self.expenses**.

CONCEPTS USED

- **Classes: methods**
  Section 13
- **Accumulator: a running total**
  Section 06
- **Dictionaries: `.get()` with a default**
  Section 05
- **List comprehension: filter with `if`**
  Section 06



WHAT WE BUILD

- **`compute_total()`**
  moved in, same name
- **`compute_by_category()`**
  moved in, same name
- **`filter()`**
  the list comprehension, moved in
- **One change**
  `expenses` becomes `self.expenses`

© Kalyan Reddy Daida

Step-05: 5.2 and 5.3 Actions Call the Methods in f5_actions.py

The actions call the book's methods, and the **f3** and **f4** files and their imports are deleted.

CONCEPTS USED

Modules: `from ... import`
Section 07

Classes: methods
Section 13

WHAT WE BUILD

`f5_actions.py`
f3 and f4 deleted

`show_total()`
calls `book.compute_total()`

`show_by_category()`
calls `book.compute_by_category()`

`filter_by_category()`
calls `book.filter()`

Deleted
the `f3` and `f4` files

Choice 3 output
Total spent: 5.50, as before

© Kalyan Reddy Daida

Step-06: Two New Things, datetime and __repr__

Every expense gets a **time stamp**, `created_at`, and a **readable way to show itself**.

1 A time stamp with `datetime`

The **book** stamps each expense with the time as it stores it.

stamp the time `expense_book.py` `store_expense()`

```
stamp = datetime.now().isoformat(timespec="seconds")
expense = Expense(description, amount, category, stamp)
```

`datetime.now()` the current date and time
`.isoformat(...)` turns it into text like `2026-09-16T21:50:17`

a fourth value `expense.py` `__init__()`

```
self.created_at = created_at
```

2 What is `__repr__`?

`__repr__` is a special method. It returns the **exact text Python shows for an object**: at the `>>>` prompt, in lists, and in `print()` when there is no `__str__`.

show every value `expense.py` `__repr__()`

```
def __repr__(self) -> str:
    return (f"Expense(description={self.description!r}, amount={self.amount!r}, "
            f"category={self.category!r}, created_at={self.created_at!r})")
```

`!r` puts quotes around text values, like `'Coffee'`

At a Python prompt *the same line, without and with __repr__*

```
>>> book.store_expense("Coffee", 3.50, "food")
```

WITHOUT __REPR__ `<expense.Expense object at 0x100e10980>` only the file and class name, and a memory address

WITH __REPR__ `Expense(description='Coffee', amount=3.5, category='food', created_at='2026-09-16T21:50:17')`

Why a time stamp?

Every expense remembers when it was added. In Step-07, `created_at` is saved to the file with the other three values.

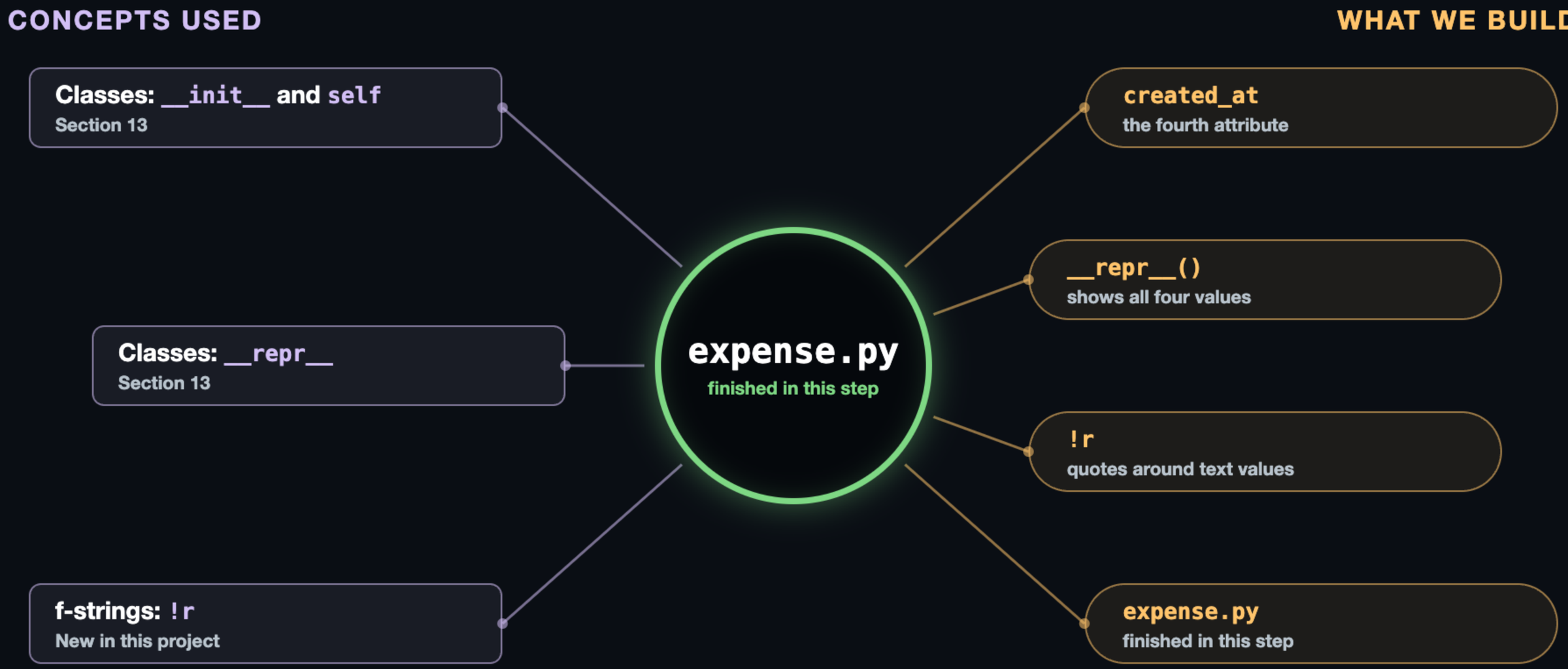
Why `__repr__`?

Without it, every expense shows up as `<expense.Expense object at 0x...>`, with no values at all. With it, you **see your data** everywhere Python shows an expense, so checking your work and fixing bugs gets easy.

© Kalyan Reddy Daida

Step-06: 6.1 The Fourth Attribute and __repr__ in expense.py

Every expense gets a **created_at** attribute and a **__repr__** that shows all four values.



© Kalyan Reddy Daida

Step-06: 6.2 and 6.3 Stamp the Time in expense_book.py

store_expense() stamps each new expense with the time, and a Python prompt shows it through **__repr__**.

CONCEPTS USED

Dates: `datetime`
Section 11

Dates: `.isoformat()`
New in this project

WHAT WE BUILD

`expense_book.py`
and the finished f5

`store_expense()`
stamps the time

`datetime.now()`
the current date and time

`.isoformat()`
text like `2026-09-16T12:43:14`

`f5_actions.py`
finished in this step

At a Python prompt
`book.store_expense()` returns an `Expense(...)`

© Kalyan Reddy Daida

Step-07: Save Your Expenses to a JSON File

The book gets a file path and a `save()` method, and choice 6 becomes **Save and quit**.

BEFORE The expenses live only in memory, in `self.expenses`. Quit, and they are gone.

AFTER Choice 6 writes them to `data/expenses.json`, next to your code.

How `save()` writes the file `expense_book.py`

1. IN MEMORY
   Expense objects
   `self.expenses`

→

2. ONE DICTIONARY EACH
   object to dict
   `"amount": expense.amount`

→

3. JSON TEXT
   dicts to text
   `json.dumps(rows, indent=2)`

→

4. THE FILE
   text to the file
   `write_text()`

WHY DICTS? `json.dumps` can only write **plain data** like dictionaries and lists, not your own `Expense` objects.

what you see `choice 6 in the menu`

```
6) Save and quit
Choose 1-6: 6
Saved. Goodbye!
```

The expenses are now in a file, next to **`main.py`**. The full code changes are on the next slide.

data/expenses.json `after adding Coffee, then choice 6`

```
[
  {
    "description": "Coffee",
    "amount": 3.5,
    "category": "food",
    "created_at": "2026-09-16T21:50:17"
  }
]
```

NEXT Step-08 loads this file back when the app starts, so your expenses **survive a restart**.

Step-07: The Code Changes in expense_book.py and main.py

+ new line − removed line grey lines stay the same

expense_book.py the book gets a file path and a save() method

```
+ import json
  from datetime import datetime
+ from pathlib import Path
  ...
-     def __init__(self) -> None:
+     def __init__(self, path: Path) -> None:
+         self.path = path
          self.expenses: list[Expense] = []
```

```
  ...
+     def save(self) -> None:
+         """Write every expense to the JSON file, making the data folder if it is missing."""
+         rows = [{"description": expense.description, "amount": expense.amount,
+                  "category": expense.category, "created_at": expense.created_at}
+                 for expense in self.expenses]
+         self.path.parent.mkdir(parents=True, exist_ok=True)
+         self.path.write_text(json.dumps(rows, indent=2))
```

`mkdir(parents=True, exist_ok=True)` makes the **data** folder, and does nothing if it already exists. `indent=2` makes the file easy to read.

main.py the book gets its file, and choice 6 saves

```
+ from pathlib import Path
  from expense_book import ExpenseBook
  ...
      print("5) Show one category")
-     print("6) Quit")
+     print("6) Save and quit")
```

```
  def main() -> None:
-     book = ExpenseBook()
+     book = ExpenseBook(Path(__file__).parent / "data" / "expenses.json")
      while True:
  ...
          elif choice == "6":
-             print("Goodbye!")
+             book.save()
+             print("Saved. Goodbye!")
              break
```

`Path(__file__).parent` is the folder that holds `main.py`, so the file always lands next to your code.

© Kalyan Reddy Daida

Step-07: 7.1 Save to JSON in expense_book.py

save() turns every expense back into a dict and writes them all to one JSON file.



© Kalyan Reddy Daida

Step-07: 7.2 The Data Path and Choice 6 in main.py

main() gives the book a file next to **main.py**, and choice 6 saves before it quits.

CONCEPTS USED

Paths: `__file__`
Section 10

Classes: methods
Section 13



WHAT WE BUILD

`Path(__file__).parent`
the folder that holds `main.py`

`data/expenses.json`
next to `main.py`

Menu line 6
Save and quit

Choice 6
saves, then prints Saved. Goodbye!

© Kalyan Reddy Daida

Step-08: The Same Trip, Out and Back

`save()` turns your **objects into text**. `load()` turns that **text back into objects**.

Saving `save()` Step-07, choice 6

Expense objects `self.expenses`

↓ a dict per expense

dictionaries one per expense, in memory

```
{"description": "Coffee",
 "amount": 3.5,
 "category": "food",
 "created_at": "2026-09-16T21:50:17"}
```

↓ `json.dumps()`

JSON text one long string

```
[
  {
    "description": "Coffee",
    ... the other three keys ...
```

↓ `write_text()`

the file `data/expenses.json`

Loading `load()` Step-08, on start

the file `data/expenses.json`

↓ `read_text()`

JSON text one long string

```
[
  {
    "description": "Coffee",
    ... the other three keys ...
```

↓ `json.loads()`

dictionaries one per row, back in memory

```
{"description": "Coffee",
 "amount": 3.5,
 "category": "food",
 "created_at": "2026-09-16T21:50:17"}
```

↓ `Expense(...)` one per row

Expense objects `self.expenses`

WHY THE MIDDLE STEPS? A file holds **text**, and JSON only knows plain data: text, numbers, lists and dictionaries. Your **Expense** objects go out as dictionaries and come back as objects, and the same four keys travel both ways: `description` `amount` `category` `created_at`

© Kalyan Reddy Daida

Step-08: Load It Back, So Your Data Survives a Restart

A `load()` method reads the file into **Expense objects**, and `main()` calls it **once, before the menu loop**.

BEFORE Step-07 writes the file, but nobody reads it. Start the app again and the book is **empty**.

AFTER `book.load()` fills the book from the file before the menu appears.

How `load()` reads the file `expense_book.py`

1. **THE FILE** — JSON text on disk — `data/expenses.json`
2. **TEXT TO DICTS** — read and parse — `json.loads(self.path.read_text())`
3. **DICTS TO OBJECTS** — one Expense per row — `for row in rows`
4. **IN THE BOOK** — ready for the menu — `self.expenses`

FIRST RUN? There is no file yet, so `if not self.path.exists()` starts with an **empty list** and returns. That is normal, not an error.

the new method `expense_book.py`

```
    def load(self) -> None:
        if not self.path.exists():
            self.expenses = []
            return
        rows = json.loads(self.path.read_text())
        self.expenses = [Expense(row["description"], row["amount"],
                                 row["category"], row["created_at"]) for row in rows]
```

run it again `after Save and quit`

```
Choose 1-6: 2
Description          Category          Amount
Coffee               food                3.50
```

The expense is **still there**. Milestone 1 lost it on restart.

one line in `main.py` `book.load()` right after the book is created, and **before** the menu loop starts

© Kalyan Reddy Daida

Step-08: 8.1 Load It Back in expense_book.py

load() reads the JSON file back into Expense objects, or starts empty when there is no file.

CONCEPTS USED

JSON: `json.loads`
Section 10

Files: `read_text()` and `.exists()`
Section 10

List comprehension: `[... for ...]`
Section 06



WHAT WE BUILD

`load()`
reads the file into **Expense** objects

Missing file
an empty book, and no error

One Expense per dict
built by a list comprehension

`expense_book.py`
finished in this step

© Kalyan Reddy Daida

Step-08: 8.2 Load on Start in main.py

main() calls **book.load()** before the menu loop, so your data survives a restart.

CONCEPTS USED

Classes: methods
Section 13

Loops: `while True`
Section 06



WHAT WE BUILD

`main()`
calls `book.load()` on start

Before the loop
the book fills from the file

Second run
lists the expense you saved

© Kalyan Reddy Daida

Step-09: A Broken File Must Not Crash Your App

One `try` / `except` around `book.load()` turns a crash into a **warning**, and the app starts empty.

break it on purpose `data/expenses.json`

```
[
  {
    "description": "Coffee",
    "amount": 3.5,
```

before the fix `the app dies before the menu`

```
  File ".../04-livecode-build-incremental/main.py", line 26, in main
    book.load()
  File ".../04-livecode-build-incremental/expense_book.py", line 54, in load
    rows = json.loads(self.path.read_text())
json.decoder.JSONDecodeError: Expecting property name enclosed in double quotes
```

the fix `main.py in main()`

```
    try:
        book.load()
    except (json.JSONDecodeError, TypeError, KeyError):
        print("Warning: data file is corrupt; starting empty.")
        book.expenses = []
```

after the fix `same broken file`

```
Warning: data file is corrupt; starting empty.

Choose 1-6: 2
No expenses yet.
```

Three errors, one reply `except (A, B, C)`

`json.JSONDecodeError`	the text is not valid JSON
`TypeError`	valid JSON, wrong shape, like `[1, 2, 3]`
`KeyError`	a record is missing a field

MISSING OR BROKEN? A **missing** file is normal: `load()` handles it with an `if` and starts empty. A **broken** file raises an error, so `main()` catches it.

© Kalyan Reddy Daida

Step-09: Run It or Import It, the __main__ Guard

Two lines decide whether the menu starts: `if __name__ == "__main__":` and an indented `main()`.

before the last line of main.py

```
main()
```

after the last two lines of main.py

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

You run it `python3 main.py`

`__name__` is **"__main__"** in this file

`the if` is **true**

`main()` runs

The menu starts

Something imports it `import main`

`__name__` is **"main"** instead

`the if` is **false**

`main()` does not run

You get the functions, with no menu

Try it at a Python prompt in your build folder, type python3

```
>>> import main                 nothing is printed: no menu, no prompt
>>> main.show_menu()            you call it yourself, and the menu appears

==== Expense Tracker 2.0 ====
1) Add expense
```

The same file can now be **a program to run** and **a module to import**.

FROM SECTION 12 You met this guard with a small package. Here it goes into **your own app** , so `main.py` can be run or imported.

Step-09: 9.1 Do Not Crash on a Broken File in main.py

A **try** / **except** around load() turns a broken data file into a warning instead of a crash.

CONCEPTS USED

Error handling: `try` / `except`
Section 09

Error handling: `except (A, B)`
Section 09

Exceptions: `json.JSONDecodeError`
New in this project

WHAT WE BUILD

`try` around `load()`
catches three errors

Broken file
a warning, then an empty book

Printed warning
Warning: data file is corrupt; starting empty.

Choice 6
saves the empty book over the broken file



© Kalyan Reddy Daida

Step-09: 9.2 Run It or Import It in main.py

if __name__ == "__main__": starts the menu when you run the file, and stays quiet when another file imports it.

CONCEPTS USED

The `__main__` guard
Section 12

Modules: `import`
Section 07



WHAT WE BUILD

`if __name__ == "__main__":`
the last two lines of `main.py`

`python3 main.py`
the menu starts

`import main`
prints nothing at all

`main.show_menu()`
you call it yourself

`main.py`
finished in this step

© Kalyan Reddy Daida

Step-10: Run It All in main.py

Run every menu choice, quit and restart to see your expenses come back, then watch a **broken** data file warn instead of crash.

CONCEPTS USED

- **Run a program:** `python3` — Section 02
- **Modules:** `from ... import` — Section 07
- **Classes:** `class` — Section 13
- **Files:** `write_text()`, `read_text()` — Section 10
- **Error handling:** `try` / `except` — Section 09



WHAT WE BUILD

- **Choices 1 to 6** — all six work in one run
- **`Total spent: 17.50`** — your three expenses, added up
- **`data/expenses.json`** — written by choice 6
- **Restart** — your expenses load again
- **Broken file** — a warning, not a crash

© Kalyan Reddy Daida

Step-11 (bonus): One venv, One Package, a Real Table

Optional, and in its own folder: Section 12 meets your own app. `expense_book.py` and `main.py` do not change at all.

set it up `in 05-livecode-build-incremental-with-rich/`

```
cd ..
cd 05-livecode-build-incremental-with-rich
python3 -m venv .venv
source .venv/bin/activate
pip install rich
```

On Windows the `source` line is `.venv\Scripts\activate` instead. Your prompt then starts with `(.venv)`, so you can see the toolbox is on.

write down what the project needs `requirements.txt`

```
pip freeze > requirements.txt
```

`rich==15.0.0` and the packages rich itself needs. Anyone who clones the project runs `pip install -r requirements.txt`. Add `.venv/` to `.gitignore`: it is rebuilt, never committed.

if you skip the venv `the app stops before the menu`

```
  File ".../f5_actions.py", line 3, in <module>
    from rich.console import Console
ModuleNotFoundError: No module named 'rich'
```

Only the part that talks to the screen changes

- CHANGES `f2_format_row.py` a new make_table()
- CHANGES `f5_actions.py` prints one table
- NO CHANGE `expense.py` `expense_book.py` `f1_is_valid_amount.py` `main.py`

ONE TOOLBOX PER PROJECT Turn it off again with `deactivate` and `rich` is gone: it lives only inside **this folder's** `.venv`, never in your system Python.

Step-11 (bonus): A Table Instead of Printed Lines

One new helper, `make_table()`, and the two actions print **one table** instead of a header and a row each.

the new helper `f2_format_row.py`

```python
from rich.table import Table
...
def make_table(expenses: list[Expense]) -> Table:
    table = Table(title="Expenses")
    table.add_column("Description")
    table.add_column("Category")
    table.add_column("Amount", justify="right")
    for expense in expenses:
        table.add_row(expense.description, expense.category,
                      f"{expense.amount:,.2f}")
    return table
```

print it `f5_actions.py list_expenses() and filter_by_category()`

```python
from f1_is_valid_amount import is_valid_amount
~~from f2_format_row import format_row~~
from rich.console import Console

from f2_format_row import make_table
...
~~print(f"{'Description':<20} {'Category':<14} {'Amount':>10}")~~
~~for expense in book.expenses:~~
~~    print(format_row(expense))~~

Console().print(make_table(book.expenses))
```

before: choice 2, printed lines

```
Description          Category          Amount
Coffee               food                3.50
Bus                  transport           2.00
```

after: choice 2, a rich table

Expenses

Description	Category	Amount
Coffee Bus	food transport	3.50 2.00

THE BOOK NEVER CHANGED

The same **ExpenseBook** answers the same questions. Only the screen code changed, which is exactly why **Step-05 moved the work into the book**.

Step-11: 11.1 to 11.4 Your First Installed Package: rich

A virtual environment, one **pip install**, and the expense list arrives as a real table.

CONCEPTS USED

- **Virtual environments: `venv`** — Section 12
- **Packages: `pip install`** — Section 12
- **Packages: `pip freeze`** — Section 12
- **Modules: `from ... import`** — Section 07



WHAT WE BUILD

- **`.venv`** — one toolbox for this project
- **`requirements.txt`** — `rich==15.0.0`
- **`make_table()`** — builds the table from your expenses
- **Choices 2 and 5** — print a bordered table
- **No venv** — `No module named 'rich'`
- **ExpenseBook** — does not change at all

© Kalyan Reddy Daida

Project 2: What You Built

One tracker, six files, and the ideas from Sections 07 to 13 working together. Your data is still there after you quit.

the six files, and the job each one has

- `expense.py` one expense, an object with four attributes and a **__repr__**
- `expense_book.py` the book: holds the list, and does the work on it
- `f1_is_valid_amount.py` checks the amount the user typed
- `f2_format_row.py` turns one expense into one aligned row
- `f5_actions.py` the five actions behind the menu
- `main.py` the menu, the loop, the save on quit, the guard

what it does when you run it

1. Adds an expense, stamped with `created_at`
2. Lists them all, one aligned row each
3. Totals the spending with `book.compute_total()`
4. Totals it per category
5. Shows one category with `book.filter()`
6. Saves to `data/expenses.json` and quits, then **loads it back on the next run**

The ideas you used, and where each one came from

`13` **classes, __init__, self, methods** · `08` **type hints on every function** · `10` **Path, write_text, read_text, json** · `09` **try / except, and except (A, B)** · `11` **datetime and .isoformat()** · `07` **your own modules, from ... import** · `12` **the __main__ guard, and venv and pip in the bonus**

Where this goes next

READY FOR MORE Your `ExpenseBook` has no `input()` and no `print()`, so the same class can sit behind a **web API**

Next in the StackSimplify family, and this project is the foundation they build on

Complete Python Bootcamp | **Python for Machine Learning** | **The MLOps Courses**

© Kalyan Reddy Daida