

Day 13 Modules, packages and environments

Self-Study Guide · About 75 minutes

By the end you will be able to

- Import code with `import`, `from ... import`, and `as`, and explain which names each form creates.
- Split a program into a module of functions and a main script, using `if __name__ == "__main__":` to keep test code from running on import.
- Use `math`, `random`, `datetime`, `os`, and `sys` from the standard library for everyday jobs.
- Create and activate a virtual environment, install a package with `pip`, and record it in `requirements.txt`.

What to remember from earlier

- You run a script by typing `python` and its file name in a terminal that is in the file's folder. ([Day 1: Setup and your first program, 1.3 Writing and Running a Script](#))
- A `NameError` means Python met a name that nothing has created yet. ([Day 1: Setup and your first program, 1.9 Reading a NameError](#))
- `range(start, stop)` stops before `stop`, so `range(1, 7)` gives 1 to 6. ([Day 5: Loops, 5.4 Counting with `range\(\)`](#))
- `counts[key] = counts.get(key, 0) + 1` counts how many times each key turns up. ([Day 8: Dictionaries, 8.9 Counting with a Dictionary](#))
- A function is defined with `def`, gets its inputs through parameters, and hands back a value with `return`. ([Day 9: Functions, part 1](#))
- A list comprehension such as `[n * 2 for n in numbers]` builds a new list in one line. ([Day 11: Comprehensions and iteration tools, 11.1 List Comprehensions](#))
- The Day 12 to-do manager keeps tasks as a list of dictionaries and does each action in its own function. ([Day 12: Checkpoint project 1](#))

Make a `day13` folder for today's files and open it in your editor, as on earlier days. Some of today's examples are small **projects** of two or more files; give each one its own folder inside `day13` (for example `day13/shop/`), as the file titles show.

Day 12's to-do manager was already long enough that finding a function took scrolling. Today you learn how Python programs are built from pieces: code that ships with Python, your own code split across files, and code other people have written, installed separately for each project.

13.1 Modules and import

On Day 2 you found a square root with `9 ** 0.5`. That works, but it is not obvious to a reader, and many jobs (rounding up, the number π , dates, random numbers) have no operator at all. Python keeps tools like these in separate files and lets each program load only what it needs.

A **module** is a file of Python code (functions, variables, and so on) that other programs can use. Python comes with hundreds of modules already written and installed; together they are called the **standard library**. To use a module, a program **imports** it with the `import` statement:

```
1 import math
```

- `import` is a keyword. `math` is the module's name: the file name without `.py`.
- Python finds the module, runs its code once, and makes a name, `math`, that refers to the whole module.
- You then reach anything inside the module with a dot: `math.sqrt` is "the `sqrt` function that lives in `math`". This is the same dot you used on Day 3 for string methods such as `name.upper()`, read as "belongs to".

Imports go at the **top** of the file, so a reader sees at once which modules the program depends on. The `math` module holds mathematical functions and constants:

```
>>> import math
>>> math.sqrt(16)
4.0
>>> math.sqrt(2)
1.4142135623730951
>>> math.pi
3.141592653589793
>>> math.floor(2.7)
2
>>> math.ceil(2.1)
3
```

- `math.sqrt(16)` is the square root, always a float.
- `math.pi` is a variable, not a function, so there are no parentheses. A variable stored in a module is often called a **constant** when it is meant never to change.
- `math.floor(2.7)` rounds **down** to the whole number below; `math.ceil(2.1)` rounds **up** to the whole number above. Both give an `int`.

WORKED EXAMPLE The area of a circle

A circular rug has a radius of 3 metres. The area of a circle is π times the radius squared.

circle_area.py

```
1 import math
2
3 radius = 3
4 area = math.pi * radius ** 2
5 print(f"Area: {area:.2f}")
6 print(f"Side of a square with the same area: {math.sqrt(area):.2f}")
```

```
Area: 28.27
Side of a square with the same area: 5.32
```

- Line 1 imports the module. Nothing is printed: importing only loads tools.
- Line 4 uses `math.pi` like any number: `3.14159... * 9` is `28.274...`, and on line 5 `:.2f` (Day 3) shows two decimal places.
- Line 6 passes `area` to `math.sqrt`, whose result is about 5.317. The square root of an area is the side of a square with that area.

Now vary the job. A wall of 23 square metres needs paint, and one can covers 10. You cannot buy 2.3 cans, so the program must round, but which way?

paint_cans.py

```
1 import math
2
3 wall_area = 23
4 area_per_can = 10
5 exact = wall_area / area_per_can
6 print(f"Exact: {exact}")
7 print(f"round(): {round(exact)} cans")
8 print(f"math.ceil(): {math.ceil(exact)} cans")
```

```
Exact: 2.3
round(): 2 cans
math.ceil(): 3 cans
```

The built-in `round()` from Day 2 goes to the **nearest** whole number, which is 2, and 2 cans leave 3 square metres unpainted. `math.ceil` always goes **up**, which is what "buy enough" means. Use `ceil` when a part-filled thing still counts as a whole one (cans, buses, pages), and `floor` when only complete ones count (full hours, whole boxes filled).

KEY IDEA

`import math` loads the `math` module and creates the name `math`. Everything inside it is reached with a dot: `math.sqrt(x)`, `math.pi`, `math.ceil(x)`.

WATCH OUT

Using a module without importing it. Every program that uses `math` needs its own `import math`, even if another program you wrote yesterday imported it:

forgot_import.py

```
1 side = math.sqrt(49)
2 print(side)
```

```
Traceback (most recent call last):
  File ".../day13/forgot_import.py", line 1, in <module>
    side = math.sqrt(49)
           ^^^^^
```

```
NameError: name 'math' is not defined. Did you forget to import 'math'?
```

This is the Day 1 `NameError`: nothing in this file has created the name `math`. Recent versions of Python add the hint "Did you forget to import 'math'?"; older ones stop after "is not defined". The fix is `import math` on line 1.

PAUSE AND TRY**Question 1** Tracing

What does this program print?

math_trace.py

```
1 import math
2
3 price = 7.6
4 print(math.floor(price), math.ceil(price))
5 print(math.sqrt(81) + 1)
```

13.2 Importing Names with from

Writing `math.` in front of every call is clear but can get long in a formula that uses `sqrt` five times. Python has a second form of import that brings chosen names straight into your file.

```
1 from math import pi, sqrt
```

- `from` names the module; `import` lists the names you want from it, separated by commas.
- Python still loads the whole module, but it creates only the names you listed, here `pi` and `sqrt`. It does **not** create the name `math`.
- You then use the names with no dot: `sqrt(25)`.

WORKED EXAMPLE A screen's diagonal

A screen 3 units wide and 4 high has a diagonal of the square root of $3^2 + 4^2$.

from_import.py

```
1 from math import pi, sqrt
2
3 width = 3
4 height = 4
5 diagonal = sqrt(width ** 2 + height ** 2)
6 print(f"Diagonal: {diagonal}")
7 print(f"Half a turn is {pi:.4f} radians")
```

```
Diagonal: 5.0
Half a turn is 3.1416 radians
```

Line 1 creates `pi` and `sqrt` in this file. Line 5 works out `3 ** 2 + 4 ** 2`, which is 25, and `sqrt(25)` is 5.0. Line 7 uses `pi` directly with the format specifier `:.4f`.

The two forms make different names:

Line	Names it creates	How you call <code>sqrt</code>	Good when
<code>import math</code>	<code>math</code>	<code>math.sqrt(25)</code>	You use many things from the module
<code>from math import sqrt</code>	<code>sqrt</code>	<code>sqrt(25)</code>	You use one or two things, often
<code>from math import pi, e</code>	<code>pi</code> and <code>e</code>	(not imported)	You want several names but not the module

`import math` keeps every call labelled with where it came from, which helps a reader.

`from math import sqrt` is shorter, but a reader who sees `sqrt` on line 40 must scroll up to find out where it came from. Both are correct; most programmers use `import module` by default and `from` for a few names they use constantly.

There is also `from math import *`, where `*` means "every name in the module". Avoid it. It hides where names came from, and it can silently replace names you already had. `math` has a constant called `e` (the number 2.718...), so:

star_import.py

```
1 e = "excellent"
2 print(e)
3 from math import *
4 print(e)
```

```
excellent
2.718281828459045
```

Line 3 created a new `e` that replaced yours, and nothing warned you. A program that later printed a grade would print a number instead.

KEY IDEA

`from module import name` creates only `name`, used without a dot. It does not create the module's name. Avoid `from module import *`.

WATCH OUT

Using the module name after a `from import`. `from math import floor` creates `floor` and nothing else, so `math.floor` fails with the same error as forgetting the import entirely: `NameError: name 'math' is not defined`. Either call `floor(...)` with no dot, or change the import to `import math`. Question 2 has an example.

PAUSE AND TRY**Question 2** Debugging

This program should print 2 full hours for 135 minutes. It stops with an error instead. Find the bug and fix it.

from_import_bug.py

```
1 from math import floor
2
3 minutes = 135
4 hours = math.floor(minutes / 60)
5 print(f"{hours} full hours")
```

13.3 Short Names with as

Some module names are long, and some clash with names you want to use yourself. Python lets you choose the name an import creates.

`import module as other_name` loads the module but calls it `other_name` in your file. The new name is an **alias**: a second name you pick for something. Here is the standard library's `statistics` module, which has a name long enough to be tiring to type:

WORKED EXAMPLE Averages with an alias

alias_stats.py

```
1 import statistics as stats
2
3 scores = [72, 88, 91, 65]
4 print(f"Mean: {stats.mean(scores)}")
5 print(f"Median: {stats.median(scores)}")
```

```
Mean: 79
Median: 80.0
```

- Line 1 loads `statistics` and creates the name `stats` for it.
- `stats.mean(scores)` adds the four scores (316) and divides by 4. When the result is a whole number and every score is an int, `mean` gives an int, 79.
- `stats.median(scores)` sorts the scores (65, 72, 88, 91) and takes the middle. With an even count there are two middle values, 72 and 88, so it gives their average, 80.0.

`as` also works on a single name from a `from import`, which helps when the module's name for something is not the word you would use:

alias_from.py

```
1 from statistics import mean as average
2
3 temperatures = [18.5, 21.0, 19.0]
4 print(f"Average: {average(temperatures)}")
```

Average: 19.5

`from statistics import mean as average` creates only the name `average`; `mean` and `statistics` do not exist in this file. Some aliases are so common that every Python programmer knows them. You will meet ones like `import numpy as np` in other people's code, where `numpy` is a package that you install yourself (13.10).

KEY IDEA

`as` renames what an import creates. After `import statistics as stats`, only `stats` exists; the name `statistics` does not.

WATCH OUT

Using the original name after an alias.

alias_wrong.py

```
1 import statistics as stats
2
3 ages = [15, 16, 16, 17]
4 print(statistics.mean(ages))
```

```
NameError: name 'statistics' is not defined. Did you forget to import 'statistics'?
```

The module was imported, but under the name `stats`, so `statistics` was never created. Use the alias, `stats.mean(ages)`, or drop `as stats` from line 1.

PAUSE AND TRY**Question 3** Multiple choice

A program starts with `import statistics as stats`. Which line works?

- A. `print(statistics.median([3, 1, 2]))`
- B. `print(stats.median([3, 1, 2]))`
- C. `print(median([3, 1, 2]))`
- D. `print(stats(median([3, 1, 2])))`

13.4 Splitting Your Code Across Files

Here is the good news about modules: any `.py` file you write is already one. If you put some functions in `pricing.py`, another file in the same folder can write `import pricing` and use them exactly as it uses `math`. This is how you split a long program across files.

Why split at all? The Day 12 to-do manager mixed two jobs in one file: functions that change the task list, and the menu loop that talks to the user. Separating them gives:

- **Shorter files**, each about one thing, so you find code faster.
- **Reuse**: a second program (a web version of the to-do list, say) could import the same task functions.
- **Safer changes**: you can change the menu without touching code that already works.

WORKED EXAMPLE A shop project in two files

A school shop adds sales tax to prices. Make a folder `day13/shop/` and put two files in it. The first is a **module**: it only defines things.

`shop/pricing.py`

```
1  """Prices for the school shop: sales tax and money formatting."""
2
3  TAX_RATE = 0.08
4
5
6  def with_tax(price):
7      """Return the price with sales tax added."""
8      return price * (1 + TAX_RATE)
9
10
11 def format_price(amount):
12     """Return an amount as dollars with two decimal places, like $4.32."""
13     return f"${amount:.2f}"
```

- Line 1 is a docstring for the whole module (Day 9 put docstrings on functions; a string at the top of a file describes the file the same way).
- `TAX_RATE` is written in capitals. That is the convention for a constant: a value set once and never changed.
- Running `pricing.py` by itself prints nothing, because it only defines a variable and two functions.

The second file is the **main script**, the file you actually run:

`shop/receipt.py`

```
1  import pricing
2
3  basket = {"notebook": 3.50, "pens": 2.25, "folder": 1.75}
4
5  total = 0
6  for item, price in basket.items():
7      cost = pricing.with_tax(price)
8      total += cost
9      print(f"{item}: {pricing.format_price(cost)}")
10 print(f"Total: {pricing.format_price(total)}")
11 print(f"Tax rate: {pricing.TAX_RATE}")
```

```
notebook: $3.78
pens: $2.43
folder: $1.89
Total: $8.10
Tax rate: 0.08
```

- Line 1, `import pricing`, finds `pricing.py` in the same folder as `receipt.py`, runs it once, and creates the name `pricing`.
- Line 7 calls `pricing.with_tax(3.50)`, which gives 3.78 (3.50×1.08). Line 9 hands that to `pricing.format_price`, which makes the string `$3.78`.
- Line 11 reads the module's variable `pricing.TAX_RATE`, just as you read `math.pi`.

To run it, the terminal must be inside the `shop` folder. Either open `shop` itself in your editor, or move the terminal there with the `cd` command ("change directory"), which works the same on macOS, Linux, and Windows:

```
cd shop
python receipt.py
```

`cd ..` moves back up one folder, to `day13`.

The `from` and `as` forms from 13.2 and 13.3 work on your own modules too:

`shop/receipt_from.py`

```
1 from pricing import format_price, with_tax
2
3 print(format_price(with_tax(10)))
4 print(format_price(with_tax(0.99)))
```

```
$10.80
$1.07
```

0.99×1.08 is 1.0692, which `:.2f` shows as `$1.07`. You can also try your module in the REPL, as long as you start the REPL from the `shop` folder:

```
>>> import pricing
>>> pricing.TAX_RATE
0.08
>>> pricing.format_price(4)
'$4.00'
>>> pricing.format_price(pricing.with_tax(25))
'$27.00'
```

The REPL shows `format_price`'s result with quotes because it is a string; `print` would show it without them.

A few things happen behind the scenes:

- **Where Python looks.** For `import pricing`, Python first looks in the folder of the script you ran, then in the standard library. That is why the two files must be in the same folder.

- **A module runs only once.** The first `import pricing` runs `pricing.py` from top to bottom. If the program imports it again (or another module imports it), Python reuses the module it already loaded instead of running the file again.
- `__pycache__`. After the first import, a folder named `__pycache__` appears next to your files. Python saves a ready-to-run copy of each imported module there so the next import is faster. You never edit it, and deleting it is harmless: Python makes it again.

When a project grows to many modules, you can group them into a sub-folder. A folder of modules that you import from is called a **package**, and you reach a module inside it with a dot, for example `from shop import pricing` for a folder `shop` holding `pricing.py`. You will mostly meet packages as code you install (13.10); for today, a folder of `.py` files that import each other is all you need.

KEY IDEA

Every `.py` file is a module. A script can `import` another file in the same folder by its name without `.py`, then use its functions and variables with a dot.

WATCH OUT

A typo in the module name, or the wrong folder. If no file has the name you import, Python stops at the `import` line:

`shop/receipt_typo.py`

```
1 import prices
2
3 print(prices.format_price(5))
```

```
ModuleNotFoundError: No module named 'prices'
```

A `ModuleNotFoundError` means Python looked in the script's folder and the standard library and found no `prices.py`. Check the spelling (the file is `pricing.py`) and check that both files are in the **same** folder.

Naming your file after a standard module. Because Python looks in your folder **first**, a file of yours called `random.py` hides the standard `random` module. Here a student kept their dice helpers in `random.py`:

`shadow/random.py`

```
1 """My dice helpers (a bad file name: it hides the standard random module)."""
2
3
4 def describe(total):
5     """Return a short description of a dice total."""
6     return f"You rolled {total}."
```

shadow/roll.py

```
1 import random
2
3 total = random.randint(1, 6) + random.randint(1, 6)
4 print(total)
```

AttributeError: module 'random' has no attribute 'randint' (consider renaming '.../shadow/random.py' since it has the same name as the standard library module named 'random' and prevents importing that standard library module)

`import random` loaded the student's file, which has no `randint`. An **AttributeError** means the thing before the dot exists but has nothing with the name after the dot. Recent versions of Python add the "consider renaming" advice; older ones stop after `'randint'`. Never name your files `random.py`, `math.py`, `statistics.py`, or after any other module you import. Rename the file (`dice_helpers.py`) and delete `__pycache__` if it is there.

PAUSE AND TRY**Question 4** Short answer

Jo saves `pricing.py` in `day13`, and `receipt.py` in `day13/shop`. Running `python receipt.py` from the `shop` folder stops with `ModuleNotFoundError: No module named 'pricing'`. Explain why, and give the fix.

13.5 Code That Runs Only When Run Directly

When you write a module, you want to test it: call each function and check the result. The natural place for those test lines is at the bottom of the module. But 13.4 said that importing runs the module's whole file, test lines included.

WORKED EXAMPLE Test lines that leak into another program

A temperature module, with test lines at the bottom:

converter_v1/convert.py

```
1 """Temperature conversions."""
2
3
4 def c_to_f(celsius):
5     """Return a Celsius temperature in Fahrenheit."""
6     return celsius * 9 / 5 + 32
7
8
9 def f_to_c(fahrenheit):
10    """Return a Fahrenheit temperature in Celsius."""
11    return (fahrenheit - 32) * 5 / 9
12
13
14 print("Testing convert.py")
15 print(c_to_f(100))
16 print(f_to_c(32))
```

Running `convert.py` itself is exactly what you want while writing it:

```
Testing convert.py
212.0
0.0
```

But now `weather.py`, in the same folder, imports it:

converter_v1/weather.py

```
1 import convert
2
3 today = 25
4 print(f"Today: {today} C is {convert.c_to_f(today)} F")
```

```
Testing convert.py
212.0
0.0
Today: 25 C is 77.0 F
```

Line 1 of `weather.py` ran all of `convert.py`, so its three test lines printed before `weather.py`'s own output. The program that only wanted a conversion now prints test results it never asked for.

Python solves this with a special variable. Every module automatically has a variable called `__name__` (two underscores on each side). Python sets it in one of two ways:

- In the file you **run** (python `show_name.py`), `__name__` is the string `"__main__"`, meaning "this is the main program".
- In a file that is **imported**, `__name__` is the module's own name, such as `"show_name"`.

Here is a one-line module that prints its own `__name__`, and a script that imports it:

name_demo/show_name.py

```
1 print(f"In show_name.py, __name__ is {__name__}")
```

name_demo/main.py

```

1 import show_name
2
3 print(f"In main.py, __name__ is {__name__}")

```

Running `python main.py`:

```

In show_name.py, __name__ is show_name
In main.py, __name__ is __main__

```

Running `python show_name.py` instead:

```

In show_name.py, __name__ is __main__

```

The same file prints a different `__name__` depending on whether it was run or imported. So a module can ask "was I run directly?" with an ordinary Day 4 `if`:

```

1 if __name__ == "__main__":
2     ...

```

Read it as "only if this file is the one being run". This line is often called the **main guard**. Here is the temperature module with its tests moved under the guard:

converter/convert.py

```

1 """Temperature conversions."""
2
3
4 def c_to_f(celsius):
5     """Return a Celsius temperature in Fahrenheit."""
6     return celsius * 9 / 5 + 32
7
8
9 def f_to_c(fahrenheit):
10    """Return a Fahrenheit temperature in Celsius."""
11    return (fahrenheit - 32) * 5 / 9
12
13
14 if __name__ == "__main__":
15     print("Testing convert.py")
16     print(c_to_f(100))
17     print(f_to_c(32))

```

Run directly, it still tests itself:

```

Testing convert.py
212.0
0.0

```

Imported by the same `weather.py`, the tests stay quiet:

```

Today: 25 C is 77.0 F

```

When `weather.py` imports `convert`, `__name__` in `convert.py` is `"convert"`, the condition is `False`, and the indented block is skipped. The function definitions above the guard still run, so `convert.c_to_f` exists.

The main guard also belongs in main scripts. Many programmers put a script's work in a function called `main()` and call it under the guard, so that even the main script can be imported (by a test, say) without starting its menu. You will see exactly that in today's Practice.

KEY IDEA

`__name__` is `"__main__"` in the file you run and the module's name in a file that is imported. Code under `if __name__ == "__main__":` runs only when the file is run directly.

WATCH OUT

Misspelling the guard. The variable and the string both need two underscores on **each** side. `if __name__ == "main":` (no underscores in the string) is never `True`, so the guarded block never runs, even when you run the file directly, and Python gives no error. `if name == "__main__":` (no underscores on the variable) stops with `NameError: name 'name' is not defined`. If your self-tests print nothing, check the underscores first.

PAUSE AND TRY

Question 5 Multiple choice

`main.py` contains `import planner`. When you run `python main.py`, what is the value of `__name__` while the code inside `planner.py` runs?

- A. `"__main__"`
- B. `"planner"`
- C. `"planner.py"`
- D. `"main"`

13.6 Random Numbers with random

You have used two standard library modules so far, `math` and `statistics`. The rest of today's tour visits four more. Here is the map:

Module	Gives you	Section
<code>math</code>	Square roots, rounding up and down, π	13.1
<code>statistics</code>	Mean, median	13.3
<code>random</code>	Random whole numbers, random picks, shuffling	13.6
<code>datetime</code>	Dates, times, and the gap between them	13.7
<code>os</code>	Files and folders on the computer	13.8
<code>sys</code>	The running program: its arguments, how it ends	13.8

On Day 5 your guessing game's secret number was written into the code, so the game was the same every time. Games, quizzes, and simulations need numbers the program cannot predict. The `random` module makes **pseudo-random** numbers: numbers produced by a calculation that is designed to look unpredictable.

Call	Gives
<code>random.randint(a, b)</code>	A whole number from a to b, including both ends
<code>random.choice(items)</code>	One item picked from a list (or string)
<code>random.shuffle(items)</code>	Puts a list in a random order, changing that list
<code>random.random()</code>	A float that is at least 0 and below 1

Notice the difference from Day 5's `range`: `range(1, 6)` stops **before** 6, but `random.randint(1, 6)` can give 6. A die is `random.randint(1, 6)`.

Because the results change every run, a program's exact output cannot be predicted, and this page never shows you "the" output of a random program. What you **can** check is the facts that hold every time:

WORKED EXAMPLE A hundred dice

`roll_check.py`

```
1 import random
2
3 rolls = [random.randint(1, 6) for _ in range(100)]
4 print(f"Rolls: {len(rolls)}")
5 print(f"Smallest is at least 1: {min(rolls) >= 1}")
6 print(f"Largest is at most 6: {max(rolls) <= 6}")
```

```
Rolls: 100
Smallest is at least 1: True
Largest is at most 6: True
```

- Line 3 is a Day 11 list comprehension that calls `random.randint(1, 6)` 100 times. The loop variable is named `_`: by convention, a single underscore means "I need the loop to run, but I don't use this value".
- The rolls themselves differ every run. Add `print(rolls)` and run the program twice to see that.
- Whatever the rolls are, there are 100 of them, and none is below 1 or above 6, so the three lines are the same every run.

The other functions, checked the same way:

random_tools.py

```
1 import random
2
3 snacks = ["apple", "pretzels", "yogurt"]
4 pick = random.choice(snacks)
5 print(f"The pick is one of the snacks: {pick in snacks}")
6
7 deck = ["A", "K", "Q", "J"]
8 random.shuffle(deck)
9 print(f"Same four cards after shuffling: {sorted(deck) == ['A', 'J', 'K', 'Q']}")
10
11 chance = random.random()
12 print(f"At least 0 and below 1: {0 <= chance < 1}")
```

```
The pick is one of the snacks: True
Same four cards after shuffling: True
At least 0 and below 1: True
```

`random.choice` picks one snack, a different one on different runs. `random.shuffle(deck)` changes the order of `deck` itself, like Day 6's `sort()`, so after it the list holds the same four cards in some order; sorting it gives back the original cards. `random.random()` is handy for chances:

`random.random() < 0.25` is `True` about one time in four.

With `random`, the Day 5 guessing game finally gets a secret it does not know. Only the first line of the game changes (plus the import):

```
1 import random
2
3 secret = random.randint(1, 10)
```

Making random repeatable with seed

Unpredictable numbers make a program hard to test: if a bug shows up on one run, you may never see the same rolls again. The pseudo-random calculation starts from a value called the **seed**.

Normally Python picks a different seed every run. `random.seed(n)` sets it yourself, and the same seed always gives the same sequence of numbers afterwards:

seed_repeat.py

```
1 import random
2
3 random.seed(2026)
4 first = [random.randint(1, 6) for _ in range(10)]
5
6 random.seed(2026)
7 second = [random.randint(1, 6) for _ in range(10)]
8
9 print(f"Same ten rolls after the same seed: {first == second}")
```

```
Same ten rolls after the same seed: True
```

Both lists start right after `random.seed(2026)`, so they are identical, and they are identical on every run of the program. Remove the second `random.seed(2026)` and `second` continues the sequence

instead, so it is (almost always) different. Seed while testing; take the seed out when you want a real game. (The same seed gives the same numbers every time on the same version of Python; a different version may give different ones.)

KEY IDEA

`random.randint(a, b)` includes both `a` and `b`. `choice` picks, `shuffle` reorders a list in place, and `random.seed(n)` makes the numbers repeat exactly from run to run.

WATCH OUT

Saving the result of `shuffle`. `shuffle` changes the list it is given and returns `None`, like Day 6's `sort()`:

`shuffle_none.py`

```
1 import random
2
3 deck = ["A", "K", "Q", "J"]
4 deck = random.shuffle(deck)
5 print(deck)
```

`None`

Line 4 shuffled the cards and then replaced `deck` with `shuffle`'s return value, `None`; the shuffled list is lost. Write `random.shuffle(deck)` on its own line and keep using `deck`.

PAUSE AND TRY**Question 6** Multiple choice

Which expression can give the number 10?

- A. `random.randint(1, 10)`
- B. `random.choice(range(1, 10))`
- C. `random.randint(0, 9)`
- D. `int(random.random() * 10)`

Question 7 Short answer

Sam's dice game rolls exactly the same numbers every time it runs. Which line is probably in the program, and why might Sam have put it there?

13.7 Dates and Times with `datetime`

How many days until an exam? What date is 30 days after a library book is borrowed? Doing this with plain numbers means remembering that months have 28 to 31 days and that some years are leap years. The `datetime` module knows the calendar for you.

It holds a few types; these are the ones you need today, usually imported with `from`:

Name	Holds	Make one with
<code>date</code>	A calendar day	<code>date(2026, 1, 15)</code> (year, month, day)
<code>datetime</code>	A day and a time of day	<code>datetime(2026, 1, 15, 9, 30)</code>
<code>timedelta</code>	A length of time, such as 30 days	<code>timedelta(days=30)</code>

```
>>> from datetime import date, timedelta
>>> launch = date(2026, 1, 15)
>>> launch.year
2026
>>> launch.month
1
>>> launch.weekday()
3
>>> print(launch)
2026-01-15
>>> print(launch + timedelta(days=30))
2026-02-14
>>> launch + timedelta(days=30)
datetime.date(2026, 2, 14)
```

- `launch.year` and `launch.month` read parts of the date, with a dot and no parentheses.
- `launch.weekday()` gives the day of the week as a number, with Monday as 0, so 3 means Thursday.
- `print(launch)` shows the date in year-month-day order, 2026-01-15.
- Adding a `timedelta` gives a new date. 30 days after 15 January is 14 February: Python counted January's 31 days for you.
- In the REPL, a date that is not printed shows as `datetime.date(2026, 2, 14)`, the code that would make it.

WORKED EXAMPLE Days until the exam

Subtracting one date from another gives a `timedelta`, and its `.days` is the gap as a whole number.

`days_until.py`

```
1 from datetime import date
2
3 start = date(2026, 1, 15)
4 exam = date(2026, 5, 11)
5 gap = exam - start
6 print(gap.days)
7 print(f"{gap.days // 7} weeks and {gap.days % 7} days")
```

```
116
16 weeks and 4 days
```

From 15 January to 11 May 2026 is 16 days left in January, then 28 (February), 31 (March), 30 (April), and 11 (May): $16 + 28 + 31 + 30 + 11 = 116$. Line 7 splits that with Day 2's `//` and `%`: $116 // 7$ is 16 weeks, and $116 \% 7$ leaves 4 days.

Times work the same way with `datetime`. To show a date or time in a chosen format, call its `strftime` method ("string from time") with codes such as `%d` (day), `%m` (month), `%Y` (four-digit year), `%H` (hour, 00 to 23), and `%M` (minutes):

clock_times.py

```
1 from datetime import datetime, timedelta
2
3 start = datetime(2026, 1, 15, 9, 30)
4 end = start + timedelta(minutes=95)
5 print(start.strftime("%d/%m/%Y"))
6 print(f"{start.strftime('%H:%M')} to {end.strftime('%H:%M')}")
```

```
15/01/2026
09:30 to 11:05
```

A lesson starting at 9:30 and lasting 95 minutes ends at 11:05: the `timedelta` carried the extra minutes into the hours. Notice the single quotes around `'%H:%M'` inside the f-string on line 6, because the f-string itself uses double quotes.

To get **today's** date, call `date.today()`; for the date and time right now, call `datetime.now()`. Their results depend on when you run the program, so this page never shows them: run `print(date.today())` yourself and you will see your own date.

KEY IDEA

`date(year, month, day)` makes a date. Adding a `timedelta` moves it by a length of time, and subtracting two dates gives a `timedelta` whose `.days` is the gap.

WATCH OUT

A module and a type with the same name. The `datetime` module contains a type also called `datetime`. After `import datetime`, the name `datetime` is the module, and a module cannot be called like a function:

datetime_call.py

```
1 import datetime
2
3 meeting = datetime(2026, 1, 15, 9, 30)
4 print(meeting)
```

```
TypeError: 'module' object is not callable. Did you mean: 'datetime.datetime(...)'?
```

Either write `datetime.datetime(2026, 1, 15, 9, 30)` (module dot type), or import the type with `from datetime import datetime`. Older versions of Python leave out the "Did you mean" part.

PAUSE AND TRY**Question 8** Tracing

A parcel ordered on 27 February 2026 takes 3 days to arrive. What does this program print? (2026 is not a leap year.)

month_end.py

```
1 from datetime import date, timedelta
2
3 order = date(2026, 2, 27)
4 arrives = order + timedelta(days=3)
5 print(arrives)
6 print((arrives - order).days)
```

13.8 The Computer and the Program: os and sys

Your programs so far have lived in their own world of variables. Sometimes a program needs to ask about the computer around it: which folder is the terminal in? Does a file exist? What did the user type after the program's name? Two standard modules answer these questions.

os: files and folders

The `os` module ("operating system") talks to the computer's files and folders. A few everyday functions:

Call	Gives
<code>os.getcwd()</code>	The current working directory : the folder the terminal is in
<code>os.listdir(".")</code>	A list of the names in a folder ("." means "this folder")
<code>os.path.exists(name)</code>	True if a file or folder with that name exists

The answers depend on your computer, so `os.getcwd()` and `os.listdir(".")` print something different for everyone; try them in your `day13` folder. A file name with no folder in front, like `"pricing.py"`, means "in the current working directory". That is why Day 1 told you to run scripts from the folder that holds them. In the `shop` folder from 13.4:

shop/check_files.py

```
1 import os
2
3 for name in ["pricing.py", "prices.py"]:
4     print(f"{name} is here: {os.path.exists(name)}")
```

Run from the `shop` folder:

```
pricing.py is here: True
prices.py is here: False
```

`os.path` is the part of `os` for file names and paths; Day 14 does much more with files and paths.

sys: the running program

The `sys` module ("system") holds facts about the running Python program. The most useful is `sys.argv`, the list of **command-line arguments**: the words typed in the terminal to start the program. Item 0 is always the script's name:

`argv_demo.py`

```
1 import sys
2
3 print(sys.argv)
4 print(len(sys.argv))
```

Run with `python argv_demo.py`:

```
['argv_demo.py']
1
```

Any extra words you type after the file name are added to the list as strings. Run with `python argv_demo.py Ana 15`:

```
['argv_demo.py', 'Ana', '15']
3
```

So a program can take its input from the command line instead of from `input()`.

`sys.exit(message)` ends the program at once, printing the message as an error:

`greet_args.py`

```
1 import sys
2
3 if len(sys.argv) < 2:
4     sys.exit("Usage: python greet_args.py NAME")
5
6 name = sys.argv[1]
7 print(f"Hello, {name}!")
```

Run with no extra words, `python greet_args.py`:

```
Usage: python greet_args.py NAME
```

`sys.argv` has only one item, so line 4 stops the program with a usage message, the traditional way to tell someone how to run a command. Running `python greet_args.py Ana` instead skips the `if`, takes `sys.argv[1]`, which is `"Ana"`, and greets Ana.

`sys` also tells a program where it is running: `sys.platform` is a short string naming the operating system (such as `"win32"`, `"darwin"` for macOS, or `"linux"`), and `sys.version` describes the version of Python. Print them on your own computer to see yours.

KEY IDEA

`os` asks about files and folders (`os.getcwd()`, `os.path.exists(name)`). `sys` asks about the running program: `sys.argv` is the list of words that started it, and `sys.exit()` stops it.

WATCH OUT

Forgetting that `sys.argv[0]` is the script. The first word the user types after the file name is `sys.argv[1]`, not `sys.argv[0]`. And when the user types nothing extra, `sys.argv[1]` does not exist: indexing past the end of a list stops the program with an `IndexError` (Day 6). That is why `greet_args.py` checks `len(sys.argv)` before it reads `sys.argv[1]`.

PAUSE AND TRY**Question 9** Short answer

Someone runs `python greet_args.py Sam Lee`. What is `sys.argv`, and what does the program print? Explain why it does not print `Hello, Sam Lee!`.

13.9 Virtual Environments

The standard library is big, but not everything is in it. Thousands of people publish their own modules for anyone to use, such as tools for fetching web pages or drawing charts. Code written by someone other than you or the Python team is **third-party** code. Before you install any of it, you need a place to put it.

Imagine two projects. An old one needs version 1 of a charting package; a new one needs version 2, which works differently. If both projects share one Python installation, only one version can be installed at a time, and upgrading for the new project breaks the old one. Installing into the Python that came with your computer can also break programs your operating system relies on, and some systems refuse to let you try.

A **virtual environment** is a private, project-only copy of Python's install space: a folder inside your project that holds its own `python` command and its own installed packages. Packages installed while it is active go into that folder and nowhere else. Each project gets its own, so projects can never break each other. The standard library's `venv` module creates one.

The commands below go in the **terminal**, not in the REPL. Start in your project's folder (for today, `day13`).

1. Create it (once per project):

```
python -m venv .venv
```

`-m venv` means "run the `venv` module as a program". `.venv` is the name of the folder to create; the leading dot is the usual choice, and some systems hide names starting with a dot. On macOS and Linux, use `python3` if `python` is not found; on Windows, `py` works too (Day 1).

2. Activate it (in each new terminal you open). The command depends on your system:

```
macOS and Linux:
source .venv/bin/activate

Windows (PowerShell):
.venv\Scripts\Activate.ps1

Windows (Command Prompt):
.venv\Scripts\activate.bat
```

To **activate** a virtual environment means to tell this terminal to use its `python` and its packages. Once it is active, the prompt shows the environment's name in parentheses, such as `(.venv)`, and the plain command `python` means the environment's Python on every system, even where you normally type `python3` or `py`.

If PowerShell refuses with a message that running scripts is disabled on this system, run this once, then activate again:

```
Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
```

3. Work. Run your scripts as usual (`python receipt.py`). VS Code also notices a `.venv` folder in the open project and offers to use it.

4. Deactivate when you are done:

```
deactivate
```

The `(.venv)` disappears, and `python` means your normal Python again. Closing the terminal also ends it.

Three rules keep environments simple:

- **One per project**, in the project's folder, named `.venv`.
- **Never put your own code inside** `.venv`, and never edit files in it. Your scripts stay beside it.
- **Do not share** `.venv`. It is large and only works on the computer that made it. Share a list of what to install instead (13.10), and anyone can rebuild it with one command.

KEY IDEA

`python -m venv .venv` creates a virtual environment; activating it makes `python` and anything you install private to this project. The prompt shows `(.venv)` while it is active.

WATCH OUT

Forgetting to activate. Activation lasts only for one terminal. Open a new terminal tomorrow and the environment is **not** active, so `python` is your normal Python again, and anything you install goes there. Before you install or run, look for `(.venv)` at the start of the prompt. If it is missing, run the activate command for your system again (you never need to create the environment a second time).

PAUSE AND TRY**Question 10** Multiple choice

Yesterday Lee made `.venv` in a project and activated it. Today Lee opens a new terminal in the same folder; the prompt does **not** show `(.venv)`. What should Lee do before installing a package?

- A. Run `python -m venv .venv` again to make a fresh environment
- B. Run the activate command for their system, such as `source .venv/bin/activate`
- C. Nothing: an environment stays active once it has been activated
- D. Delete `.venv`, because it stopped working

13.10 Installing Packages with pip

Third-party code is published on the **Python Package Index, PyPI** (pypi.org), a free public library of hundreds of thousands of packages. A **package**, in this sense, is a bundle of modules that someone has published so others can install it. The tool that downloads and installs packages from PyPI is **pip**, and it comes with Python.

With your virtual environment **active**, install a package by name. `requests`, a popular package for fetching web pages (Day 22 uses it), makes a good first one:

```
python -m pip install requests
```

`python -m pip` means "run pip with this Python", which, with `(.venv)` showing, is the environment's Python, so the package lands in `.venv`. You will see pip download files and then a line that starts with `Successfully installed`. Packages often depend on other packages; pip installs those too, so the message may list several names. Once installed, the package is imported like any module: `import requests`.

Other pip commands you will use:

Command	Does
<code>python -m pip list</code>	Lists every package installed in this environment
<code>python -m pip show requests</code>	Shows one package's version, summary, and location
<code>python -m pip uninstall requests</code>	Removes a package (it asks you to confirm)
<code>python -m pip freeze > requirements.txt</code>	Saves the exact installed packages and versions to a file

```
python -m pip install -r requirements.txt
```

 Installs everything listed in that file

requirements.txt

13.9 said not to share `.venv`. Instead, a project keeps a short text file, named `requirements.txt` by convention, that lists the packages it needs, one per line. You can write it by hand:

```
requests
```

or let pip write it with `python -m pip freeze > requirements.txt`. `freeze` prints each installed package as its name, two equals signs, and the exact version (`requests==` then the version number), and `>` is a terminal feature that sends that output into the file instead of the screen. Pinning exact versions this way means everyone who rebuilds the project gets the same versions you tested with.

Someone who downloads your project then rebuilds the environment with three commands (shown for macOS and Linux; use the Windows activate command from 13.9 on Windows):

```
python -m venv .venv
source .venv/bin/activate
python -m pip install -r requirements.txt
```

The project folder, then, looks like this: your `.py` files, `requirements.txt` beside them, and a `.venv` folder that each person makes on their own computer.

KEY IDEA

With the environment active, `python -m pip install name` installs a package from PyPI into it. `requirements.txt` lists a project's packages, and `python -m pip install -r requirements.txt` installs them all.

WATCH OUT

Typing pip commands at the `>>>` prompt. `pip` is a terminal command, not Python code. Typed at the REPL's `>>>`, `pip install requests` is a `SyntaxError`, just as `python --version` gave an error there on Day 1. Type `exit()` to leave the REPL, then run the command at the ordinary terminal prompt.

Installing in one place and running in another. If you install a package while the environment is not active and then run your script with it active (or the other way round), `import requests` fails with `ModuleNotFoundError: No module named 'requests'`. The package is installed, just in a different Python. Activate the environment, then install and run in that same terminal.

PAUSE AND TRY

Question 11 Short answer

Ana zips up her whole project folder, including `.venv`, to send to Ben. Explain why she should leave `.venv` out, and what she should send instead so Ben can run the project.

Putting it together

A student wants to plan reading for an exam: how many days are left, how many pages to read each day, and a schedule. The planning functions go in a module, and a short main script uses them. Make a folder `day13/exam_planner/` with two files:

`exam_planner/planner.py`

```
1  """Study planning helpers: days left, pages per day, and a schedule."""
2
3  import math
4  from datetime import timedelta
5
6
7  def days_between(start, end):
8      """Return the number of days from start to end."""
9      return (end - start).days
10
11
12  def pages_per_day(pages, days):
13      """Return the pages to read each day to finish in time, rounded up."""
14      return math.ceil(pages / days)
15
16
17  def schedule(start, days, pages_each):
18      """Return a list of (date, pages) pairs, one for each study day."""
19      return [(start + timedelta(days=offset), pages_each) for offset in range(days)]
20
21
22  if __name__ == "__main__":
23      from datetime import date
24
25      print(days_between(date(2026, 3, 1), date(2026, 3, 31)))
26      print(pages_per_day(100, 30))
```

`exam_planner/main.py`

```
1  from datetime import date
2
3  import planner
4
5  start = date(2026, 4, 20)
6  exam = date(2026, 5, 11)
7  pages = 250
8
9  days = planner.days_between(start, exam)
10 each = planner.pages_per_day(pages, days)
11 print(f"{days} days until the exam: read {each} pages a day.")
12 for day, amount in planner.schedule(start, 3, each):
13     print(f"{day.strftime('%d/%m')}: {amount} pages")
14 print("...")
```

Running `python main.py` from the `exam_planner` folder:

```
21 days until the exam: read 12 pages a day.
20/04: 12 pages
21/04: 12 pages
22/04: 12 pages
...
```

Running `python planner.py` runs its self-test instead:

```
30
4
```

Piece	What it does	Idea
<code>import math in planner.py</code>	Gives <code>math.ceil</code> for rounding up	13.1
<code>from datetime import timedelta</code>	Creates only the name <code>timedelta</code>	13.2
<code>import planner in main.py</code>	Runs <code>planner.py</code> once and creates the name <code>planner</code>	13.4
<code>days_between(start, exam)</code>	11 May minus 20 April is a <code>timedelta</code> of 21 days; <code>.days</code> is 21	13.7
<code>pages_per_day(250, 21)</code>	250 / 21 is 11.9..., and <code>math.ceil</code> makes it 12, so the reading finishes	13.1
<code>schedule(start, 3, 12)</code>	A comprehension that adds 0, 1, then 2 days to the start date	13.7
<code>if __name__ == "__main__":</code>	The self-test runs for <code>python planner.py</code> but not when <code>main.py</code> imports it	13.5

- The self-test checks March, which has 31 days: from 1 to 31 March is 30 days, and 100 pages over 30 days is 3.33..., rounded **up** to 4. Rounding down to 3 a day would leave 10 pages unread on exam day.
- `main.py` uses fixed dates so that this page can show its output. In real use, `start = date.today()` (13.7) makes the plan start from the day you run it.
- The self-test needs `date` but the module's functions do not, so `from datetime import date` sits inside the guard. Imports usually go at the top, but an import only the self-test uses can live with the self-test.

Vocabulary

Term	Definition	Example
module	A file of Python code that other programs can import	<code>math</code> , <code>pricing.py</code>
standard library	The modules that come with Python	<code>math</code> , <code>random</code> , <code>datetime</code>
import	Load a module so a program can use it	<code>import math</code>
from import	Import chosen names from a module, used without a dot	<code>from math import sqrt</code>
alias	A name you choose for an import with <code>as</code>	<code>import statistics as stats</code>

constant	A variable meant never to change, written in capitals	<code>TAX_RATE = 0.08</code>
main script	The file you run; it imports the others	<code>receipt.py</code>
ModuleNotFoundError	Python could not find a module with that name	<code>import prices</code>
AttributeError	The thing before the dot has nothing with the name after it	<code>random.randint</code> from your own file
<code>__pycache__</code>	A folder where Python saves ready-to-run copies of imported modules	appears after an import
package	A folder of modules; or a bundle of modules published to install	<code>requests</code>
<code>__name__</code>	" <code>__main__</code> " in the file being run, the module's name when imported	<code>"pricing"</code>
main guard	<code>if __name__ == "__main__":</code> , code that runs only when run directly	self-tests, <code>main()</code>
pseudo-random	Numbers from a calculation designed to look unpredictable	<code>random.randint(1, 6)</code>
seed	The starting value for random numbers; the same seed repeats them	<code>random.seed(2026)</code>
<code>timedelta</code>	A length of time, added to or subtracted from dates	<code>timedelta(days=30)</code>
current working directory	The folder the terminal (and so the program) is in	<code>os.getcwd()</code>
command-line arguments	Words typed after the script name, in <code>sys.argv</code>	<code>python greet_args.py Ana</code>
<code>cd</code>	The terminal command that changes the current folder	<code>cd shop</code> , <code>cd ..</code>
third-party	Code published by someone other than you or the Python team	<code>requests</code>
virtual environment	A project-only folder with its own Python and installed packages	<code>.venv</code>
activate	Make a terminal use a virtual environment	<code>source .venv/bin/activate</code>
PyPI	The Python Package Index, where packages are published	<code>pypi.org</code>
<code>pip</code>	The tool that installs packages from PyPI	<code>python -m pip install requests</code>
<code>requirements.txt</code>	A file listing the packages a project needs	<code>requests</code>

Summary

- A **module** is a file of Python code. `import math` creates the name `math` (use `math.sqrt`); `from math import sqrt` creates only `sqrt`; `as` picks the name an import creates.
- Every `.py` file you write is a module, so a project can be split into a module of functions and a **main script** that imports it, kept in the same folder. Never name a file after a module you import.
- `__name__` is `"__main__"` only in the file being run, so code under `if __name__ == "__main__":` runs when the file is run but not when it is imported.
- The standard library covers everyday jobs: `math` (`sqrt`, `ceil`, `floor`, `pi`), `random` (`randint` includes both ends, `choice`, `shuffle`, `seed`), `datetime` (`date`, `timedelta`, `strftime`), `os` (folders and files), and `sys` (`argv`, `exit`).
- A **virtual environment** (`python -m venv .venv`, then activate) gives each project its own installed packages. With it active, `python -m pip install` adds packages from **PyPI**, and `requirements.txt` records them so anyone can rebuild the environment.

Practice

Do these in your `day13` folder. Type each program yourself, run it, and only then open the answer.

Question 12 Short answer

A to-do manager in two files. Move the Day 12 to-do manager into a project folder `todo_app/`. Put the task functions (`show_menu`, `add_task`, `list_tasks`, `complete_task`, `delete_task`) in `tasks.py`, and the menu loop, `main()`, in `main.py`, which imports them. Give each file a main guard: `tasks.py` should test itself when run directly, and `main.py` should call `main()` only when run directly.

Question 13 Short answer

Dice tally. Simulate rolling two dice 1,000 times with `random`. Count how many times each total from 2 to 12 comes up, and print the tallies in order of total.

Question 14 Short answer

Your first environment. In your `day13` folder, create a virtual environment, activate it, install one package from PyPI, and check that it was installed.

Review questions

Question 15 Tracing

These two files are in the same folder. What does `python app.py` print?

review_app/helpers.py

```
1 print("Loading helpers")
2
3
4 def double(number):
5     """Return twice the number."""
6     return number * 2
7
8
9 if __name__ == "__main__":
10     print("Testing:", double(4))
```

review_app/app.py

```
1 import helpers
2 import helpers
3
4 print("App starts")
5 print(helpers.double(21))
```

Question 16 Debugging

This program should print the date of a deadline. Find the bug and fix it.

review_from_bug.py

```
1 from datetime import date
2
3 deadline = datetime.date(2026, 3, 1)
4 print(deadline)
```

Question 17 Tracing

A summer course runs from 1 June to 1 July 2026. What does this program print?

review_weeks.py

```
1 import math
2 from datetime import date
3
4 start = date(2026, 6, 1)
5 end = date(2026, 7, 1)
6 days = (end - start).days
7 print(days)
8 print(days / 7)
9 print(math.ceil(days / 7))
```

Question 18 Multiple choice

A file starts with these two lines:

```
1 import math as m
2 from random import randint
```

Which line works later in the file?

- A. `print(math.ceil(2.5))`
- B. `print(random.randint(1, 6))`
- C. `print(m.ceil(2.5) + randint(1, 6))`
- D. `print(m.randint(1, 6))`

Question 19 Short answer

Kai writes a program to practise square roots and saves it as `math.py`. It starts with `import math`, and stops with `AttributeError: module 'math' has no attribute 'sqrt'`. Kai checks: the spelling of `sqrt` is right. Explain what went wrong and how to fix it.

Question 20 Multiple choice

You download a classmate's project. It has `main.py` and `requirements.txt`, but no `.venv`. In which order should you run these commands in the project folder?

- **P:** `python -m pip install -r requirements.txt`
- **Q:** `python -m venv .venv`
- **R:** the activate command for your system

- **S:** `python main.py`

- A. $P \rightarrow Q \rightarrow R \rightarrow S$
- B. $Q \rightarrow R \rightarrow P \rightarrow S$
- C. $Q \rightarrow P \rightarrow R \rightarrow S$
- D. $R \rightarrow Q \rightarrow P \rightarrow S$

Go further

The official Python Tutorial covers modules, `__name__`, and packages in Modules, and environments and pip in Virtual Environments and Packages. Its Brief Tour of the Standard Library shows more modules than today's tour.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can import a module and use its functions with a dot. Reread: 13.1 Modules and import
- ☐ I can choose between `math.ceil`, `math.floor`, and `round`. Reread: 13.1 Modules and import
- ☐ I can say which names from `... import` creates. Reread: 13.2 Importing Names with `from`
- ☐ I can give an import a different name with `as`. Reread: 13.3 Short Names with `as`
- ☐ I can split a program into a module and a main script in one folder. Reread: 13.4 Splitting Your Code Across Files
- ☐ I can explain what if `__name__ == "__main__"` does and when to use it. Reread: 13.5 Code That Runs Only When Run Directly
- ☐ I can use `randint`, `choice`, `shuffle`, and `seed`, and say what output is certain. Reread: 13.6 Random Numbers with `random`
- ☐ I can make dates, add a `timedelta`, and find the days between two dates. Reread: 13.7 Dates and Times with `datetime`
- ☐ I can check for a file with `os` and read command-line arguments with `sys`. Reread: 13.8 The Computer and the Program: `os` and `sys`
- ☐ I can create, activate, and deactivate a virtual environment. Reread: 13.9 Virtual Environments
- ☐ I can install a package with `pip` and record it in `requirements.txt`. Reread: 13.10 Installing Packages with `pip`