Module 4 of 20

Functions

def and return, the full signature with defaults, *args, **kwargs and keyword-only parameters, LEGB scope with closures and late binding, recursion with memoisation, lambdas and the key-function idiom, and the type hints and docstrings that publish a function's contract.

A Python function is an object: created by `def`, bound to a name, passed around like any value, and called with parentheses. That is the fact this module keeps returning to, because it explains the key-function idiom, closures, decorators, dispatch tables and the mutable-default trap all at once. The module also fixes the one scoping rule — assignment makes a name local to the whole function — that produces `UnboundLocalError`, motivates `global` and `nonlocal`, and gives closures their late-binding behaviour.

Defining functions covers `return`, the `None` that comes back otherwise, the in-place methods that return it on purpose, and the discipline of I/O in `main` with pure helpers. Parameters and arguments takes the signature apart: positional and keyword, defaults evaluated once, `*args` and `**kwargs`, `*` and `/`, and unpacking at the call. Scope and closures states LEGB, the two keywords, and the loop-variable trap with its fixes. Recursion gives the base-case discipline, the call stack, the depth limit and `functools.cache`. Lambdas and higher-order functions covers `key=`, `map` / `filter`, `partial`, `reduce` and `operator`. Type hints and docstrings introduces the annotation vocabulary and PEP 257.

The exercises are whole programs: statistics helpers that return tuples, a pipeline of functions held in a list, an argument echo built on `*args` / `**kwargs`, the mutable-default bug fixed with `None`, a counter factory with `nonlocal`, late binding demonstrated and fixed, a recursive flattener, cached Fibonacci, multi-key sorting, a `partial`-built pipeline, an annotation report, and a run-time checker driven by `__annotations__`. The checkpoint adds memoised grid paths around obstacles, a keyword-only signature exercised through calls that succeed or raise, and function composition from closure factories.

CONTENTS

1	Defining functions — def, return and functions as values	12 min
2	Parameters and arguments — positional, keyword, defaults, *args and **kwargs	15 min

3	Scope and closures — LEGB, global, nonlocal and late binding	15 min
4	Recursion — base cases, the call stack and memoisation	14 min
5	Lambdas and higher-order functions	13 min
6	Type hints and docstrings — the contract a function publishes	12 min
✓	Checkpoint — Functions	25 min

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

LESSON 1 · 12 MIN

Defining functions — def, return and functions as values

A function in Python is an object created by a `def` statement, bound to a name like any other value, and called with parentheses. That one sentence explains most of what surprises people later: a function can be passed to another function, stored in a list, returned from a call, given attributes, and redefined. This lesson covers the statement, what `return` does and does not do, the `None` that every function returns when it says nothing, docstrings, and the discipline — one job, no hidden state, inputs in and a result out — that makes a function worth writing.

The statement

```
def area(width, height):
    """Return the area of a width × height rectangle."""
    return width * height

print(area(3, 4))      # 12
```

`def` runs when the interpreter reaches it: it compiles the body, creates a function object, and binds the name `area` to it. The body does not run until the function is called. That is why a function must be defined *above* the code that calls it at module level (a call at the top of a file to a function defined below it is a `NameError`), but functions can call each other in any order as long as every `def` has executed before the first *call*.

The parameters — `width`, `height` — are local names bound to the arguments when the function is called. Everything assigned inside the body is local too, created on entry and gone on return (Module 4 lesson 3).

---

return

`return expr` evaluates the expression, ends the call and hands the value back. A `return` with no expression, or falling off the end of the body, returns `None`. There is no way to return nothing at all — the caller always gets an object.

```python
def find_index(xs, target):
    for i, x in enumerate(xs):
        if x == target:
            return i          # leaves the loop and the function at once
    return -1                 # the "not found" answer; without it the caller
```

Several values are returned as one tuple, and unpacked at the call site:

```python
def min_max(xs):
    return min(xs), max(xs)

lo, hi = min_max([3, 1, 4])
```

`return` inside a loop is the cleanest early exit (Module 3). `return` inside a `try` still runs the `finally` block (Module 10). And code after a `return` in the same block never runs — an unreachable line is a bug, not a comment.

The None trap

A function that computes something but forgets to `return` it hands back `None`, and the caller's next line fails with `AttributeError: 'NoneType' object has no attribute …` or `TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'`. The same trap hides in methods that work in place: `xs.sort()`, `xs.append(x)`, `d.update(...)`, `random.shuffle(xs)` all return `None` by design, so `xs = xs.sort()` throws the list away. The rule the standard library follows — a method that mutates returns `None`; a method that builds a new object returns it — is worth adopting in your own code.

Functions are objects

```python
def double(x):
    return 2 * x

f = double                # another name for the same function
print(f(21))              # 42
ops = [double, str, len]  # functions in a list
print(double.__name__)    # 'double'
print(type(double))       # <class 'function'>
```

Because functions are values, they can be arguments (`sorted(words, key=len)`, `map(double, xs)`), return values (a function that builds and returns another — Module 4 lesson 3), and dictionary values (a dispatch table, Module 3). Calling is the only thing parentheses do: `double` is the function, `double(3)` is a call. Forgetting the parentheses passes the function where its result was wanted, and since a function object is truthy, `if is_ready:` is a silent bug.

Docstrings and help

The first statement of a body, if it is a string, is the docstring; `help(area)` and editors show it, and `area.__doc__` holds it. Write it as what the function *does for the caller*, in the imperative: "Return the area…", not "This function computes…". A one-line docstring is enough for a small function; a longer one describes parameters, the return value and the exceptions raised (Module 15 covers the conventions). Type hints — `def area(width: float, height: float) → float:` — are the other half of the contract and are covered in this module's last lesson.

What makes a function good

One job. A function that reads input, computes and prints does three, and none of them can be tested or reused alone. Read in `main`, compute in a function that takes values and returns values, print in `main`.

No hidden inputs. A function that reads a global has an input its signature does not show; pass it as a parameter.

No surprise outputs. A function that returns a value and also mutates its argument or prints is doing two things; pick one and say so in the name (`sort` versus `sorted`).

A name that says what it returns, for functions that return: `is_prime`, `parse_line`, `best_score`. A verb for functions that act: `save`, `print_report`.

```
def main():
    n = int(input())
    scores = [int(input()) for _ in range(n)]
    lo, hi = min_max(scores)
    print(f"{lo} {hi}")
```

That shape — `main` does I/O, helpers are pure — is how every reference solution in this track is written, and it is what makes the helpers testable in Module 18.

Pitfalls

Calling a function above its `def` at module level.

`xs = xs.sort()` and other in-place methods assigned.

A missing `return` on one path (the loop found nothing, the `else` branch was not written).

Passing `f` where `f()` was meant.

A helper that prints its result instead of returning it — the caller cannot use it.

A docstring that repeats the name instead of describing the contract.

Key takeaways

`def` creates a function object when executed and binds a name; the body runs only when called.

`return` ends the call with a value; no `return` means `None`; several values come back as a tuple.

In-place methods return `None` by convention; never assign their result.

Functions are values: pass them, store them, return them; parentheses call.

One job, explicit inputs, a returned result, a docstring stating the contract; I/O in `main`.

EXERCISE 1

Statistics helpers

Write three pure functions and a `main` that does all the input and output. `min_max(xs)` returns the smallest and largest values **as a tuple**; `mean(xs)` returns the average; `spread(xs)` returns `max - min`. `main` reads the numbers, unpacks the tuple, and prints the report. For an empty input print `no data`.

Input: one line of integers (possibly empty). **Output:** `min <a> max <b> mean <m> spread <s>` with the mean to two decimals, or `no data`.

```
3 1 4 1 5
```

prints

```
min 1 max 5 mean 2.80 spread 4
```

INPUT 1

```
3 1 4 1 5
```

OUTPUT 1

```
min 1 max 5 mean 2.80 spread 4
```

REFERENCE SOLUTION

```python
def min_max(xs):
    return min(xs), max(xs)


def mean(xs):
    return sum(xs) / len(xs)


def spread(xs):
    lo, hi = min_max(xs)
    return hi - lo


def main():
    xs = list(map(int, input().split()))
    if not xs:
        print("no data")
        return
    lo, hi = min_max(xs)
    print(f"min {lo} max {hi} mean {mean(xs):.2f} spread {spread(xs)}")


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

EXERCISE 2

A pipeline of functions

Functions are values. The starter defines `double`, `square`, `negate` and `increment`. Read a line naming functions in order (possibly empty), build a **list of function objects** from a dictionary of names, then read a line of integers and pass each through the whole list in order.

Input: a line of function names, then a line of integers. **Output:** `<x> → <result>` per integer.

```
double square
1 2 3
```

prints

```
1 → 4
2 → 16
3 → 36
```

INPUT 1

```
double square
1 2 3
```

OUTPUT 1

```
1 → 4
2 → 16
3 → 36
```

REFERENCE SOLUTION

```python
def double(x):
    return 2 * x


def square(x):
    return x * x


def negate(x):
    return -x


def increment(x):
    return x + 1


TABLE = {"double": double, "square": square, "negate": negate, "increment": incremer

names = input().split()
pipeline = [TABLE[name] for name in names]
for x in map(int, input().split()):
    value = x
    for fn in pipeline:
        value = fn(value)
    print(f"{x} → {value}")
```

QUIZ

1. What does this print?

```
def f(xs):
    xs.append(1)

result = f([])
print(result)
```

A. `[1]`

B. `None`

C. `[]`

D. `TypeError`

2. Why does `xs = xs.sort()` lose the list?

A. `sort` returns a sorted copy but `xs` is reassigned too early

B. `sort` sorts in place and returns `None`, so `xs` is rebound to `None`

C. `sort` raises an exception for non-empty lists

D. It does not — `xs` is sorted

3. What happens?

```
print(area(2, 3))

def area(w, h):
    return w * h
```

A. Prints `6`

B. `NameError: name 'area' is not defined`

C. `SyntaxError`

D. Prints `None`

4. What is `type(len)` and what does `if len:` do?

A. `builtin_function_or_method`; the condition is always true because a function object is truthy

B. `int`; the condition tests whether the length is non-zero

C. It raises `TypeError` because `len` needs an argument

D. `NoneType`; the condition is false

5. Which docstring follows the convention?

A. `"""This function takes a list and returns its total."""`

B. `"""Return the sum of the numbers in xs."""`

C. `# Return the sum of the numbers in xs`

D. `"""total()"""`

ANSWER KEY · DEFINING FUNCTIONS — DEF, RETURN AND FUNCTIONS AS VALUES

1. B The function has no `return`, so the call evaluates to `None`. The list was mutated, but the caller never kept a reference to it.

2. B In-place methods return `None` by convention so that they are not confused with the copying versions. Either call `xs.sort()` alone or use `xs = sorted(xs)`.

3. B A `def` binds its name only when executed. The call on line 1 runs before the `def` on line 3 has been reached, so the name does not exist yet.

4. A Without parentheses `len` is the function object itself, which is truthy like any object without `__bool__`. The bug is silent, which is why `is_ready` versus `is_ready()` is worth a second look.

5. B An imperative one-line summary of what the function does for the caller. The first restates the signature in prose, the third is a comment (not stored in `__doc__`), and the last repeats the name.

LESSON 2 · 15 MIN

Parameters and arguments — positional, keyword, defaults, *args and **kwargs

Python's calling convention is richer than most languages': arguments can be passed by position or by name, parameters can have defaults, a function can accept any number of positional or keyword arguments, and a signature can *forbid* one style or the other. Every part of that is useful and one part — a mutable default — is the most famous bug in the language. This lesson takes the signature apart piece by piece, gives the ordering rule that ties the pieces together, and shows the unpacking that goes the other way, from a list or dict into a call.

Positional and keyword arguments

```
def greet(name, greeting):
    return f"{greeting}, {name}!"

greet("Ada", "Hello")                  # positional: matched by order
greet(greeting="Hi", name="Ada")       # keyword: matched by name, any order
greet("Ada", greeting="Hi")            # mixed: positionals first
```

Positional arguments must come before keyword arguments in a call. Passing the same parameter twice, or a keyword that does not exist, is a `TypeError`. Keyword arguments make a call readable when the parameters are of the same type — `move(x=3, y=4)` rather than `move(3, 4)` — and let you skip defaults in the middle of a signature.

Default values

```
def power(base, exponent=2):
    return base ** exponent

power(3)          # 9
power(3, 3)       # 27
power(3, exponent=3)
```

Parameters with defaults must come after those without (`def f(a=1, b)` is a syntax error). The default expression is evaluated **once, when the `def` runs**, and the resulting object is stored with the function and reused by every call that omits the argument. For an immutable default that is invisible. For a mutable one it is the bug:

```
def append_to(x, xs=[]):      # one list, created at def time, shared by all c
    xs.append(x)
    return xs

print(append_to(1))           # [1]
print(append_to(2))           # [1, 2]   — not [2]
```

The idiom is a `None` sentinel and a fresh object inside:

```
def append_to(x, xs=None):
    if xs is None:
        xs = []
    xs.append(x)
    return xs
```

The same applies to `{}`, `set()`, and any call whose result is meant to be fresh per invocation (`time.time()` as a default is evaluated once, at definition).

---

*args and **kwargs

A parameter named `*args` collects extra positional arguments into a **tuple**; `**kwargs` collects extra keyword arguments into a **dict**. The names are conventional; the stars are the syntax.

```
def log(level, *messages, **fields):
    text = " ".join(str(m) for m in messages)
    extras = ", ".join(f"{k}={v}" for k, v in fields.items())
    return f"[{level}] {text} {extras}".rstrip()

log("INFO", "started", "ok", user="ada", pid=42)
# '[INFO] started ok user=ada, pid=42'
```

`*args` is how `print` accepts any number of values and how `max(1, 2, 3)` works. `**kwargs` is how a wrapper forwards options it does not itself understand. A function that takes `*args, **kwargs` and passes them on unchanged is the shape of every decorator (Module 16).

Keyword-only and positional-only parameters

A bare `*` in the signature makes everything after it keyword-only; a `/` makes everything before it positional-only:

```
def connect(host, port, *, timeout=10, retries=3):   # timeout and retries mus
    ...
connect("db", 5432, timeout=5)
connect("db", 5432, 5)          # TypeError: takes 2 positional arguments but

def divmod_(a, b, /):                                # a and b cannot be passe
    ...
```

Keyword-only parameters stop a call like `connect("db", 5432, 5, 2)` whose meaning nobody can read; they are the right choice for boolean flags and options. Positional-only parameters (3.8) let a library rename its parameters later without breaking callers, and are what most built-ins use (`len(obj=xs)` is a `TypeError`).

The full ordering of a signature is: positional-only, `/`, ordinary, `*args` or `*`, keyword-only, `**kwargs`:

```
def f(pos_only, /, normal, *args, kw_only, **kwargs): ...
```

Unpacking at the call

The stars work in the other direction too. `f(*seq)` spreads a sequence into positional arguments; `f(**mapping)` spreads a dict into keyword arguments:

```
point = (3, 4)
move(*point)                          # move(3, 4)

options = {"timeout": 5, "retries": 1}
connect("db", 5432, **options)        # connect("db", 5432, timeout=5, retrie

print(*range(5))                      # 0 1 2 3 4
first, *rest = [1, 2, 3]              # the same star in an assignment
```

A common shape is a function that takes a list and a function that takes separate arguments meeting through a star: `max(*xs)` versus `max(xs)` both work, and the former fails on an

empty list with a different error.

Arguments are objects, not copies

Every argument is passed the same way: the parameter name is bound to the same object the caller passed. Nothing is copied. A function can mutate a list or dict it receives and the caller sees it; it cannot rebind the caller's name (Module 2 lesson 4). Say in the docstring which you do — "modifies `xs` in place" or "returns a new list" — and never both.

Pitfalls

- A mutable default. Use `None` and create inside.
- Positional arguments after keyword ones in a call.
- Too many positional arguments where keyword-only was meant; `*` protects.
- `f(xs)` versus `f(*xs)` — passing one list versus its elements.
- Forgetting that `*args` is a tuple (immutable) and `**kwargs` a dict.
- Shadowing a parameter by assigning to it and then expecting the caller to see the change.

Key takeaways

- Arguments match by position then by name; keywords make same-typed arguments readable.
- Defaults are evaluated once at `def` time — never use a mutable default; use `None`.
- `*args` gathers extra positionals into a tuple, `**kwargs` extra keywords into a dict; `*` and `/` make parameters keyword-only or positional-only.
- Signature order: positional-only `/`, ordinary, `*args`, keyword-only, `**kwargs`.
- `f(*seq)` and `f(**dict)` unpack into a call; every argument is the caller's object, not a copy.

EXERCISE 1

Echo the arguments

Write `describe(*args, **kwargs)` returning a two-line description of what it was called with, then call it by **unpacking** a list and a dict built from one input line: a token containing `=` is a keyword argument (`name=value`, both kept as strings), any other token is positional.

Input: one line of tokens (possibly empty). **Output:** `positional: <values separated by spaces>` (or `(none)`), then `keyword: <k=v pairs separated by ", ">` (or `(none)`), in the order given.

```
1 2 name=ada age=3
```

prints

```
positional: 1 2
keyword: name=ada, age=3
```

INPUT 1

```
1 2 name=ada age=3
```

OUTPUT 1

```
positional: 1 2
keyword: name=ada, age=3
```

REFERENCE SOLUTION

```python
def describe(*args, **kwargs):
    pos = " ".join(args) if args else "(none)"
    kw = ", ".join(f"{k}={v}" for k, v in kwargs.items()) if kwargs else "(none)"
    return f"positional: {pos}\nkeyword: {kw}"


tokens = input().split()
positional = [t for t in tokens if "=" not in t]
keyword = dict(t.split("=", 1) for t in tokens if "=" in t)
print(describe(*positional, **keyword))
```

EXERCISE 2

The mutable default, fixed

The starter's `collect(word, bag=[])` is the classic bug: the default list is created once and shared. Each input line is a group of words; for each line the program calls `collect` for the first word **without** passing a bag, and for the following words passes the bag back in, then prints the bag. With the bug, the second line's bag still contains the first line's words. Fix the function with the `None` idiom so every line starts fresh — do not change the calling code.

Input: lines of words. **Output:** one line per input line: the words collected, separated by spaces.

```
a b
c d
```

prints

```
a b
c d
```

INPUT 1

```
a b
c d
```

OUTPUT 1

```
a b
c d
```

REFERENCE SOLUTION

```
import sys


def collect(word, bag=None):
    if bag is None:
        bag = []
    bag.append(word)
    return bag


for line in sys.stdin:
    words = line.split()
    if not words:
        continue
    bag = collect(words[0])
    for word in words[1:]:
        bag = collect(word, bag)
    print(" ".join(bag))
```

QUIZ

1. What does this print?

```
def add(x, xs=[]):
    xs.append(x)
    return xs

add(1)
print(add(2))
```

A. `[2]`

B. `[1, 2]`

C. `[[1], 2]`

D. `TypeError`

2. Which call is a `TypeError` for `def f(a, b=2, *, c=3)`?

A. `f(1)`

B. `f(1, 5, c=6)`

C. `f(1, 5, 6)`

D. `f(a=1, c=0)`

3. Inside `def f(*args, **kwargs)`, what are `args` and `kwargs`?

A. A list and a list

B. A tuple and a dict

C. A dict and a tuple

D. Two tuples

4. What is the difference between `max(xs)` and `max(*xs)` for `xs = [3, 1, 2]`?

A. None — both return 3

B. `max(xs)` compares the elements; `max(*xs)` passes the list as a single argument

C. `max(*xs)` is a `TypeError`

D. `max(xs)` returns the list

5. Which signature order is valid?

A. `def f(a=1, b)`

B. `def f(**kw, *args)`

C. `def f(a, /, b, *, c)`

D. `def f(*, a, /, b)`

ANSWER KEY · PARAMETERS AND ARGUMENTS — POSITIONAL, KEYWORD, DEFAULTS, *ARGS AND **KWARGS

1. B The default list is created once when `def` runs and shared by every call that omits `xs`. The first call appended `1` to it; the second sees that same list.

2. C Everything after the bare `*` is keyword-only, so a third positional argument is rejected. The other calls supply `a` and use defaults or keywords correctly.

3. B Extra positional arguments are collected into a tuple; extra keyword arguments into a dict keyed by name. The names are conventional; the stars are the syntax.

4. A `max` accepts either one iterable or several positional values, and the star spreads the list into the latter form. They differ only on an empty list (`max([])` raises `ValueError`; `max(*[])` is `max()` with no arguments, a `TypeError`).

5. C Positional-only parameters come first, then `/`, then ordinary ones, then `*` and keyword-only ones, then `**kwargs`. A default before a non-default, `**kw` before `*args`, and `/` after `*` are all syntax errors.

LESSON 3 · 15 MIN

Scope and closures — LEGB, global, nonlocal and late binding

Where does a name come from? Python answers with one rule — **LEGB**: local, enclosing, global, built-in — and one twist: *assignment* anywhere in a function makes the name local to the whole function, even on lines before the assignment. That twist produces `UnboundLocalError`, the `global` and `nonlocal` keywords exist to override it, and closures — functions that remember the enclosing scope they were created in — follow from the same rule. This lesson fixes the rule, the two keywords, the closure mechanism, and the late-binding behaviour that surprises everyone who builds functions in a loop.

The four scopes

```
import builtins          # B: built-in — len, print, range, ValueError

counter = 0              # G: module (global) scope

def outer():
    factor = 10          # E: enclosing scope, from inner's point of view
    def inner(x):
        y = x * factor   # L: local — x, y
        return y
    return inner
```

A name is looked up innermost-first: local, then any enclosing function scopes, then the module, then built-ins; the first hit wins. That order is why a local `list = []` hides the built-in for the rest of the function, and why `print` works in every function without an import. Classes have their own scope rules (Module 8); comprehensions have their own local scope (Module 6), which is why the loop variable of a comprehension does not leak.

Assignment makes a name local

The compiler decides a name's scope by scanning the *whole* function body: if the name is assigned anywhere in it — by `=`, `+=`, `for`, `import`, `def`, `with … as`, `except … as` — it is local everywhere in that function.

```
count = 0

def bump():
    count += 1        # UnboundLocalError: cannot access local variable 'count
                      # where it is not associated with a value
```

`count += 1` is an assignment, so `count` is local to `bump`, and reading it before assigning is an error — the global is not consulted at all. Reading without assigning is fine:

```
def show():
    print(count)      # reads the global: 0
```

This is the rule to internalise: a function can *read* outer names freely; the moment it *assigns* one, that name is a new local unless declared otherwise.

---

global and nonlocal

`global name` declares that assignments inside the function bind the *module-level* name; `nonlocal name` declares they bind the name in the nearest *enclosing function* scope.

```
count = 0

def bump():
    global count
    count += 1

def make_counter():
    n = 0
    def step():
        nonlocal n
        n += 1
        return n
    return step
```

`nonlocal` needs an existing binding in an enclosing function (not the module); `global` creates the module name if it does not exist. Both are rare in good code: a function that needs `global` to assign is usually a function that should return a value, or a piece of state that should be an object's attribute. `nonlocal` has one legitimate home — a closure that keeps a counter or accumulator — and `make_counter` above is that case.

Closures

A closure is a function that refers to names from an enclosing function scope and keeps them alive after that scope has returned:

```
def multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply

double = multiplier(2)
triple = multiplier(3)
print(double(5), triple(5))     # 10 15
```

Each call to `multiplier` creates a new `factor` and a new `multiply` that closes over it. The captured names live in *cells* (`double.__closure__` shows them), and the closure reads the cell's *current* value each time it runs — the variable is captured, not its value at creation. That is the mechanism behind the next section, and behind decorators (Module 16), callbacks, and `functools.partial`.

Late binding in loops

```
adders = []
for i in range(3):
    adders.append(lambda x: x + i)

print([f(10) for f in adders])    # [12, 12, 12] — not [10, 11, 12]
```

All three lambdas close over the *same* `i`, and by the time any of them is called the loop has finished and `i` is `2`. The fixes bind the value at creation time:

```
adders.append(lambda x, i=i: x + i)              # default argument: evaluate
adders.append(functools.partial(lambda x, i: x + i, i=i))
adders = [multiplier_adder(i) for i in range(3)]  # a factory function: a fres
```

The default-argument trick is the shortest and the most common; the factory is the clearest. The same trap appears with callbacks registered in a loop and with generators created in a loop (Module 11).

The module scope and main

A script's top-level names are globals for every function in that file, which is exactly why work belongs in `main()`: names created inside `main` are locals, invisible to helpers, and a helper that accidentally uses one fails loudly instead of silently reading the wrong thing. Constants (`MAX_SIZE = 100`) at module level are the one kind of global a function may read without apology.

`globals()` and `locals()` return the current scope's names as a dict — useful for inspection, never for assignment. `del name` unbinds a local or global.

Pitfalls

- `x += 1` in a function that meant to update a module-level `x` — `UnboundLocalError`; declare `global` or, better, return the new value.
- A closure in a loop capturing the loop variable — late binding; bind with a default argument.
- Shadowing a built-in or an outer name and reading the wrong one later.
- `nonlocal` for a name at module level (a `SyntaxError`); that is `global`.
- Relying on a global for state that should be passed in or held by an object.
- Expecting a comprehension's loop variable to be visible afterwards (it is not; a `for` loop's is).

Key takeaways

- Names are resolved local → enclosing → global → built-in; first match wins.
- Assigning a name anywhere in a function makes it local for the whole function; reading before that assignment is `UnboundLocalError`.
- `global` binds the module name, `nonlocal` the enclosing function's; both are rare and `nonlocal` counters are the legitimate case.
- A closure captures variables, not values, and reads them when called.
- Functions built in a loop see the loop variable's final value; bind it with `i=i` or a factory.

EXERCISE 1

Counter factory

Write `make_counter(start, step)` that returns a function with no parameters; each call returns the current value and then advances it by `step` — the state lives in the closure and is updated with `nonlocal`. Each input line creates its **own** counter and prints its first `n` values, showing that counters are independent.

Input: lines `start step n`. **Output:** one line per input line with the `n` values, space-separated.

```
10 5 3
0 -1 4
```

prints

```
10 15 20
0 -1 -2 -3
```

INPUT 1

```
10 5 3
0 -1 4
```

OUTPUT 1

```
10 15 20
0 -1 -2 -3
```

REFERENCE SOLUTION

```
import sys


def make_counter(start, step):
    current = start

    def step_fn():
        nonlocal current
        value = current
        current += step
        return value
    return step_fn


for line in sys.stdin:
    start, step, n = map(int, line.split())
    counter = make_counter(start, step)
    print(" ".join(str(counter()) for _ in range(n)))
```

EXERCISE 2

Late binding, shown and fixed

Build two lists of `k` adder functions in a loop over `i`: a **late-bound** list of `lambda x: x + i`, and a **bound** list that captures each `i` with a default argument (`lambda x, i=i: x + i`). Apply every function in each list to `x` and print both rows, so the bug and the fix are visible side by side.

Input: one line `k x` with `k ⩾ 1`. **Output:** `bound: <results>` then `late: <results>`.

```
3 10
```

prints

```
bound: 10 11 12
late: 12 12 12
```

INPUT 1

```
3 10
```

OUTPUT 1

```
bound: 10 11 12
late: 12 12 12
```

REFERENCE SOLUTION

```
k, x = map(int, input().split())
bound = []
late = []
for i in range(k):
    bound.append(lambda x, i=i: x + i)
    late.append(lambda x: x + i)
print("bound:", " ".join(str(f(x)) for f in bound))
print("late:", " ".join(str(f(x)) for f in late))
```

QUIZ

1. What happens?

```
total = 0

def add(x):
    total += x

add(5)
```

A. `total` becomes 5

B. `UnboundLocalError`

C. `NameError`

D. `SyntaxError`

2. What does this print?

```
fs = [lambda: i for i in range(3)]
print([f() for f in fs])
```

A. `[0, 1, 2]`

B. `[2, 2, 2]`

C. `[3, 3, 3]`

D. `[None, None, None]`

3. In what order are names looked up inside a function?

A. Global, local, enclosing, built-in

B. Local, enclosing, global, built-in

C. Built-in, global, enclosing, local

D. Local, global, enclosing, built-in

4. Which keyword lets an inner function rebind a variable of the enclosing function?

A. `global`

B. `nonlocal`

C. `outer`

D. `static`

5. What does `make(2)(5)` return?

```
def make(n):
    def f(x):
        return x * n
    return f
```

A. `10`

B. `7`

C. A function

D. `NameError` because `n` is gone

ANSWER KEY · SCOPE AND CLOSURES — LEGB, GLOBAL, NONLOCAL AND LATE BINDING

1. B `total += x` assigns `total`, which makes it local to `add` everywhere in the function; reading it before the assignment fails. `global total` would fix it; returning the new value is better.

2. B Each lambda closes over the variable `i`, not its value at creation; when called after the comprehension finished, `i` is 2. `lambda i=i: i` would bind each value.

3. B LEGB: the innermost scope first, then enclosing function scopes, the module, and finally the built-ins. A local `list` therefore hides the built-in for that function.

4. B `nonlocal` targets the nearest enclosing function scope; `global` targets the module. The other two are not Python keywords.

5. A `make(2)` returns `f`, a closure that keeps `n` alive; calling it with 5 multiplies. The enclosing frame has returned but the captured variable lives in a cell.

LESSON 4 · 14 MIN

Recursion — base cases, the call stack and memoisation

A recursive function calls itself on a smaller version of its input and stops at a case small enough to answer directly. Python supports it like any language, with one limit that matters more here than elsewhere: the interpreter refuses to nest calls more than about 1 000 deep by default, and the stack it uses is real memory. This lesson gives the two-part shape of every recursive function, shows what the call stack does on each call, explains `RecursionError` and `sys.setrecursionlimit`, and turns exponential recursion into linear with memoisation — by hand and with `functools.cache`.

The shape

```
def factorial(n):
    if n ⩽ 1:                    # base case: answered directly
        return 1
    return n * factorial(n - 1)   # recursive case: a smaller problem, plus or
```

Two questions design any recursive function. *What is the smallest input, and what is its answer?* — the base case, written first, because a recursion without one never ends. *If I had the answer for a smaller input, how would I get the answer for this one?* — the recursive case, which must move toward the base case on every path.

```
def sum_digits(n):
    return n if n < 10 else n % 10 + sum_digits(n // 10)

def flatten(xs):                  # recursion over nested structure
    out = []
    for x in xs:
        if isinstance(x, list):
            out.extend(flatten(x))
        else:
            out.append(x)
    return out
```

Recursion over *structure* — nested lists, trees, JSON — is where it is clearly the right tool: the shape of the code follows the shape of the data. Recursion over *numbers* (`factorial`, `fib`) is usually a loop in disguise and is shown for teaching.

The call stack

Each call creates a frame holding its parameters and locals; the frame lives until the call returns. `factorial(4)` stacks four frames — `n=4`, `n=3`, `n=2`, `n=1` — then unwinds, multiplying on the way back:

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

Everything *after* the recursive call in the body runs during the unwind, in reverse order of the calls — which is why a function that prints before recursing prints top-down and one that prints after recursing prints bottom-up, and why reversing a list recursively is "the reverse of the rest, then the first element".

The recursion limit

CPython caps the depth at `sys.getrecursionlimit()` — 1 000 by default — and raises `RecursionError: maximum recursion depth exceeded` when a call would exceed it. A recursion with no base case hits it immediately; a correct recursion over a long list (`sum_list(xs)` recursing once per element) hits it at a thousand elements. `sys.setrecursionlimit(200_000)` raises the cap and works on this track's judge for depths in the tens of thousands, but the cap exists because each Python frame costs real memory and a deep enough recursion crashes the interpreter rather than raising. The safer answers, in order: make the recursion shallow (divide in half rather than peeling one element — depth log n), or rewrite it as a loop with an explicit stack (a list you `append` to and `pop` from), which is what any recursive traversal becomes when the input is large.

Python does **not** optimise tail calls: `return f(n - 1)` still adds a frame. A tail-recursive function is exactly the one that converts to a `while` loop mechanically, so convert it.

Memoisation

```
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)      # fib(35) makes ~30 m
```

The naive version recomputes `fib(k)` exponentially many times. Remembering each result the first time it is computed makes every call after the first O(1):

```
from functools import cache

@cache
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)      # fib(500) is instant
```

`@cache` (3.9; `@lru_cache(maxsize=None)` before it) wraps the function with a dictionary keyed by the arguments. The arguments must be hashable — ints, strings, tuples, not lists — and the function must be pure: same arguments, same result, no side effects. `fib.cache_info()` reports hits and misses; `fib.cache_clear()` empties it. The hand-written version is a dict parameter or a module-level dict checked at the top of the function:

```
def fib(n, memo={}):        # a mutable default used on purpose — the cache sl
    if n < 2:
        return n
    if n not in memo:
        memo[n] = fib(n - 1, memo) + fib(n - 2, memo)
    return memo[n]
```

Memoised recursion is *top-down dynamic programming*; the loop that fills a table from the base case up is *bottom-up*, uses no stack, and is what to reach for when the depth would be large.

Generating combinations

Recursion is the natural way to enumerate choices — every subset, every permutation, every path — because each level of the recursion makes one decision:

```python
def subsets(xs):
    if not xs:
        return [[]]
    rest = subsets(xs[1:])
    return rest + [[xs[0]] + s for s in rest]

print(subsets([1, 2, 3]))
# [[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```

`itertools.permutations`, `combinations` and `product` (Module 11) do the standard ones; write the recursion when there is a constraint to prune on — the backtracking pattern.

Pitfalls

No base case, or a recursive case that does not shrink the input on some path.

Recursing once per element over an input that can be long — `RecursionError` at ~1 000.

Building a new list on every call (`xs[1:]`) — quadratic; pass an index instead.

Memoising a function with side effects or unhashable arguments.

Expecting tail-call elimination.

Reading `RecursionError` as "recursion is too slow" — it is too *deep*.

Key takeaways

Base case first, then a recursive case that strictly shrinks the input.

Each call is a frame; code after the recursive call runs during the unwind, in reverse.

The default depth limit is 1 000; raise it with `sys.setrecursionlimit` for the judge, or recurse on halves, or use an explicit stack.

`@functools.cache` memoises a pure function with hashable arguments — exponential to linear.

Recursion fits nested data and choice enumeration; numeric recursion is usually a loop.

EXERCISE 1

Flatten and measure

Read one nested list literal (parse it with `ast.literal_eval`) and write two recursive functions: `flatten(xs)` returns every non-list element in order, and `depth(xs)` returns the nesting depth — `[]` and `[1, 2]` have depth 1, `[[1]]` has depth 2.

Input: one line holding a nested list of integers. **Output:** `flat: <elements space-separated>` (or `flat: (empty)`), then `depth: <d>`.

```
[1, [2, [3, 4]], 5]
```

prints

```
flat: 1 2 3 4 5
depth: 3
```

INPUT 1

```
[1, [2, [3, 4]], 5]
```

OUTPUT 1

```
flat: 1 2 3 4 5
depth: 3
```

REFERENCE SOLUTION

```python
import ast


def flatten(xs):
    out = []
    for x in xs:
        if isinstance(x, list):
            out.extend(flatten(x))
        else:
            out.append(x)
    return out


def depth(xs):
    inner = [depth(x) for x in xs if isinstance(x, list)]
    return 1 + max(inner, default=0)


data = ast.literal_eval(input())
flat = flatten(data)
print("flat:", " ".join(map(str, flat)) if flat else "(empty)")
print(f"depth: {depth(data)}")
```

EXERCISE 2

Fibonacci with a cache

Write the two-line recursive Fibonacci (`fib(0) = 0` , `fib(1) = 1`) and decorate it with `functools.cache` so that large arguments finish instantly. Read `n` queries and print each answer. Arguments go up to 500, so raise the recursion limit first.

Input: `n` , then `n` integers. **Output:** one Fibonacci number per line.

```
3
10
50
90
```

prints

```
55
12586269025
2880067194370816120
```

INPUT 1

```
3
10
50
90
```

OUTPUT 1

```
55
12586269025
2880067194370816120
```

REFERENCE SOLUTION

```python
import sys
from functools import cache

sys.setrecursionlimit(10_000)


@cache
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)


n = int(input())
for _ in range(n):
    print(fib(int(input())))
```

QUIZ

1. What is wrong with this function?

```
def count_down(n):
    print(n)
    count_down(n - 1)
```

A. Nothing

B. It has no base case, so it recurses until `RecursionError`

C. It should return `n`

D. `print` cannot be used inside recursion

2. What does this print?

```
def show(n):
    if n == 0:
        return
    show(n - 1)
    print(n, end=" ")

show(3)
```

A. `3 2 1`

B. `1 2 3`

C. `0 1 2 3`

D. Nothing

3. Which requirement does `@functools.cache` place on the decorated function?

A. It must return an int

B. Its arguments must be hashable and it should be pure (same inputs, same result)

C. It must be recursive

D. It must take exactly one argument

4. A correct recursion over a list of 5 000 elements, one element per call, raises `RecursionError`. What is the right fix?

A. Add a base case

B. Rewrite as a loop or an explicit stack, or recurse on halves; raising the limit is the last resort

C. Use `@cache`

D. Convert the list to a tuple

5. Does CPython eliminate tail calls (`return f(n - 1)` reusing the frame)?

A. Yes, always

B. No — every call adds a frame

C. Only with `@cache`

D. Only in functions without default arguments

ANSWER KEY · RECURSION — BASE CASES, THE CALL STACK AND MEMOISATION

1. B Nothing stops the recursion at 0; each call adds a frame until the interpreter's limit (1 000 by default) raises `RecursionError`.

2. B The print comes *after* the recursive call, so it runs during the unwind: the innermost call (n=1) prints first.

3. B The cache is a dict keyed by the arguments, so lists cannot be keys, and a cached result is only correct if the function has no side effects and depends only on its arguments.

4. B The recursion is correct but too deep for the default limit. `sys.setrecursionlimit` works on the judge but every frame costs memory; a loop, an explicit stack, or a log-depth divide-and-conquer removes the problem.

5. B Python deliberately keeps every frame for clear tracebacks. A tail-recursive function should be written as a `while` loop.

LESSON 5 · 13 MIN

Lambdas and higher-order functions

A higher-order function takes a function as an argument or returns one, and Python is full of them: `sorted(key=…)`, `max(key=…)`, `map`, `filter`, `functools.reduce`, `functools.partial`, and every callback and decorator you will meet. A `lambda` is the anonymous one-expression function you hand to them when a `def` would be ceremony. This lesson covers the `lambda` syntax and its single limitation, the key-function idiom that sorts and selects by any criterion, `map` / `filter` and when a comprehension beats them, `partial` for fixing arguments, `reduce` for folds, and the `operator` module that names every operator as a function.

---

lambda

```
square = lambda x: x * x          # equivalent to def square(x): return x * x
print(square(4))                  # 16
```

`lambda params: expression` builds a function object whose body is a single *expression* — no statements, no `return` (the expression's value is returned), no assignment, no loops, no `if` statement (the conditional *expression* is fine). It can take defaults, `*args` and `**kwargs` like any function. Its `__name__` is `'<lambda>'`, which is the reason not to assign one to a name as above: `def` gives the function a real name for tracebacks and `help`, at the cost of one more line. A lambda belongs *inline*, as an argument, where the reader sees it and its use together.

Key functions

Every ordering built-in takes `key=`, a one-argument function applied to each element to produce the value that is actually compared:

```
words = ["banana", "Apple", "cherry"]
sorted(words)                          # ['Apple', 'banana', 'cherry'] — upperc
sorted(words, key=str.lower)           # ['Apple', 'banana', 'cherry'] — case-i
sorted(words, key=len)                 # ['Apple', 'banana', 'cherry'] — by len
max(words, key=len)                    # 'banana'
sorted(pairs, key=lambda p: (-p[1], p[0]))    # by second value descending, th
```

The key is computed once per element (not on every comparison), so an expensive key is fine. Returning a tuple sorts by several criteria in order; negating a numeric component reverses just that criterion; `reverse=True` reverses the whole order. Any callable works as a key — a built-in (`len`, `str.lower`, `abs`), a lambda, a `def`, or the `operator` helpers below.

---

map and filter

`map(f, iterable)` applies `f` to each element lazily; `filter(pred, iterable)` keeps the elements for which `pred` is true. Both return iterators, so wrap in `list()` to see them:

```
list(map(int, ["1", "2", "3"]))                  # [1, 2, 3]
list(filter(str.isdigit, ["1", "a", "22"]))      # ['1', '22']
list(map(lambda x, y: x + y, [1, 2], [10, 20]))  # [11, 22] — map takes severa
```

`map(int, tokens)` and `filter(None, xs)` (drops falsy values) are the idiomatic uses: an existing function applied to everything. The moment you write `map(lambda x: …)` or `filter(lambda x: …)`, a comprehension says the same thing more readably — `[x * 2 for x in xs]`, `[x for x in xs if x > 0]` — and can combine both (Module 6). Choose `map` when the function already exists; choose the comprehension when you would have to write a lambda.

---

functools.partial

`partial(f, *args, **kwargs)` returns a new function with some of `f`'s arguments fixed:

```
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
cube = partial(power, exponent=3)
print(square(5), cube(2))          # 25 8

int_from_binary = partial(int, base=2)
list(map(int_from_binary, ["101", "11"]))   # [5, 3]
```

It replaces `lambda x: power(x, 2)`, and unlike the lambda it carries its function and fixed arguments as inspectable attributes (`square.func`, `square.keywords`). It is also the cleanest fix for late binding in loops (Module 4 lesson 3), because the argument is bound at the moment `partial` is called.

---

functools.reduce

`reduce(f, iterable, initial)` folds a sequence to one value by applying `f(accumulator, element)` left to right:

```
from functools import reduce
import operator

reduce(operator.mul, [1, 2, 3, 4], 1)       # 24 — the product
reduce(lambda a, b: a if len(a) ⩾ len(b) else b, words)   # the longest word
```

Python deliberately left `reduce` out of the built-ins: `sum`, `min`, `max`, `any`, `all`, `math.prod` and `"".join` cover the common folds by name, and a `for` loop with an accumulator is clearer than a `reduce` with a lambda. Reach for it when the combining function *is* a named function and the fold has no built-in.

The operator module

`operator.add`, `mul`, `neg`, `lt`, `eq`, `contains`, `getitem` and the rest are the operators as functions — the arguments `reduce`, `map` and `sorted` want when a lambda would only wrap an operator. Three factories are used constantly with `key=`:

```
from operator import itemgetter, attrgetter, methodcaller

sorted(rows, key=itemgetter(2))          # by column 2
sorted(rows, key=itemgetter(1, 0))       # by column 1, then 0 — returns a tup
sorted(people, key=attrgetter("age"))    # by an attribute
sorted(words, key=methodcaller("lower")) # by calling a method
```

`itemgetter(1)` is `lambda x: x[1]` with a name and a small speed advantage.

Functions returning functions

The factory pattern — a function that builds and returns a specialised function — is the closure of the previous lesson put to work:

```
def make_validator(lo, hi):
    def valid(x):
        return lo ⩽ x ⩽ hi
    return valid

in_percent = make_validator(0, 100)
print(in_percent(50), in_percent(150))    # True False
```

Decorators (Module 16) are exactly this with the function itself as the input.

Pitfalls

- A lambda that needs a statement — it cannot; write a `def`.
- Assigning a lambda to a name instead of using `def`.
- Forgetting `list()` around `map` / `filter` and printing `<map object at …>`; or iterating one twice (it is exhausted after the first pass).
- `key=len()` — calling instead of passing; `key=len`.
- `reduce` with a lambda where `sum` or a loop is clearer.
- `sorted(xs, key=lambda x: -x)` on strings — negation needs numbers; use `reverse=True`.

Key takeaways

- `lambda params: expression` is an inline single-expression function; use `def` for anything named or multi-line.
- `key=` on `sorted` / `min` / `max` selects the comparison value; tuples sort by several criteria, negation reverses one.
- `map` / `filter` apply existing functions lazily; prefer a comprehension when you would write a lambda.

`partial` fixes arguments; `reduce` folds with a named function; `operator` names the operators and `itemgetter` / `attrgetter` make keys.

A function returning a function is a closure factory — the shape decorators build on.

EXERCISE 1

Sort by several keys

Read `n` records `name score time` and sort them by score **descending**, then by time **ascending**, then by name ascending, using a single `sorted` call with a key function returning a tuple. Then print the best record by the same ordering using `min` with the same key.

Input: `n`, then `n` lines. **Output:** the sorted records as `name score time`, then `best: <name>`.

```
3
ada 90 12
bob 90 10
cy 80 5
```

prints

```
bob 90 10
ada 90 12
cy 80 5
best: bob
```

INPUT 1

```
3
ada 90 12
bob 90 10
cy 80 5
```

OUTPUT 1

```
bob 90 10
ada 90 12
cy 80 5
best: bob
```

REFERENCE SOLUTION

```python
n = int(input())
records = []
for _ in range(n):
    name, score, time = input().split()
    records.append((name, int(score), int(time)))

key = lambda r: (-r[1], r[2], r[0])
ordered = sorted(records, key=key)
for name, score, time in ordered:
    print(name, score, time)
best = min(records, key=key)
print(f"best: {best[0]}")
```

EXERCISE 2

A pipeline from partial

The first line lists operations as `name value` pairs: `add k`, `sub k` (subtract `k`), `mul k`, `pow k`. Build each step as a `functools.partial` of the matching two-argument helper with `k` fixed, then apply the steps in order to every integer on the second line using `map` and print the results.

Input: a line of operation pairs (possibly empty), then a line of integers. **Output:** the results, space-separated.

```
add 3 mul 2
1 2 3
```

prints

```
8 10 12
```

INPUT 1

```
add 3 mul 2
1 2 3
```

OUTPUT 1

```
8 10 12
```

REFERENCE SOLUTION

```python
from functools import partial


def add(k, x):
    return x + k


def sub(k, x):
    return x - k


def mul(k, x):
    return x * k


def power(k, x):
    return x ** k


HELPERS = {"add": add, "sub": sub, "mul": mul, "pow": power}

tokens = input().split()
steps = [partial(HELPERS[tokens[i]], int(tokens[i + 1])) for i in range(0, len(toker
nums = list(map(int, input().split()))


def run(x):
    for step in steps:
        x = step(x)
    return x


print(*map(run, nums))
```

QUIZ

1. What does `sorted(["bb", "a", "ccc"], key=len, reverse=True)` return?

A. `['ccc', 'bb', 'a']`

B. `['a', 'bb', 'ccc']`

C. `['ccc', 'a', 'bb']`

D. `[3, 2, 1]`

2. Which is a valid lambda?

A. `lambda x: return x + 1`

B. `lambda x: x + 1`

C. `lambda x: if x: 1`

D. `lambda: x = 1`

3. What does `list(map(str.upper, ["a", "b"]))` produce?

A. `['A', 'B']`

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

C. `TypeError`

D. `<map object>`

4. What does `partial(pow, 2)(10)` compute?

A. `2 ** 10` = 1024

B. `10 ** 2` = 100

C. `pow(10, 2, 2)`

D. `TypeError`

5. Which key sorts a list of `(name, age)` pairs by age ascending and then name descending?

A. `key=lambda p: (p[1], p[0])` with `reverse=True`

B. `key=lambda p: (p[1], -p[0])`

C. Sort by name descending first, then a stable sort by age ascending

D. `key=itemgetter(1, 0)`

6. Why is `reduce` not a built-in?

A. It was removed for being slow

B. Because `sum`, `min`, `max`, `any`, `all` and a plain loop cover the common folds more readably

C. It cannot take a lambda

D. It is a built-in — `functools` just re-exports it

ANSWER KEY · LAMBDAS AND HIGHER-ORDER FUNCTIONS

1. A The key maps each string to its length, the sort orders by that value, and `reverse=True` flips it. The elements themselves are returned, not the keys.

2. B A lambda body is one expression; `return`, `if` statements and assignments are statements and not allowed. The conditional *expression* `1 if x else 0` would be fine.

3. A `str.upper` is an unbound method taking the string as its argument, so `map` applies it to each element; `list` materialises the lazy result.

4. A `partial` fixes the *first* positional argument, so the call becomes `pow(2, 10)`. To fix the exponent use `partial(pow, exp=…)` — which `pow` does not accept — or a helper with the parameters in the other order.

5. C Strings cannot be negated, so a mixed-direction tuple key is impossible in one expression; two stable sorts, least significant first, achieve it. Options 1 and 4 give both keys the same direction.

6. B It lives in `functools` by design; a `for` loop with an accumulator is usually clearer than `reduce` with a lambda, and the named folds exist for the common cases.

LESSON 6 · 12 MIN

Type hints and docstrings — the contract a function publishes

A function's signature says what it takes; its docstring says what it does; its type hints say what kinds of values go in and come out. None of the three is enforced by the interpreter — Python will happily call `area("3", 4)` and fail inside — but together they are the contract that editors complete against, that `mypy` checks, that `help()` displays, and that the next reader relies on. This lesson introduces annotations and the built-in generic syntax, states what hints do and do not do at run time, and fixes the docstring conventions; Module 15 goes deep on the type system.

Annotations

```
def area(width: float, height: float) -> float:
    return width * height

def greet(name: str, times: int = 1) -> str:
    return f"Hello, {name}! " * times
```

`name: type` annotates a parameter; `-> type` annotates the return. The annotations are stored in `area.__annotations__` (`{'width': float, 'height': float, 'return': float}`) and otherwise ignored: no conversion, no check, no error. `area("3", 4)` raises `TypeError` from the multiplication, not from the hint. Variables can be annotated too — `total: int = 0`, or `count: int` with no value, which declares without binding.

The hint vocabulary

Since 3.9 the built-in collections are their own generics, and since 3.10 `|` writes a union:

Hint	Meaning
`int`, `str`, `float`, `bool`, `bytes`	that type (an `int` hint also accepts `bool`; a `float` hint accepts `int`)
`list[int]`	a list whose elements are ints
`dict[str, int]`	a dict from str keys to int values
`tuple[int, str]`	a two-tuple of exactly those types
`tuple[int, ...]`	a tuple of any length of ints
`set[str]`, `frozenset[str]`	sets
`int | None`	an int or `None` — the optional value (older code: `Optional[int]`)
`int | str`	either (older: `Union[int, str]`)
`Any` (from `typing`)	anything; switches checking off for that value
`Callable[[int, str], bool]` (from `collections.abc`)	a function taking an int and a str, returning a bool
`Iterable[int]`, `Sequence[int]`, `Mapping[str, int]` (from `collections.abc`)	anything iterable / indexable / dict-like — the right hints for *parameters*, because they accept more than `list`
`None`	as a return: the function returns nothing

The rule for parameters is to accept the *widest* type that works (`Iterable[int]` rather than `list[int]` if you only loop over it) and to return the *narrowest* concrete type you actually produce (`list[int]`).

```python
from collections.abc import Iterable, Callable

def total(xs: Iterable[int]) -> int:
    return sum(xs)

def apply_all(fs: list[Callable[[int], int]], x: int) -> list[int]:
    return [f(x) for f in fs]

def find(xs: list[str], target: str) -> int | None:
    return xs.index(target) if target in xs else None
```

`int | None` is the honest return type of a lookup that can fail, and the checker will insist the caller test for `None` before using the result — which is the bug that hint exists to catch.

What hints buy you

- **Editor completion**: with `s: str`, typing `s.` lists string methods.
- **Static checking**: `mypy program.py` or `pyright` reports `area("3", 4)` as an error before the program runs, and follows `None` through every path.
- **Documentation**: the signature says `-> int | None` where a docstring would have to spell it out.
- **Run-time introspection**: `dataclasses` builds fields from annotations; `typing.get_type_hints` resolves them; libraries such as pydantic and FastAPI validate real data against them.

What they cost is a few characters per parameter and the discipline to keep them true; a wrong hint is worse than none. A function in a judged exercise is fine with or without them; a function in a codebase is expected to have them at its public boundary.

Docstrings

```python
def moving_average(xs: list[float], window: int) -> list[float]:
    """Return the averages of each consecutive `window` values of `xs`.

    The result has `len(xs) - window + 1` entries. Raises ValueError if
    `window` is less than 1 or greater than `len(xs)`.
    """
```

PEP 257's rules: a one-line summary in the imperative mood ("Return…", "Compute…", "Raise…"), ending in a full stop, fitting on the first line; a blank line; then details — the meaning of parameters where the names do not say it, the shape of the result, the exceptions raised, any side effects. Triple double quotes, closing quotes on their own line for multi-line docstrings. The summary should not repeat the signature ("takes a list and an int") — the reader can see that — but say what the function *means*. `help(moving_average)` prints the signature and the docstring; `moving_average.__doc__` is the text.

Google-style and NumPy-style docstrings add structured `Args:` / `Returns:` / `Raises:` sections that documentation tools render; pick one style per project. For a private helper, one line is enough; for a function others will call, write the contract.

Reading a signature with everything

```
def search(
    items: Sequence[str],
    query: str,
    *,
    limit: int | None = None,
    key: Callable[[str], str] = str.lower,
) → list[str]:
    """Return the items matching `query`, at most `limit` of them."""
```

Positional inputs, keyword-only options after `*`, hints on each, a default that is a function, a return hint, a docstring. That is what a function in a well-kept codebase looks like, and every piece of it has been introduced in this module.

Pitfalls

Believing a hint converts or checks at run time. It does neither.

`list` where `Iterable` would accept a tuple, a set or a generator too.

Forgetting `| None` on a return that can be `None`; the checker cannot then catch the missing test.

`Optional[int]` meaning "the parameter is optional" — it means "may be `None`"; an optional parameter is one with a default.

A docstring that restates the code or the signature.

Mutable default values hinted as `list[int] = []` — the hint does not fix the shared default.

Key takeaways

`param: type` and `→ type` record the contract; the interpreter stores and ignores them.

`list[int]`, `dict[str, int]`, `tuple[int, ...]`, `X | None`, `Callable[[…], R]`, and `Iterable` / `Sequence` / `Mapping` for parameters.

Hints pay in editor completion, static checking and documentation; keep them true.

A docstring is an imperative one-line summary, a blank line, then the details a caller needs.

Accept the widest parameter type that works, return the narrowest concrete one.

EXERCISE 1

Inspect the contract

Write `mean(xs: list[float]) → float` with a one-line docstring, then print its result for the input **and** a report read from the function object itself: its `__name__`, the first line of its `__doc__`, and its annotations formatted from `__annotations__` (a class prints as its `__name__`; a generic like `list[float]` prints as its `str`).

Input: one line of numbers (at least one). **Output:** `mean <value>` with two decimals, then `name: mean`, `doc: <first docstring line>`, `param xs: list[float]`, `returns: float`.

```
1 2 4
```

prints

```
mean 2.33
name: mean
doc: Return the arithmetic mean of xs.
param xs: list[float]
returns: float
```

INPUT 1

```
1 2 4
```

OUTPUT 1

```
mean 2.33
name: mean
doc: Return the arithmetic mean of xs.
param xs: list[float]
returns: float
```

REFERENCE SOLUTION

```python
def mean(xs: list[float]) -> float:
    """Return the arithmetic mean of xs."""
    return sum(xs) / len(xs)


def hint_name(h):
    return h.__name__ if isinstance(h, type) else str(h)


values = [float(t) for t in input().split()]
print(f"mean {mean(values):.2f}")
print(f"name: {mean.__name__}")
print(f"doc: {mean.__doc__.strip().splitlines()[0]}")
hints = dict(mean.__annotations__)
ret = hints.pop("return")
for param, hint in hints.items():
    print(f"param {param}: {hint_name(hint)}")
print(f"returns: {hint_name(ret)}")
```

EXERCISE 2

A checker driven by annotations

Hints do nothing at run time — unless you make them. Three hinted functions are given: `add(a: int, b: int)`, `repeat(s: str, n: int)` and `scale(x: float, k: float)`. Write `call(f, *args)` that reads `f.__annotations__` and checks each argument with `isinstance` against its hint before calling (`int` is acceptable where `float` is hinted, as PEP 484 allows). Each input line is a function name followed by argument literals (parse them with `ast.literal_eval`).

Input: `n`, then `n` lines. **Output:** `ok <result>` or `type error: <param> expects <type>` (the first failing parameter).

```
3
add 1 2
repeat "ab" 3
add 1 "x"
```

prints

```
ok 3
ok abababab
type error: b expects int
```

INPUT 1

```
3
add 1 2
repeat "ab" 3
add 1 "x"
```

OUTPUT 1

```
ok 3
ok abababab
type error: b expects int
```

REFERENCE SOLUTION

```
import ast


def add(a: int, b: int) -> int:
    return a + b


def repeat(s: str, n: int) -> str:
    return s * n


def scale(x: float, k: float) -> float:
    return x * k


FUNCTIONS = {"add": add, "repeat": repeat, "scale": scale}


def accepts(value, hint):
    if hint is float:
        return isinstance(value, (int, float)) and not isinstance(value, bool)
    return isinstance(value, hint)


def call(f, *args):
    params = [(p, h) for p, h in f.__annotations__.items() if p != "return"]
    for (param, hint), value in zip(params, args):
        if not accepts(value, hint):
            return f"type error: {param} expects {hint.__name__}"
    return f"ok {f(*args)}"


n = int(input())
for _ in range(n):
    name, *literals = input().split()
    args = [ast.literal_eval(tok) for tok in literals]
    print(call(FUNCTIONS[name], *args))
```

QUIZ

1. What happens when `def f(x: int) -> int: return x` is called as `f("a")`?

A. `TypeError` from the hint

B. Returns `"a"` — hints are not enforced at run time

C. `ValueError`

D. A warning is printed and `"a"` is returned

2. Which hint is right for a parameter the function only iterates over?

A. `list[int]`

B. `Iterable[int]` from `collections.abc`

C. `Any`

D. `tuple[int]`

3. What does the hint `int | None` on a return type tell a caller?

A. The function returns an int, or raises

B. The result may be `None` and must be checked before use

C. The parameter is optional

D. The function returns `None` on error and prints a message

4. Where does `help(f)` get its text?

A. From the comments above the `def`

B. From `f.__doc__`, the first-statement string of the body

C. From `f.__annotations__`

D. From the source file's header

5. Which summary line follows PEP 257?

A. `"""returns the largest value"""`

B. `"""Return the largest value in xs."""`

C. `"""This function will return the largest value."""`

D. `"""largest(xs) → int"""`

ANSWER KEY · TYPE HINTS AND DOCSTRINGS — THE CONTRACT A FUNCTION PUBLISHES

1. B Annotations are stored and otherwise ignored by the interpreter. A checker such as mypy would flag the call; the running program does nothing special.

2. B Accept the widest type that works: an `Iterable` hint lets callers pass a list, tuple, set, generator or range. `list[int]` needlessly rejects the others; `Any` switches checking off.

3. B A union with `None` documents a lookup that can fail, and a type checker will insist on a `None` test before the value is used as an int. Optional *parameters* are those with defaults — a different thing.

4. B The docstring is the string literal that is the first statement of the function body; comments are discarded by the compiler. `help` shows the signature (including hints) and the docstring.

5. B Imperative mood, capitalised, ending in a full stop, describing the contract rather than restating the signature.

CHECKPOINT · 25 MIN

Checkpoint — Functions

This checkpoint covers the whole module: `def` and `return` and the `None` that comes back when nothing is returned, the signature — positional, keyword, defaults, `*args`, `**kwargs`, `*` and `/` — and the mutable-default trap, LEGB scope with `global`, `nonlocal`, closures and late binding, recursion with its depth limit and memoisation, lambdas and the key-function idiom with `map`, `filter`, `partial`, `reduce` and `operator`, and type hints and docstrings.

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

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

- What does a function return when it has no `return`, and which list methods return that on purpose?
- Why is `def f(xs=[])` a bug, and what is the idiom?
- What is the difference between `f(xs)` and `f(*xs)`?
- Why does `count += 1` inside a function raise `UnboundLocalError`, and what are the two ways out?
- What do three lambdas created in a `for` loop over `i` see when called later, and how do you fix it?
- What does `@functools.cache` require of the function it wraps?
- In `sorted(pairs, key=lambda p: (-p[1], p[0]))`, what is the order?

The three programs are a memoised path counter over a grid with obstacles, a small argument-parsing function that accepts positional and keyword-style arguments and reports what it received, and a pipeline that composes a list of functions built by factories.

EXERCISE 1

Grid paths around obstacles

Count the paths from the top-left cell to the bottom-right cell of a grid, moving only right or down and never entering a `#` cell. Write a recursive `paths(r, c)` memoised with `functools.cache`: the answer for a cell is the sum of the answers for the cell to its right and the cell below it, with the bottom-right cell counting 1 and any blocked or out-of-range cell counting 0.

Input: `R C`, then `R` lines of `.` and `#`. **Output:** the number of paths.

```
3 3
...
.#.
...
```

prints

```
2
```

INPUT 1

```
3 3
...
.#.
...
```

OUTPUT 1

```
2
```

INPUT 2

```
2 2
..
..
```

OUTPUT 2

```
2
```

REFERENCE SOLUTION

```python
from functools import cache

R, C = map(int, input().split())
grid = [input() for _ in range(R)]


@cache
def paths(r, c):
    if r >= R or c >= C or grid[r][c] == "#":
        return 0
    if r == R - 1 and c == C - 1:
        return 1
    return paths(r + 1, c) + paths(r, c + 1)


print(paths(0, 0))
```

EXERCISE 2

A keyword-only signature under test

Define `connect(host, port=80, *, secure=False)` returning `host=<host> port=<port> secure=<secure>`. Then read `n` call expressions written as Python source, evaluate each with `eval` in a `try`, and print the result — or `TypeError` when the call does not fit the signature (too many positional arguments, a missing `host`, an unknown keyword).

Input: `n`, then `n` call expressions. **Output:** one line per call.

```
3
connect("db")
connect("db", 443, secure=True)
connect("db", 1, True)
```

prints

```
host=db port=80 secure=False
host=db port=443 secure=True
TypeError
```

INPUT 1

```
3
connect("db")
connect("db", 443, secure=True)
connect("db", 1, True)
```

OUTPUT 1

```
host=db port=80 secure=False
host=db port=443 secure=True
TypeError
```

REFERENCE SOLUTION

```
def connect(host, port=80, *, secure=False):
    return f"host={host} port={port} secure={secure}"


n = int(input())
for _ in range(n):
    expr = input()
    try:
        print(eval(expr))
    except TypeError:
        print("TypeError")
```

EXERCISE 3

Compose

Build a pipeline of closures. The first line lists steps: `add k`, `mul k` and `clamp lo hi`; write a factory for each (`make_add(k)`, `make_mul(k)`, `make_clamp(lo, hi)`) that returns a one-argument function.

Compose all the steps into a **single** function with `functools.reduce` and a `compose(f, g)` helper, then apply it to every integer on the second line.

Input: a line of steps (possibly empty), then a line of integers. **Output:** the results, space-separated.

```
add 1 mul 3 clamp 0 10
-5 0 2 5
```

prints

```
0 3 9 10
```

INPUT 1

```
add 1 mul 3 clamp 0 10
-5 0 2 5
```

OUTPUT 1

```
0 3 9 10
```

REFERENCE SOLUTION

```
from functools import reduce


def make_add(k):
    return lambda x: x + k


def make_mul(k):
    return lambda x: x * k


def make_clamp(lo, hi):
    return lambda x: max(lo, min(hi, x))


def compose(f, g):
    return lambda x: g(f(x))


tokens = input().split()
steps = []
i = 0
while i < len(tokens):
    if tokens[i] == "clamp":
        steps.append(make_clamp(int(tokens[i + 1]), int(tokens[i + 2])))
        i += 3
    else:
        factory = {"add": make_add, "mul": make_mul}[tokens[i]]
        steps.append(factory(int(tokens[i + 1])))
        i += 2

pipeline = reduce(compose, steps, lambda x: x)
print(*(pipeline(int(t)) for t in input().split()))
```

CHECKPOINT QUESTIONS

1. What does this print?

```
def f():
    return

print(f() is None)
```

A. `True`

B. `False`

C. `None`

D. `SyntaxError`

2. Which default is safe?

A. `def f(xs=[])`

B. `def f(xs={})`

C. `def f(xs=None)` then `xs = [] if xs is None else xs`

D. `def f(xs=list())`

3. What does `print(*[1, 2], *[3])` print?

A. `[1, 2] [3]`

B. `1 2 3`

C. `(1, 2, 3)`

D. `TypeError`

4. What does this print?

```
x = 1
def f():
    print(x)
    x = 2
f()
```

A. `1`

B. `2`

C. `UnboundLocalError`

D. `None`

5. Which fix makes the lambdas capture each `i`?

```
fs = [lambda: i for i in range(3)]
```

A. `[lambda: i for i in range(3)][::-1]`

B. `[lambda i=i: i for i in range(3)]`

C. `[lambda: int(i) for i in range(3)]`

D. `[lambda: i.copy() for i in range(3)]`

6. What does `@functools.cache` change about `fib(n)`?

A. It makes the recursion iterative

B. It stores each result so repeated arguments return instantly — exponential becomes linear

C. It raises the recursion limit

D. It converts the result to a float

7. What is `sorted([(1, "b"), (1, "a"), (0, "z")], key=lambda p: (-p[0], p[1]))`?

A. `[(1, 'a'), (1, 'b'), (0, 'z')]`

B. `[(0, 'z'), (1, 'a'), (1, 'b')]`

C. `[(1, 'b'), (1, 'a'), (0, 'z')]`

D. `TypeError`

8. What does `list(filter(None, [0, 1, "", "a", None, []]))` produce?

A. `[1, 'a']`

B. `[0, '', None, []]`

C. `[]`

D. `TypeError`

9. What is `f.__annotations__` for `def f(a: int, b) → str`?

A. `{'a': int, 'b': None, 'return': str}`

B. `{'a': int, 'return': str}`

C. `{'a': 'int', 'return': 'str'}`

D. `{'return': str}`

10. Which statement about `nonlocal` is correct?

A. It refers to a module-level name

B. It lets an inner function rebind a name in an enclosing *function* scope, which must already exist

C. It creates the variable if it does not exist

D. It is required to *read* an enclosing variable

11. What does `make_counter` return, and what does calling the result twice give?

```
def make_counter():
    n = 0
    def step():
        nonlocal n
        n += 1
        return n
    return step
```

A. It returns 1; calling it is a `TypeError`

B. A function; calling it twice gives 1 then 2

C. A function; calling it twice gives 1 then 1

D. `None`

12. A function's docstring is the string on its *second* line, after a comment on the first. Is it stored in `__doc__`?

A. No — the comment breaks it

B. Yes — comments are not statements, so the string is still the first statement

C. Only if it uses triple quotes

D. Only in Python 3.12+

ANSWER KEY · CHECKPOINT — FUNCTIONS

1. A A bare `return` returns `None`, the same as falling off the end. Comparing with `is None` gives `True`.

2. C Any expression in the default is evaluated once at `def` time — `list()` no less than `[]`. The `None` sentinel with creation inside the body gives each call a fresh object.

3. B Each star spreads its list into separate positional arguments, so `print` receives 1, 2, 3.

4. C The assignment on the second line makes `x` local to the whole function, so the `print` on the first line reads a local that has no value yet.

5. B A default argument is evaluated when the lambda is created, binding the current value. Reversing, converting or copying inside the body still reads `i` late.

6. B The decorator wraps the function with a dictionary keyed by arguments; the call tree collapses because every sub-problem is solved once.

7. A The negated first component sorts descending; ties are broken by the string ascending. Tuples compare element by element.

8. A `filter(None, ...)` keeps the truthy elements. Zero, the empty string, `None` and the empty list are falsy and dropped.

9. B Only annotated parameters appear; `b` has no entry. The values are the objects themselves (the classes), not strings, unless `from __future__ import annotations` is in effect.

10. B Reading needs nothing; rebinding needs `nonlocal`, and the name must be bound in an enclosing function (not the module — that is `global`).

11. B The closure keeps `n` in a cell and `nonlocal` lets `step` update it, so each call continues from the previous value.

12. B The compiler discards comments entirely; the string literal remains the first statement of the body and becomes the docstring. Any quote style works, though triple double quotes are the convention.