Module 19 of 20

Memory, performance and the interpreter

The object model with reference counting and the cycle collector; the cost model of the built-in operations; bytecode, code objects and the 3.11 specialising interpreter; measuring with timeit, cProfile and tracemalloc; numeric performance with array, bytes and NumPy; and the checklist for writing fast Python.

Python is fast enough for most programs and slow in ways that are entirely predictable once you know three things: every value is a heap object with a reference count, every built-in operation has a fixed cost, and the interpreter executes one bytecode instruction at a time with no compiler to save you from a quadratic loop. This module makes those three things visible — the object header and the collector, the table of costs, the bytecode behind a line — and then turns them into practice: measuring before optimising, moving numeric work into unboxed arrays, and a checklist that starts with the algorithm and ends with the hot loop.

The object model covers what objects cost, references versus values, reference counting, cycles and the generational collector, weak references and the memory levers. The cost model gives the table of list, dict, set, string and deque costs, the loop shapes that hide a quadratic, and the hidden costs of copying, hashing and calls. Bytecode and the interpreter covers code objects, reading `dis`, the four kinds of load, frames, constant folding and the 3.11 specialiser. Measuring covers `perf_counter`, `timeit`, `cProfile`, `tracemalloc` and benchmarking hygiene. Numeric performance covers `array`, `bytes`, `memoryview` and NumPy vectorisation. Writing fast Python is the ordered checklist.

The exercises are whole programs: a reference-count simulator and a cycle-collector simulator, a sliding-window maximum and a membership test sized so that only the right data structure finishes in time, a code-object inspector and a name classifier, a profile-table reader and a stopwatch with an injected clock, vectorised statistics and byte checksums, memoised grid paths and prefix-sum range queries. The checkpoint adds top-k with a heap, order-preserving de-duplication at scale, and a frequency tally with stable ties.

CONTENTS

1 The object model — objects, references, reference counting and the cycle collector 15 min

2	The cost model — what the built-in operations really cost	15 min
3	Bytecode and the interpreter — code objects, dis, name lookup and the 3.11 specialiser	15 min
4	Measuring — timeit, perf_counter, cProfile, tracemalloc and benchmarking hygiene	14 min
5	Numeric performance — boxed numbers, array, bytes and NumPy vectorisation	14 min
6	Writing fast Python — the checklist, from algorithm to micro-optimisation	14 min
✓	Checkpoint — Memory, performance and the interpreter	25 min

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

LESSON 1 · 15 MIN

The object model — objects, references, reference counting and the cycle collector

Every value in Python is an object on the heap with a header — a reference count and a pointer to its type — and every variable, container slot and attribute is a *reference* to one. That single fact explains the memory cost of an integer (28 bytes, not 4), why `a = b` copies nothing, why `del` rarely frees anything, when an object actually disappears, and why two objects that point at each other need a separate collector. This lesson covers the object header and what things cost, references and the `is` / `==` distinction, reference counting and its immediate frees, the generational cycle collector and the `gc` module, weak references, and the levers — `__slots__`, sharing, generators — that reduce memory.

What an object costs

```
import sys
sys.getsizeof(0)            # 28  — header + one 30-bit digit
sys.getsizeof(2**64)        # 36  — more digits
sys.getsizeof("")           # 49; each ASCII character adds 1
sys.getsizeof([])           # 56; each slot adds 8 (a pointer), over-allocated
sys.getsizeof((1, 2))       # 56
sys.getsizeof({})           # 64; a compact table that grows in steps
sys.getsizeof(1.5)          # 24
```

The numbers are for the object itself, not what it references: a list of a million ints is 8 MB of pointers plus 28 MB of int objects. (Never print these in a judged program — they vary by build; the lesson quotes CPython 3.11 on a 64-bit Linux.) Small ints from −5 to 256 are pre-allocated and shared; short identifier-like strings are *interned* and shared; everything else is a fresh allocation. Containers hold pointers, so a tuple of three ints is 56 bytes plus three ints that may already exist.

References, not values

```
a = [1, 2, 3]
b = a                    # a second reference to the same list — no copy
b.append(4)
a                        # [1, 2, 3, 4]
a is b                   # True: same object
c = a[:]                 # a shallow copy: new list, same element objects
c is a, c == a           # (False, True)
```

Assignment binds a name to an object; passing an argument binds a parameter; storing in a container stores the pointer. `is` compares identity (the same object), `==` compares value (the type's `__eq__`); the idiom `x is None` is right because there is one `None`, and `x is 5` is wrong because it depends on the small-int cache. `copy.copy` is the shallow copy, `copy.deepcopy` copies recursively, and neither is needed for immutable objects, which can be shared freely because nobody can change them.

Reference counting

Each object's header holds the number of references to it. Binding a name or storing in a container increments it; rebinding, `del`, a container being freed or a function returning (its locals go away) decrements it; at zero the object is freed *immediately* and its `__del__`, if any, runs. This is why a file closed in a `with` block or a large list dropped at the end of a function is released at a predictable moment, and why `del x` frees nothing when another reference exists — it removes one name, not the object. `sys.getrefcount(x)` reports the count (one higher than expected, because the argument is a reference too).

The cost of reference counting is that every operation on an object touches its header, which is one reason a thread-safe interpreter needs the GIL (Module 17), and one reason CPython is slower than a tracing-GC language at pointer-heavy code.

Cycles and the collector

```
a = []
b = [a]
a.append(b)              # a → b → a
del a, b                 # both counts are still 1: each holds the other
```

Reference counting cannot free a cycle — each object's count never reaches zero. CPython's *cycle collector* finds them: it tracks container objects in three generations, periodically walks the youngest generation, subtracts references that come from within the tracked set, and frees what is not reachable from outside. Young objects are examined often, survivors are promoted, and old generations are examined rarely, on the observation that most objects die young. `gc.collect()` runs a full pass and returns the count freed; `gc.disable()` is a real optimisation for a short allocation-heavy phase that creates no cycles; `gc.get_count()` and `gc.get_threshold()` expose the state. Objects that cannot participate in cycles — ints, strings, tuples of atoms — are not tracked at all.

Weak references

`weakref.ref(obj)` refers to an object without counting; the referent is freed when the strong references go, and the weak ref then returns `None`. `weakref.WeakValueDictionary` and `WeakKeyDictionary` are caches and registries that do not keep their entries alive — the standard answer to "a cache that keeps everything forever". Most built-in types cannot be weakly referenced (lists, dicts, ints); instances of your own classes can, unless `__slots__` omits `__weakref__`.

Reducing memory

- `__slots__ = ("x", "y")` on a class replaces the per-instance `__dict__` with fixed slots: roughly 48 bytes per instance instead of ~150, and faster attribute access; the price is no ad-hoc attributes.
- A generator or `itertools` pipeline processes items one at a time instead of materialising a list; `sum(x * x for x in xs)` allocates nothing.
- `array.array("d")`, `bytes`, `bytearray` and NumPy arrays store raw numbers, 8 bytes each, instead of 8-byte pointers to 24-byte floats.
- Share immutable data; `sys.intern` for many repeated strings; tuples instead of lists for fixed records; `dict` keys drawn from a small set are shared through key-sharing dicts on instances.
- Delete or scope out references to large intermediates as soon as they are done; the object is freed when the last reference goes, not when the function ends.

Pitfalls

Believing `del x` frees the object.

`x is 5` or `s is "abc"` — works by accident of caching, fails elsewhere.

A cache dict that holds every object ever created.

`__del__` methods that assume the order of finalisation at interpreter shutdown.

Cycles through closures or bound methods (`self.callback = self.method`) that hold large objects until the collector runs.

Reading `getsizeof` of a container as the size of its contents.

Key takeaways

Every value is a heap object with a refcount and a type pointer; an int costs 28 bytes, a container costs 8 bytes per pointer.

Names, parameters and container slots are references; `is` is identity, `==` is value; copying is explicit.

Reference counting frees an object the moment its last reference goes; `del` removes a name.

Cycles need the generational collector; `gc.collect()` forces it; weak references do not keep objects alive.

`__slots__`, generators, raw arrays and sharing immutable data are the memory levers.

EXERCISE 1

Reference counting by hand

Simulate CPython's reference counting. Read commands until EOF: `new <var> <obj>` binds a variable to an object (creating the object if it is not live); `alias <var2> <var1>` binds `var2` to whatever `var1` refers to; `del <var>` unbinds. Rebinding or deleting a variable drops one reference from its old object, and when an object's count reaches zero print `freed <obj>` immediately. After the commands print `live <obj>=<count>` for every live object sorted by name, or `live none`.

Input: one command per line. **Output:** freed lines as they happen, then the live objects.

```
new a O1
alias b a
del a
new b O2
new c O2
```

prints

```
freed O1
live O2=2
```

INPUT 1

```
new a O1
alias b a
del a
new b O2
new c O2
```

OUTPUT 1

```
freed O1
live O2=2
```

REFERENCE SOLUTION

```
import sys

bindings = {}   # variable -> object name
counts = {}     # object name -> reference count


def release(var):
    """Drop the reference held by var, freeing the object at zero."""
    obj = bindings.pop(var, None)
    if obj is None:
        return
    counts[obj] -= 1
    if counts[obj] == 0:
        del counts[obj]
        print("freed", obj)


def bind(var, obj):
    release(var)
    bindings[var] = obj
    counts[obj] = counts.get(obj, 0) + 1


for line in sys.stdin:
    parts = line.split()
    if not parts:
        continue
    command = parts[0]
    if command == "new":
        bind(parts[1], parts[2])
    elif command == "alias":
        bind(parts[1], bindings[parts[2]])
    elif command == "del":
        release(parts[1])

if counts:
    for obj in sorted(counts):
        print(f"live {obj}={counts[obj]}")
else:
    print("live none")
```

EXERCISE 2

The cycle collector by hand

Simulate why reference counting needs a cycle collector. Read commands until EOF: `var <name> <obj>` binds a variable to an object, `ref <obj1> <obj2>` makes `obj1` hold a reference to `obj2`,

and `del <name>` unbinds a variable. An object's refcount is the number of variables bound to it plus the number of `ref` edges into it; an object is *reachable* if some bound variable reaches it through `ref` edges. Print `<obj> refcount=<n> <reachable|garbage>` for every object mentioned, sorted by name, then `collected <number of garbage objects>`.

Input: one command per line. **Output:** one line per object, then the count.

```
var a O1
var b O2
ref O1 O2
ref O2 O1
del a
del b
```

prints

```
O1 refcount=1 garbage
O2 refcount=1 garbage
collected 2
```

INPUT 1

```
var a O1
var b O2
ref O1 O2
ref O2 O1
del a
del b
```

OUTPUT 1

```
O1 refcount=1 garbage
O2 refcount=1 garbage
collected 2
```

REFERENCE SOLUTION

```
import sys
from collections import defaultdict

bindings = {}                 # variable → object
edges = defaultdict(list)     # object → objects it references
objects = set()

for line in sys.stdin:
    parts = line.split()
    if not parts:
        continue
    if parts[0] == "var":
        bindings[parts[1]] = parts[2]
        objects.add(parts[2])
    elif parts[0] == "ref":
        edges[parts[1]].append(parts[2])
        objects.update(parts[1:3])
    elif parts[0] == "del":
        bindings.pop(parts[1], None)


def reachable():
    seen = set()
    stack = list(bindings.values())
    while stack:
        obj = stack.pop()
        if obj in seen:
            continue
        seen.add(obj)
        stack.extend(edges[obj])
    return seen


alive = reachable()
counts = {obj: 0 for obj in objects}
for obj in bindings.values():
    counts[obj] += 1
for targets in edges.values():
    for target in targets:
        counts[target] += 1

garbage = 0
for obj in sorted(objects):
    status = "reachable" if obj in alive else "garbage"
    garbage += status == "garbage"
    print(f"{obj} refcount={counts[obj]} {status}")
print("collected", garbage)
```

QUIZ

1. What does `del x` do?

A. Frees the object immediately

B. Removes the name `x`, dropping one reference; the object is freed only if that was the last

C. Marks the object for the collector

D. Nothing

2. Why is `x is 5` wrong even though it often works?

A. `is` cannot compare ints

B. It depends on the small-int cache; identity is not value

C. 5 is not an object

D. It raises

3. Why can reference counting alone not free `a → b → a` after `del a, b`?

A. Lists are never freed

B. Each object still holds a reference to the other, so neither count reaches zero

C. `del` is lazy

D. They are interned

4. What does `__slots__` change?

A. Nothing at run time

B. Instances store attributes in fixed slots instead of a per-instance dict: less memory, faster access, no ad-hoc attributes

C. Makes the class immutable

D. Enables weak references

5. What is a `weakref.ref(obj)` for?

A. A faster reference

B. Referring to an object without keeping it alive — caches and registries

C. A copy

D. Thread safety

ANSWER KEY · THE OBJECT MODEL — OBJECTS, REFERENCES, REFERENCE COUNTING AND THE CYCLE COLLECTOR

1. B Reference counting frees at zero; another name or container slot keeps the object alive.

2. B `==` is value; `is` is identity, correct for singletons like `None`.

3. B The generational cycle collector finds unreachable cycles and frees them.

4. B Roughly 48 bytes per instance instead of ~150 for a small object.

5. B When the strong references go, the weak reference returns `None`.

LESSON 2 · 15 MIN

The cost model — what the built-in operations really cost

A Python program's speed is decided first by its algorithm and second by which built-in operations it leans on, because each built-in has a cost fixed by its implementation: appending to a list is amortised constant time, inserting at the front is linear, `x in some_list` scans, `x in some_set` hashes. Knowing the table below is what lets you look at a loop and see the quadratic hiding in it. This lesson gives the costs of the list, dict, set, string and deque operations, the hidden costs — copying, hashing, allocation, function calls — and the substitutions that turn a slow shape into a fast one.

The table

Operation	Cost	Note
`lst[i]`, `lst[i] = x`, `len(lst)`	O(1)	array of pointers
`lst.append(x)`, `lst.pop()`	O(1) amortised	over-allocation
`lst.insert(0, x)`, `lst.pop(0)`, `del lst[0]`	O(n)	shifts everything
`x in lst`, `lst.index(x)`, `lst.remove(x)`	O(n)	linear scan
`lst[a:b]`	O(b − a)	copies
`lst.sort()`, `sorted(xs)`	O(n log n)	Timsort; O(n) on nearly sorted data
`min`, `max`, `sum`	O(n)	C loop
`d[k]`, `d[k] = v`, `k in d`, `del d[k]`	O(1) average	hashing; O(n) worst case
`s.add`, `x in s`, `s.remove`	O(1) average	hashing
`a & b`, `a | b`, `a - b` (sets)	O(len a + len b)	
`str + str`	O(len)	a new string each time
`"".join(parts)`	O(total)	one allocation
`s in text`, `text.find(s)`	O(n)	fast C search
`deque.append/appendleft/pop/popleft`	O(1)	doubly linked blocks
`deque[i]`	O(n)	middle access is slow
`heapq.heappush/heappop`	O(log n)	`nsmallest(k)` is O(n log k)
`bisect.bisect`	O(log n)	but `insort` is O(n) for the shift

The shapes that hide a quadratic

```
result = []
for item in items:
    if item not in result:          # O(n) scan inside an O(n) loop: O(n²)
        result.append(item)

seen = set()                        # O(1) membership: O(n) overall
result = [x for x in items if not (x in seen or seen.add(x))]
```

```
text = ""
for piece in pieces:
    text += piece                   # may copy the whole string each time
text = "".join(pieces)              # one pass

queue = [...]
while queue:
    item = queue.pop(0)             # O(n) per pop: O(n²) drain
from collections import deque
queue = deque([...]); queue.popleft()   # O(1)

for i in range(len(xs)):
    for j in range(i):              # honest O(n²): sometimes necessary, alway
        ...
```

The pattern is always the same: a linear operation (`in` on a list, `pop(0)` , `insert(0)` , string concatenation, `list.remove`) inside a loop over the same data. The fix is the data structure whose operation is constant: a set or dict for membership, a deque for both ends, a join for strings, a heap for repeated minimums, `bisect` on a sorted list for range questions.

Sorting, and the cheap version of it

`sorted` with a key is O(n log n) and hard to beat with anything hand-written; sorting once and then binary searching (`bisect`) answers many queries in O(log n) each. `heapq.nlargest(k, xs)` is O(n log k) — much cheaper than sorting when k is small — and `min` / `max` are O(n) with no allocation. `sorted(xs)[:k]` when k = 3 is the common waste.

Hidden costs

Copying. A slice, `list(xs)`, `dict(d)`, `sorted(xs)`, `+` on sequences and `str.replace` all allocate a copy; inside a loop that is a hidden factor of n. `itertools.islice` and views (`d.keys()`, `memoryview`) avoid it.

Hashing. A dict lookup hashes the key; strings cache their hash, tuples recompute it from their elements each time. Very long tuple keys are not free.

Allocation. Every int above 256, every float, every tuple is an allocation; `sum(x * x for x in xs)` allocates a float per element. Vectorised NumPy (lesson 5) avoids the per-element object.

Function calls. A Python-level call costs on the order of 50–100 ns in 3.11 — cheap, but a million calls in a hot loop is 100 ms. Inlining the body or using a comprehension removes it; `map(f, xs)` with a built-in `f` stays in C.

Attribute and global lookup. `self.items.append` in a loop performs two lookups per iteration; `append = self.items.append` before the loop performs them once (lesson 3 explains why).

Exceptions. Raising is a few microseconds; a `try` that does not raise is free in 3.11 (zero-cost exceptions). `try/except KeyError` around a lookup that usually succeeds beats `if k in d: d[k]`.

Memory as a cost

Every structure above has a size: a dict costs roughly three times the memory of the equivalent list of tuples; a set of a million ints is ~32 MB; a list of a million floats is 8 MB of pointers plus 24 MB of floats. Memory is time — allocation, cache misses, the collector's scans — and the judge's 256 MB limit is reachable with careless intermediates. Generators, `array`, NumPy and `__slots__` are the levers.

An estimate you can do in your head

CPython executes roughly 10–50 million simple bytecode operations a second; a Python-level loop iteration doing a few operations costs about 50–100 ns; a C-level loop (`sum`, `max`, `str.join`, `sorted`, set operations) runs 10–100 times faster per element. So 10^6 iterations of a simple loop is ~0.1 s, 10^7 is a second, and 10^8 is not going to finish under a one-second limit unless it happens in C. Given n, count the operations, and you know before running whether the design is viable.

Pitfalls

`x in list` inside a loop.

`pop(0)` / `insert(0, …)` as a queue.

String concatenation in a loop to build output; use `join` or write lines as you go.

`sorted(xs)[0]` for the minimum, `sorted(xs)[:3]` for the top three.

Slicing inside a loop (`xs[i:]`) — each slice copies.

Estimating by intuition instead of counting operations.

Key takeaways

List index/append are O(1), front operations and membership O(n); dict and set operations are O(1) average; sorting is O(n log n).

A linear operation inside a loop over the same data is a quadratic; replace it with a set, dict, deque, join, heap or bisect.

Copies, hashing, allocation, function calls and attribute lookups are the hidden per-iteration costs.

Memory is time; generators, raw arrays and slots reduce it.

~10^7 simple Python operations a second: count the operations and predict the running time before you run.

EXERCISE 1

Sliding window maximum

Read `n k seed` . Generate `xs = [rng.randrange(1_000_000) for _ in range(n)]` with `rng = random.Random(seed)` . Compute the maximum of every window of `k` consecutive values in O(n) with a monotonic `collections.deque` of indices — the naive O(n·k) scan will not finish the hidden case. Print `windows <n - k + 1>` , `first <maximum of the first window>` , `last <maximum of the last window>` and `sum <sum of all window maxima>` .

Input: `n k seed` . **Output:** four lines.

```
8 3 1
```

prints

```
windows 6
first 888598
last 267459
sum 4575363
```

INPUT 1

```
8 3 1
```

OUTPUT 1

```
windows 6
first 888598
last 267459
sum 4575363
```

REFERENCE SOLUTION

```python
import random
from collections import deque

n, k, seed = map(int, input().split())
rng = random.Random(seed)
xs = [rng.randrange(1_000_000) for _ in range(n)]


def window_maxima(values, k):
    """Yield the maximum of each window of size k, in O(n)."""
    window = deque()
    for i, value in enumerate(values):
        while window and values[window[-1]] <= value:
            window.pop()
        window.append(i)
        if window[0] <= i - k:
            window.popleft()
        if i >= k - 1:
            yield values[window[0]]


maxima = list(window_maxima(xs, k))
print("windows", len(maxima))
print("first", maxima[0])
print("last", maxima[-1])
print("sum", sum(maxima))
```

EXERCISE 2

Membership at scale

Read `n m seed`. With `rng = random.Random(seed)`, generate `items = [rng.randrange(10**5) for _ in range(n)]` and then `queries = [rng.randrange(10**5) for _ in range(m)]` (in that order). Print `distinct <number of distinct items>` and `hits <number of queries that occur among the items>`. A list membership test per query is O(n·m) and will time out on the hidden case; build a set.

Input: `n m seed`. **Output:** two lines.

```
2000 2000 1
```

prints

```
distinct 1980
hits 44
```

INPUT 1

```
2000 2000 1
```

OUTPUT 1

```
distinct 1980
hits 44
```

REFERENCE SOLUTION

```python
import random

n, m, seed = map(int, input().split())
rng = random.Random(seed)
items = [rng.randrange(10**5) for _ in range(n)]
queries = [rng.randrange(10**5) for _ in range(m)]

present = set(items)
print("distinct", len(present))
print("hits", sum(q in present for q in queries))
```

QUIZ

1. What is the cost of `lst.pop(0)` on a list of n items?

A. O(1)

B. O(n) — every remaining element shifts

C. O(log n)

D. O(n log n)

2. Which loop is quadratic?

A. `for x in xs: seen.add(x)`

B. `for x in xs: if x not in result_list: result_list.append(x)`

C. `for x in xs: total += x`

D. `for x in sorted(xs): ...`

3. How should a string be built from ten thousand pieces?

A. `s += piece` in a loop

B. `"".join(pieces)`

C. `s = s + piece`

D. `str(pieces)`

4. What is the cost of `heapq.nlargest(k, xs)` and why prefer it over `sorted(xs)[-k:]`?

A. O(n log n); no reason

B. O(n log k) — much cheaper when k is small

C. O(k)

D. O(n²)

5. Roughly how many simple Python-level loop iterations run in a second?

A. About a thousand

B. About ten million

C. About a billion

D. Unbounded

ANSWER KEY · THE COST MODEL — WHAT THE BUILT-IN OPERATIONS REALLY COST

1. B `deque.popleft()` is O(1); a list is an array.

2. B A linear `in` on a list inside a linear loop over the same data.

3. B One allocation and one pass; repeated concatenation may copy the whole string each time.

4. B A heap of size k over n items; sorting does far more work for the top three.

5. B ~50–100 ns per iteration; 10^8 iterations will not finish under a one-second limit unless the loop runs in C.

LESSON 3 · 15 MIN

Bytecode and the interpreter — code objects, dis, name lookup and the 3.11 specialiser

CPython does not run your source; it compiles each function to *bytecode* — a sequence of small instructions for a stack machine — and an interpreter loop executes those. Seeing the bytecode explains the costs of the previous lesson: why a local variable is faster than a global, why `self.x` in a loop is two instructions where `x` is one, what a function call actually does, and what 3.11's specialising interpreter changed. This lesson covers the compilation pipeline and code objects, reading `dis` output, the four kinds of name load, the frame and the value stack, `.pyc` caching and constant folding, and the adaptive interpreter that makes 3.11 the fastest CPython so far.

From source to code object

```
src = "def f(a, b):\n    total = a + b * 2\n    return total\n"
code = compile(src, "<src>", "exec")     # a module code object
ns = {}
exec(code, ns)                           # runs it: defines f in ns
f = ns["f"]
f.__code__.co_varnames                   # ('a', 'b', 'total') — the locals
f.__code__.co_argcount                   # 2
f.__code__.co_consts                     # (None, 2)
f.__code__.co_names                      # () — no globals or attributes refe
```

Parsing produces an AST (`ast.parse`), the compiler turns it into a *code object* per module, function, class and comprehension, and `exec` or a call runs one. A code object is immutable data: `co_code` (the bytes), `co_consts` (constants, including nested code objects), `co_names` (global and attribute names), `co_varnames` (locals, parameters first), `co_argcount`, `co_nlocals`, `co_firstlineno` and a line table. A function object is a code object plus its globals, defaults and closure. The importer caches module code objects as `__pycache__/*.pyc` keyed by the source's modification time, so the compile happens once.

Reading dis

```
import dis
dis.dis(f)
```

```
  2           0 RESUME                   0
  3           2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 LOAD_CONST               1 (2)
              8 BINARY_OP                5 (*)
             10 BINARY_OP                0 (+)
             12 STORE_FAST               2 (total)
  4          14 LOAD_FAST                2 (total)
             16 RETURN_VALUE
```

Columns: source line, byte offset, opcode name, argument, and the resolved argument in parentheses. The machine is a stack: `LOAD_*` pushes, `BINARY_OP` pops two and pushes the result, `STORE_FAST` pops into a local slot, `RETURN_VALUE` pops and returns. `dis.get_instructions(f)` yields the same as `Instruction` objects (`opname`, `arg`, `argval`) for programs that want to count or inspect. The exact instruction set changes between minor versions — 3.11 introduced `RESUME`, `BINARY_OP` and the cache entries below — so bytecode inspection is a per-version diagnostic, never something to print in a portable program's output.

The four loads

Instruction	Used for	Cost
`LOAD_CONST`	literals, folded constants	array index into `co_consts`
`LOAD_FAST`	local variables and parameters	array index into the frame's slots
`LOAD_GLOBAL`	module globals and built-ins	a dict lookup in globals, then in builtins
`LOAD_ATTR` / `LOAD_METHOD`	`obj.name`	type lookup, descriptor check, instance dict
`LOAD_DEREF`	closure variables	cell dereference

That table is the reason for two classic micro-optimisations: binding a global or built-in to a local before a hot loop (`_len = len`) turns dict lookups into slot reads, and binding a method (`append = out.append`) removes an attribute lookup per iteration. In 3.11 the specialiser (below) narrows the gap, so these matter less than they did — measure before rewriting for them.

Frames and calls

A call creates a *frame*: the code object, a slot array for locals, a value stack, the instruction pointer, and a link to the caller. 3.11 allocates frames in a contiguous per-thread stack and skips creating a Python frame object unless something asks for it (a traceback, `sys._getframe`), which made calls about 1.3× cheaper than 3.10. `sys.setrecursionlimit` bounds the depth; the default 1000 protects the C stack. Generators keep a suspended frame; a closure references cells rather than the enclosing frame.

The compiler's own optimisations

`x = 2 * 60 * 60` stores `7200` — constant folding; `if 0:` blocks are dropped; `"a" "b"` is joined; a `for` over a literal list `[1, 2, 3]` iterates a constant tuple; `x in [a, b]` with constants becomes a frozenset test. Nothing else: no inlining, no loop hoisting, no type inference. What you write is what runs, one instruction per operation.

The 3.11 specialising adaptive interpreter

PEP 659: generic instructions rewrite themselves at run time into specialised versions once they observe stable types. `BINARY_OP` on two ints becomes `BINARY_OP_ADD_INT`; `LOAD_ATTR` on an instance whose class has not changed becomes `LOAD_ATTR_INSTANCE_VALUE` with an inline cache holding the slot index; `LOAD_GLOBAL` caches the dict version and the index; `CALL` to a Python function skips the generic machinery. Each adaptive instruction has cache entries after it in the bytecode (the invisible gaps in `dis` offsets). Together with the cheaper frames and *zero-cost exceptions* — a `try` block costs nothing unless it raises, because the handler is found from a table rather than pushed at run time — 3.11 is 10–60 % faster than 3.10 on typical code, with no change to the source. The consequences for you: stable types in hot loops specialise well (mixing ints and floats in one loop de-specialises), attribute access on plain classes is fast, and `try/except` around a usually-successful operation is the right idiom.

What the bytecode says about style

Comprehensions compile to their own code object with a tight loop; `map` and `filter` with C functions stay in C for the whole iteration; a `while` loop with manual index arithmetic executes several instructions per step that `for … in range` performs in C. Reading `dis` of two candidate versions of a hot function is the quickest way to see which does less work — provided you then measure, because instruction count is not time.

Pitfalls

- Printing `dis` output or instruction counts from a program that should run on more than one version.
- Assuming a global lookup is free in a hot loop (it is a dict lookup, cached or not).
- Deep recursion without raising the limit — the default 1000 is a `RecursionError`, not a segfault.
- Expecting the compiler to hoist or inline anything.
- Mixed types in a hot loop that keep de-specialising.
- Relying on `.pyc` files as a distribution format (they are version-specific).

Key takeaways

- Source compiles to code objects (`co_code`, `co_consts`, `co_names`, `co_varnames`); functions are code plus globals, defaults and closure; `.pyc` caches the compile.
- `dis` shows the stack machine: loads push, operations pop and push, stores pop; instruction sets differ per version.
- `LOAD_FAST` is a slot read, `LOAD_GLOBAL` a dict lookup, `LOAD_ATTR` a type-and-instance lookup — the source of the local-binding optimisations.
- Frames are cheap in 3.11; the compiler folds constants and nothing more.
- The specialising interpreter rewrites instructions for observed types, exceptions are zero-cost, and stable types in hot loops are what it rewards.

EXERCISE 1

Inspect a code object

Read Python source from standard input that defines a function `f`. Compile it with `compile(src, "<src>", "exec")`, run it with `exec` into a fresh namespace, take `f.__code__`, and print four lines: `names <co_names joined by ", ">`, `varnames <co_varnames joined>`, `argcount <co_argcount>` and `consts <co_consts joined>` where each constant is shown by `repr` except nested code objects, shown as `<code NAME>`. An empty tuple prints as `-`.

Input: the source. **Output:** four lines.

```
def f(a, b):
    total = a + b * 2
    return total
```

prints

```
names -
varnames a, b, total
argcount 2
consts None, 2
```

INPUT 1

```
def f(a, b):
    total = a + b * 2
    return total
```

OUTPUT 1

```
names -
varnames a, b, total
argcount 2
consts None, 2
```

REFERENCE SOLUTION

```
import sys
import types

src = sys.stdin.read()
namespace = {}
exec(compile(src, "<src>", "exec"), namespace)
code = namespace["f"].__code__


def show(value):
    if isinstance(value, types.CodeType):
        return f"<code {value.co_name}>"
    return repr(value)


def joined(items):
    return ", ".join(items) if items else "-"


print("names", joined(code.co_names))
print("varnames", joined(code.co_varnames))
print("argcount", code.co_argcount)
print("consts", joined(show(c) for c in code.co_consts))
```

EXERCISE 2

Classify the names

Read Python source that defines a function `f`, compile and run it as before. Using `f.__code__` and `dis.get_instructions(f)`, print `locals <co_varnames joined by ", ">`, `globals <sorted distinct names loaded by LOAD_GLOBAL>` and `attributes <sorted names in co_names that are not globals>` — an empty group prints `-`. Whether a name is a local, a global or an attribute is decided by the compiler and is the same in every version, unlike instruction counts.

Input: the source. **Output:** three lines.

```
def f(items):
    out = []
    for item in items:
        out.append(len(str(item)))
    return out
```

prints

```
locals items, out, item
globals len, str
attributes append
```

INPUT 1

```
def f(items):
    out = []
    for item in items:
        out.append(len(str(item)))
    return out
```

OUTPUT 1

```
locals items, out, item
globals len, str
attributes append
```

REFERENCE SOLUTION

```
import dis
import sys

src = sys.stdin.read()
namespace = {}
exec(compile(src, "<src>", "exec"), namespace)
f = namespace["f"]


def joined(names):
    return ", ".join(names) if names else "-"


code = f.__code__
global_names = {ins.argval for ins in dis.get_instructions(f) if ins.opname == "LOAD
attribute_names = sorted(name for name in code.co_names if name not in global_names)

print("locals", joined(code.co_varnames))
print("globals", joined(sorted(global_names)))
print("attributes", joined(attribute_names))
```

QUIZ

1. What does `LOAD_FAST` do?

A. A dict lookup in globals

B. Reads a local variable from the frame's slot array by index

C. Loads a constant

D. Calls a function

2. What does the compiler do with `x = 2 * 60 * 60`?

A. Emits two multiplications

B. Folds it to the constant 7200

C. Raises

D. Defers it

3. What does 3.11's specialising interpreter do?

A. Compiles to machine code

B. Rewrites generic instructions into type-specialised versions with inline caches once types prove stable

C. Removes the GIL

D. Parallelises loops

4. What does a `try` block cost in 3.11 when nothing is raised?

A. A stack push per entry

B. Nothing — handlers are found from a table only when an exception occurs

C. A function call

D. A dict lookup

5. Why must a program never print `dis` output or instruction counts as its answer?

A. `dis` is slow

B. The instruction set changes between minor versions, so the output is not portable

C. It is disallowed

D. It prints to stderr

ANSWER KEY · BYTECODE AND THE INTERPRETER — CODE OBJECTS, DIS, NAME LOOKUP AND THE 3.11 SPECIALISER

1. B That is why locals are cheaper than globals (`LOAD_GLOBAL` is a dict lookup, cached or not).

2. B Constant folding is one of the few optimisations the compiler performs; it hoists and inlines nothing.

3. B `BINARY_OP` on two ints becomes `BINARY_OP_ADD_INT` ; mixed types in a hot loop de-specialise.

4. B Zero-cost exceptions make `try/except` the right idiom for the rare failing case.

5. B Bytecode is a per-version diagnostic; code-object names and constants are stable, instruction sets are not.

LESSON 4 · 14 MIN

Measuring — timeit, perf_counter, cProfile, tracemalloc and benchmarking hygiene

Performance work without measurement is guessing, and most guesses about where a program spends its time are wrong. Python ships the tools to find out: `time.perf_counter` for a stopwatch, `timeit` for micro-benchmarks that handle repetition and noise, `cProfile` with `pstats` to attribute time to functions, and `tracemalloc` to attribute memory to lines. This lesson covers each, how to read their output, the hygiene that makes a measurement mean something — warm-up, repetition, minimum rather than mean, isolating the thing measured — and the discipline of profiling before optimising and stopping when the numbers say so.

---

perf_counter: the stopwatch

```
from time import perf_counter

start = perf_counter()
result = work()
elapsed = perf_counter() - start        # seconds, as a float, monotonic, high
print(f"work took {elapsed:.3f}s")
```

`perf_counter` is the clock for measuring durations (monotonic, fractional seconds, not affected by system clock changes); `time.time()` is wall-clock and can jump; `process_time` counts CPU time only. A small context manager wrapping this is the standard tool for timing phases of a real program:

```
from contextlib import contextmanager

@contextmanager
def timed(label, clock=perf_counter, report=print):
    start = clock()
    yield
    report(f"{label} {clock() - start:.3f}s")
```

Injecting the clock and the reporter is what makes it testable: a test passes a fake clock returning a scripted sequence and asserts on the labels and durations without any real time passing — the design from Module 18.

timeit: micro-benchmarks done right

```
python -m timeit -s "xs = list(range(1000))" "sum(xs)"
# 5000 loops, best of 5: 4.1 usec per loop
python -m timeit -s "xs = list(range(1000))" "total = 0" "for x in xs: total +
# 5000 loops, best of 5: 28 usec per loop
```

```
import timeit
timeit.timeit("sum(xs)", setup="xs = list(range(1000))", number=10_000)     #
min(timeit.repeat("sum(xs)", setup=..., number=10_000, repeat=5))           #
```

`timeit` disables garbage collection, runs the statement `number` times, repeats that `repeat` times and reports the *best* — the minimum is the right statistic for a benchmark, because noise only ever adds time. Setup code (building the data) is excluded. Compare alternatives on the same machine in the same session, at the same input size, and with a size large enough that the per-call overhead is not what you are measuring.

cProfile: where the time goes

```
python -m cProfile -s cumulative app.py | head -20
```

```
         2003 function calls in 1.204 seconds
   Ordered by: cumulative time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    1.204    1.204 app.py:30(main)
     1000    0.902    0.001    1.150    0.001 app.py:12(parse)
     1000    0.248    0.000    0.248    0.000 app.py:5(tokenize)
```

`ncalls` is how often the function ran; `tottime` is time inside it *excluding* callees; `cumtime` includes callees; `percall` divides each. Sort by `tottime` to find the function doing the work, by `cumtime` to find the path that leads there. In code: `cProfile.Profile()` as a context manager, then `pstats.Stats(profile).sort_stats("tottime").print_stats(10)`. The profiler adds overhead (roughly 2×) that is uneven across call-heavy code, so use it to find the hot spot and `timeit` to compare fixes. Line-level profilers (`line_profiler`) and sampling profilers (`py-spy`, which attaches to a running process with no code changes) are the third-party next steps; `python -X importtime` profiles import time specifically.

tracemalloc: where the memory goes

```
import tracemalloc
tracemalloc.start()
before = tracemalloc.take_snapshot()
build()
after = tracemalloc.take_snapshot()
for stat in after.compare_to(before, "lineno")[:5]:
    print(stat)                       # file:line: size=…, count=…, average=…
current, peak = tracemalloc.get_traced_memory()
```

`tracemalloc` records every allocation with its call site; comparing two snapshots shows which lines allocated the growth. `get_traced_memory` gives the current and peak — the peak is what decides whether a program fits its limit. It slows the program and itself uses memory; run it once to find the culprit, then remove it.

Hygiene

Warm up. The first run pays for imports, `.pyc` compilation, cache filling and the 3.11 specialiser; measure the second run onward.

Repeat and take the minimum for micro-benchmarks; for whole programs, several runs and the median.

Isolate. Time the computation, not the `print`, not the input parsing, not the file open — unless those are the question.

Same conditions. Same machine, same Python, same data, nothing else running; a laptop on battery throttles.

Realistic size. Constant factors dominate small inputs; asymptotics dominate large ones; measure at the size that matters.

Hypothesis first. "parse is 75 % of the time" is a claim the profile confirms or refutes before any code changes.

Stop. When the target is met or the remaining hot spot is essential work, stop; every optimisation costs clarity.

Amdahl's rule

If a phase is 75 % of the run time, making it infinitely fast gains at most 4×; making a 5 % phase infinitely fast gains 5 %. Optimise the largest slice first, re-profile, repeat — the largest slice changes after each fix. This single division is the difference between a productive afternoon and a wasted one.

Pitfalls

- `time.time()` for durations.
- Reporting the mean of a noisy benchmark instead of the minimum.
- Timing code that includes I/O or the profiler's own overhead.
- Optimising the function that *looks* slow without profiling.
- Measuring at n = 10 and extrapolating to n = 10^7.
- Printing measured times from a program whose output must be reproducible — report computed quantities instead.

Key takeaways

- `perf_counter` for durations, wrapped in a context manager with an injectable clock; `timeit` for micro-benchmarks with `repeat` and the minimum.
- `cProfile` attributes time to functions: `tottime` finds the worker, `cumtime` the path; `pstats` sorts and prints.
- `tracemalloc` snapshots attribute memory growth to lines and report the peak.
- Warm up, repeat, isolate, fix conditions and sizes, form a hypothesis, stop when done.
- Amdahl: optimise the largest slice; re-profile after each change.

EXERCISE 1

Read a profile

Read lines `<ncalls> <tottime> <cumtime> <function>` until EOF — the columns of a `cProfile` report. Let the total be the sum of `tottime`. Print the top three functions by `tottime` (descending; ties by name) as `<function> tottime=<t:.3f> percall=<t/ncalls:.6f> share=<percentage of total:.1f>%`, then `total <sum:.3f>s`.

Input: one function per line. **Output:** up to three lines, then the total.

```
1 0.001 1.204 main
1000 0.902 1.150 parse
1000 0.248 0.248 tokenize
```

prints

```
parse tottime=0.902 percall=0.000902 share=78.4%
tokenize tottime=0.248 percall=0.000248 share=21.5%
main tottime=0.001 percall=0.001000 share=0.1%
total 1.151s
```

INPUT 1

```
1 0.001 1.204 main
1000 0.902 1.150 parse
1000 0.248 0.248 tokenize
```

OUTPUT 1

```
parse tottime=0.902 percall=0.000902
share=78.4%
tokenize tottime=0.248 percall=0.000248
share=21.5%
main tottime=0.001 percall=0.001000
share=0.1%
total 1.151s
```

REFERENCE SOLUTION

```python
import sys

rows = []
for line in sys.stdin:
    parts = line.split()
    if len(parts) != 4:
        continue
    ncalls, tottime, cumtime, name = int(parts[0]), float(parts[1]), float(parts[2])
    rows.append((name, ncalls, tottime, cumtime))

total = sum(row[2] for row in rows)
for name, ncalls, tottime, _ in sorted(rows, key=lambda row: (-row[2], row[0]))[:3]:
    print(f"{name} tottime={tottime:.3f} percall={tottime / ncalls:.6f} share={100 *
print(f"total {total:.3f}s")
```

EXERCISE 2

A stopwatch with an injected clock

Write `Stopwatch(clock)` whose `measure(label)` is a context manager (use `contextlib.contextmanager`) that reads `clock()` on entry and on exit and records `(label, elapsed)`. Read a line of clock readings (floats) and a line of labels; make the clock return the

readings in order (`iter(readings).__next__`) and measure one empty block per label. Print `<label> <elapsed:.3f>s` per label, `total <sum:.3f>s` and `slowest <label>` (the first maximum).

Input: the readings, then the labels. **Output:** one line per label, then two summary lines.

```
0.0 0.25 0.3 1.1
parse render
```

prints

```
parse 0.250s
render 0.800s
total 1.050s
slowest render
```

INPUT 1

```
0.0 0.25 0.3 1.1
parse render
```

OUTPUT 1

```
parse 0.250s
render 0.800s
total 1.050s
slowest render
```

REFERENCE SOLUTION

```
from contextlib import contextmanager


class Stopwatch:
    def __init__(self, clock):
        self.clock = clock
        self.records = []

    @contextmanager
    def measure(self, label):
        start = self.clock()
        try:
            yield
        finally:
            self.records.append((label, self.clock() - start))


readings = [float(x) for x in input().split()]
labels = input().split()
clock = iter(readings).__next__
watch = Stopwatch(clock)
for label in labels:
    with watch.measure(label):
        pass

for label, elapsed in watch.records:
    print(f"{label} {elapsed:.3f}s")
print(f"total {sum(elapsed for _, elapsed in watch.records):.3f}s")
print("slowest", max(watch.records, key=lambda record: record[1])[0])
```

QUIZ

1. Which clock is right for measuring a duration?

A. `time.time()`

B. `time.perf_counter()`

C. `datetime.now()`

D. `time.ctime()`

2. Which statistic should a `timeit` benchmark report?

A. The mean

B. The minimum of the repeats — noise only adds time

C. The maximum

D. The first run

3. In a `cProfile` report, what is `tottime`?

A. Time including callees

B. Time inside the function itself, excluding callees

C. Wall-clock time

D. Time per call

4. What does `tracemalloc` attribute allocations to?

A. Threads

B. The source lines (call sites) that allocated

C. Types only

D. Nothing

5. A phase takes 75 % of the run time. What is the most its optimisation can gain?

A. 75 %

B. 4× — Amdahl's rule

C. Unlimited

D. 1.75×

ANSWER KEY · MEASURING — TIMEIT, PERF_COUNTER, CPROFILE, TRACEMALLOC AND BENCHMARKING HYGIENE

1. B Monotonic and high resolution; wall-clock time can jump.

2. B `min(timeit.repeat(...))` is the number to quote.

3. B `cumtime` includes callees; `tottime` finds the function doing the work.

4. B Comparing two snapshots shows which lines caused the growth; `get_traced_memory` gives current and peak.

5. B Even infinitely fast, the other 25 % remains; optimise the largest slice first and re-profile.

LESSON 5 · 14 MIN

Numeric performance — boxed numbers, array, bytes and NumPy vectorisation

Numeric code is where Python's object model costs the most: every integer and float in a list is a separate heap object, every `+` dispatches through the type, and a loop that adds a million floats performs a million allocations. The escape is to keep numbers *unboxed* — raw machine values in a contiguous block — and to run the loop in C: built-ins such as `sum` and `max`, the `array` and `bytes` types, and above all NumPy, whose arrays and vectorised operations are how numeric Python reaches C speed without leaving Python. This lesson covers why the loop is slow, the standard-library raw types (`array`, `bytes`, `bytearray`, `memoryview`), NumPy arrays, dtypes and broadcasting, the vectorisation habit, reductions, and what remains for Numba and Cython.

Why the loop is slow

```
total = 0.0
for x in xs:                # each x: a pointer to a 24-byte float object
    total += x * 2.0        # x * 2.0 allocates a float; += allocates another
```

Per iteration: two type dispatches, two allocations, two reference-count updates, and the interpreter loop overhead. Roughly 50–100 ns each, so 10^7 elements is a second. `sum(x * 2.0 for x in xs)` removes the `+=` object churn and the loop bytecode but still allocates per element; `2.0 * sum(xs)` is a C loop over pointers and ten times faster; a NumPy `2.0 * xs` over a float64 array is a C loop over raw doubles, a hundred times faster, and it also uses a tenth of the memory.

array, bytes, bytearray, memoryview

```
from array import array
a = array("d", [1.0, 2.0, 3.0])       # typecode: 'b' 'B' 'i' 'I' 'q' 'Q' 'f'
a.append(4.0); a[0]; sum(a)           # list-like, 8 bytes per element, no obj
a.tobytes(); array("d").frombytes(raw)

data = bytes(range(256))              # immutable bytes; indexing gives ints
data.count(0); data.find(b"\x10"); sum(data)      # C-speed scans
buf = bytearray(1024)                 # mutable; buf[0] = 255
view = memoryview(buf)[512:]          # a window with no copy; view[0] = 1 wr
```

`array` is the standard library's typed, compact sequence: the same memory layout as C, iteration still yields Python objects, but storage and I/O (`tofile`, `frombytes`) are raw. `bytes` and `bytearray` are byte sequences with fast C methods (`count`, `find`, `translate`, `split`); `memoryview` slices any buffer without copying — the tool for parsing binary formats and for passing sub-ranges to functions without allocation. `struct.pack` / `unpack` convert between Python values and packed bytes.

NumPy arrays

```
import numpy as np
xs = np.array([1.0, 2.5, 4.0])                  # dtype float64, contiguous
xs = np.arange(1_000_000, dtype=np.int64)       # 8 MB, one object
xs.shape, xs.dtype, xs.ndim                     # ((1000000,), dtype('int64'),
ys = xs * 2 + 1                                 # elementwise, one C loop, no
mask = xs % 3 == 0                              # a boolean array
xs[mask].sum(); (xs > 500_000).sum()            # boolean indexing; counting
xs.mean(), xs.max(), xs.std(), np.sqrt(xs)      # reductions and ufuncs
grid = np.zeros((3, 4)); grid[1, :] = 5         # 2-D, row 1
np.dot(a, b); a @ b                             # linear algebra in BLAS
```

An `ndarray` is a typed block of memory with a shape; operations are *ufuncs* that loop in C and produce new arrays (or write in place with `out=`, `+=`). `dtype` matters: `int64` overflows silently at 2^{63} (no automatic promotion to big ints), `float32` halves memory and precision, `bool` arrays are one byte per element. Conversion between a Python list and an array (`np.array(list)`, `arr.tolist()`) is O(n) and boxes every element — do it once at the boundary, never inside a loop.

Broadcasting

```
row = np.array([1, 2, 3])            # shape (3,)
col = np.array([[10], [20]])         # shape (2, 1)
row + col                            # shape (2, 3): [[11, 12, 13], [21, 22, 2
prices * (1 - discount[:, None])     # a column of discounts applied to each r
```

Arrays of different shapes combine by *broadcasting*: dimensions are aligned from the right, a dimension of size 1 stretches to match, and the loop runs once over the result. It replaces nested loops over rows and columns with one expression, and it is the mechanism behind "subtract the column mean from every row" and every normalisation.

The vectorisation habit

The rule: no Python-level loop over array elements. `for i in range(len(xs)): ys[i] = f(xs[i])` throws away the whole benefit; the vectorised form is `ys = f(xs)` where `f` is composed of ufuncs and slicing. Conditionals become `np.where(cond, a, b)`; cumulative operations `np.cumsum`; windows, `np.convolve` or stride tricks; pairwise distances, broadcasting; grouping, `np.unique(return_counts=True)` or `np.bincount`. When a computation genuinely needs a per-element loop with state that will not vectorise, that is the case for Numba (`@njit` compiles the loop to machine code) or Cython — both outside the standard library and outside the judge.

Reductions and precision

`xs.sum()` on float64 uses pairwise summation and is more accurate than a naive loop; `math.fsum` is exact-rounded for Python floats. Integer reductions on `int64` overflow silently — `np.arange(10**6)**3` wraps — so choose `dtype=np.float64` or `object` when values grow, or compute in Python ints when exactness matters. `np.isclose` / `np.allclose` compare floats with tolerance.

Where the standard library is enough

`sum`, `max`, `min`, `sorted`, `math.fsum`, `statistics.mean/pstdev`, `bytes.count`, `str.count`, `collections.Counter`, `itertools.accumulate` are all C loops; for a single pass over a list of

a million numbers they are within a small factor of NumPy and need no conversion. NumPy earns its import when there are several operations over the same data, 2-D structure, broadcasting, or arrays that live for the whole program.

Pitfalls

A Python loop over a NumPy array element by element.

Growing an array with `np.append` in a loop (each call copies) — build a list, convert once, or preallocate.

`int64` overflow read as a bug in the algorithm.

Converting list ↔ array inside a hot loop.

`arr == list` comparisons producing an array where a boolean was expected (`if arr == x:` raises).

Importing NumPy for one `sum`.

Key takeaways

Boxed numbers cost an allocation and a dispatch per operation; the fix is unboxed storage plus C loops.

`array`, `bytes`, `bytearray` and `memoryview` are the standard library's raw, copy-free types.

NumPy arrays are typed blocks; ufuncs, boolean masks, reductions and broadcasting replace loops; dtype decides overflow and memory.

Vectorise: no Python loop over elements; `where`, `cumsum`, `unique`, `bincount` express the common loops.

Built-in reductions are C loops too; NumPy pays off for repeated, multi-dimensional or long-lived numeric data.

EXERCISE 1

Vectorised statistics

Read `n seed`. Generate `xs = [rng.randrange(1000) for _ in range(n)]` with `rng = random.Random(seed)`, convert once to a NumPy array with `dtype=np.int64`, and compute everything with array operations — no Python loop over elements. Print `count <n>`, `mean`

`<mean:.3f>`, `max <max>`, `above 500 = <count of values greater than 500>`, `evens <count of even values>` and `sumsq <sum of squares>`.

Input: `n seed`. **Output:** six lines.

```
10 1
```

prints

```
count 10
mean 492.000
max 867
above 500 = 6
evens 4
sumsq 3345254
```

INPUT 1

```
10 1
```

OUTPUT 1

```
count 10
mean 492.000
max 867
above 500 = 6
evens 4
sumsq 3345254
```

REFERENCE SOLUTION

```python
import random

import numpy as np

n, seed = map(int, input().split())
rng = random.Random(seed)
xs = [rng.randrange(1000) for _ in range(n)]
arr = np.array(xs, dtype=np.int64)

print("count", arr.size)
print(f"mean {arr.mean():.3f}")
print("max", int(arr.max()))
print("above 500 =", int((arr > 500).sum()))
print("evens", int((arr % 2 == 0).sum()))
print("sumsq", int((arr * arr).sum()))
```

EXERCISE 2

Checksums over bytes

Read `n seed`. Generate `data = random.Random(seed).randbytes(n)`. Print `bytes <n>`, `zeros <data.count(0)>`, `max <max(data)>`, `sum16 <sum(data) % 65536>` and `fletcher16 <checksum>`, where the Fletcher-16 checksum iterates over a `memoryview` of the data keeping two running sums modulo 255 (`s1 = (s1 + byte) % 255`, `s2 = (s2 + s1) % 255`) and returns `(s2 << 8) | s1`. The first four values come from C-speed byte methods; the last is the one Python loop.

Input: `n seed`. **Output:** five lines.

```
16 1
```

prints

```
bytes 16
zeros 0
max 245
sum16 2245
fletcher16 32461
```

INPUT 1

```
16 1
```

OUTPUT 1

```
bytes 16
zeros 0
max 245
sum16 2245
fletcher16 32461
```

REFERENCE SOLUTION

```python
import random

n, seed = map(int, input().split())
data = random.Random(seed).randbytes(n)


def fletcher16(buffer):
    s1 = s2 = 0
    for byte in memoryview(buffer):
        s1 = (s1 + byte) % 255
        s2 = (s2 + s1) % 255
    return (s2 << 8) | s1


print("bytes", n)
print("zeros", data.count(0))
print("max", max(data))
print("sum16", sum(data) % 65536)
print("fletcher16", fletcher16(data))
```

QUIZ

1. Why is a Python loop summing floats slow?

A. Floats are big

B. Each element is a boxed object: a dispatch, an allocation and refcount updates per operation

C. Loops are compiled

D. The GIL

2. What does `memoryview(buf)[512:]` cost?

A. A copy of the tail

B. Nothing but a small view object — writes go through to `buf`

C. A new `bytes`

D. A list

3. What happens to `np.arange(10**6, dtype=np.int64) ** 3` at the top?

A. Promotion to Python ints

B. Silent overflow — int64 wraps

C. An exception

D. Conversion to float

4. Which expression is vectorised?

A. `[f(x) for x in arr]`

B. `np.where(arr > 0, arr, 0)`

C. `for i in range(len(arr)): out[i] = arr[i] * 2`

D. `list(map(f, arr))`

5. How do `row` of shape (3,) and `col` of shape (2, 1) add?

A. They cannot

B. By broadcasting to shape (2, 3)

C. Elementwise after truncation

D. As a dot product

ANSWER KEY · NUMERIC PERFORMANCE — BOXED NUMBERS, ARRAY, BYTES AND NUMPY VECTORISATION

1. B Unboxed storage plus a C loop — `sum`, `array`, NumPy — removes all three costs.

2. B Views slice any buffer without copying.

3. B Choose the dtype for the values' range; NumPy does not grow integers.

4. B One C loop over the whole array; the others iterate element by element in Python.

5. B Dimensions align from the right and size-1 dimensions stretch.

LESSON 6 · 14 MIN

Writing fast Python — the checklist, from algorithm to micro-optimisation

Making Python fast is a sequence of questions asked in a fixed order, because each level dwarfs the next: is the algorithm right, is the data structure right, is the work happening in C, is the hot loop lean, and only then the micro-optimisations that shave constants. A change at the first level turns hours into seconds; a change at the last turns a second into 0.8 of one. This lesson is the checklist, with the idioms at each level — precomputation and memoisation, the right container, built-ins and comprehensions, fast I/O, local binding, `__slots__`, early exit — and the judgement about when to stop.

---

1. The algorithm

Count the operations as a function of n before writing code. A nested loop over the same data is n²; a sort is n log n; a single pass is n. Most "Python is slow" problems are quadratic algorithms that would be slow in C too. The classic fixes: **precompute** what is asked repeatedly (prefix sums answer any range sum in O(1) after an O(n) pass; a dict from key to row replaces a search per query), **memoise** overlapping subproblems (`@functools.cache` on a recursive function turns exponential into polynomial, or write the table iteratively), **sort once** and binary-search, **hash** for membership and grouping, **early exit** when the answer is known.

```
prefix = [0]
for x in xs:
    prefix.append(prefix[-1] + x)
range_sum = lambda i, j: prefix[j] - prefix[i]     # O(1) per query

@functools.cache
def paths(r, c):
    if blocked[r][c]: return 0
    if r == c == 0: return 1
    return (paths(r - 1, c) if r else 0) + (paths(r, c - 1) if c else 0)
```

---

2. The data structure

From the cost model: `set` / `dict` for membership and lookup, `deque` for a queue, `heapq` for repeated minimum, `bisect` for sorted queries, `Counter` for tallies, `defaultdict(list)` for grouping, tuples for fixed records, `array` /NumPy for numbers. Choosing the structure is choosing the complexity of every operation the rest of the program performs.

---

3. Do the work in C

Built-ins and C-implemented methods run the whole loop without the interpreter: `sum`, `min`, `max`, `any`, `all`, `sorted` with a key, `str.join`, `str.split`, `str.count`, `list.sort`, `map` / `filter` with a built-in or C function, `set` operations, `dict.update`, `itertools`, `Counter`, `bytes` methods, `re` on a whole string. A comprehension is faster than an explicit loop with `append` (its own code object, no method lookup per iteration), and a generator expression inside `sum` avoids the intermediate list. The general form: express the computation as a pipeline of built-ins over whole collections rather than as element-by-element statements.

---

4. Fast I/O

```
import sys
data = sys.stdin.buffer.read().split()       # one read, one split: bytes toke
n = int(data[0]); xs = list(map(int, data[1:n + 1]))

out = []
for row in rows:
    out.append(f"{row.name} {row.total}")
sys.stdout.write("\n".join(out) + "\n")      # one write, or print("\n".join(
```

`input()` per line and `print` per line each cost a system call and a decode/encode; for a hundred thousand lines that is most of the run time. Read everything once, split, convert with `map(int, …)`; build output in a list and write it once. `sys.stdin.readline` is the middle ground when lines must be processed as they arrive.

---

5. The hot loop

Once the profiler names the loop that matters:

Hoist everything loop-invariant out of it: attribute lookups (`append = out.append`), global and built-in names (`_len = len`), constant expressions, a `re.compile`.

Avoid allocations per iteration — slicing, `str` concatenation, tuple packing where not needed, creating a small object that is immediately discarded.

Keep types stable so 3.11's specialiser stays specialised.

Use `try/except` for the rare case instead of a check per iteration.

Replace a Python-level function call per element with an inline expression, or the whole loop with a comprehension or C-level call.

`__slots__` on classes instantiated in the millions: less memory and faster attribute access.

```
def count_matches(words, vocab):
    contains = vocab.__contains__          # hoisted bound method
    return sum(1 for w in words if contains(w))
```

Each of these is worth 10–40 % on a loop that is already the bottleneck, and nothing on a loop that is not.

---

6. The escape hatches

When the algorithm is optimal and the loop is lean and it is still too slow: NumPy for numeric arrays; `multiprocessing` for CPU-bound work that splits; `functools.lru_cache` for repeated calls; a C extension, Cython, Numba or Rust via PyO3 for the innermost kernel; PyPy for a whole program that is pure Python. Each is a real project decision, not a first resort.

When to stop

The target is a number — under a second, under 256 MB, ten thousand requests a second — and the checklist runs until it is met, no further. Every optimisation trades readability, and the readable version is what gets maintained. Profile, apply the highest-level fix that applies, measure, and if the target is met, leave the rest alone. The `-O` flag, `.pyc` files and "avoid function calls" folklore are not on the list because their effect is negligible next to the algorithm and the data structure.

A worked pass

A script that reads 200 000 records, finds duplicates and prints summaries takes 40 s. Profile: 38 s in `if record in seen_list`. Level 2: `seen` becomes a set — 1.2 s. Profile again: 0.8 s in `print` per line. Level 4: collect and write once — 0.5 s. Target was 2 s; stop. The two changes were four lines; no loop was hand-tuned.

Pitfalls

- Micro-optimising before profiling.
- Tuning a loop that is 3 % of the run time.
- `input()` / `print()` per line at scale.
- A memoised recursive function without `sys.setrecursionlimit` on deep inputs.
- Rewriting readable code into a comprehension nobody can read for 5 %.
- Reaching for C before checking the algorithm.

Key takeaways

- Order of attack: algorithm, data structure, work in C, I/O, the hot loop, escape hatches — each level dwarfs the next.
- Precompute, memoise, sort-and-bisect, hash, early-exit are the algorithmic moves; prefix sums and `@cache` are the two to know cold.
- Built-ins, comprehensions and `join` keep loops in C; read input once and write output once.
- In the hot loop only: hoist lookups, avoid allocations, keep types stable, `try/except` for the rare case, `__slots__` for many instances.
- Stop when the target is met; readability is the default.

EXERCISE 1

Grid paths with @cache

Read `n seed p`. Build an `n × n` grid where cell `(r, c)` is blocked when `rng.randrange(100) < p` (draw the cells row by row, but the start `(0, 0)` and the end `(n-1, n-1)` are never blocked — still draw for them, then force them open). Count the paths from the start to the end moving only right or down through open cells with a recursive function decorated with `functools.cache` — without the

cache the hidden case is exponential. Print `blocked <count of blocked cells>` and `paths <count>`.

Input: `n seed p`. **Output:** two lines.

```
4 5 30
```

prints

```
blocked 3
paths 9
```

INPUT 1

```
4 5 30
```

OUTPUT 1

```
blocked 3
paths 9
```

REFERENCE SOLUTION

```python
import random
import sys
from functools import cache

sys.setrecursionlimit(10000)
n, seed, p = map(int, input().split())
rng = random.Random(seed)
blocked = [[rng.randrange(100) < p for _ in range(n)] for _ in range(n)]
blocked[0][0] = blocked[n - 1][n - 1] = False


@cache
def paths(r, c):
    if r >= n or c >= n or blocked[r][c]:
        return 0
    if r == n - 1 and c == n - 1:
        return 1
    return paths(r + 1, c) + paths(r, c + 1)


print("blocked", sum(map(sum, blocked)))
print("paths", paths(0, 0))
```

EXERCISE 2

Prefix sums for range queries

Read `n q seed`. Generate `xs = [rng.randrange(-1000, 1000) for _ in range(n)]`, then `q` queries each drawn as `i = rng.randrange(n)` followed by `j = rng.randrange(i, n)` (inclusive range `i..j`). Answer every query in O(1) with a prefix-sum array built once — a fresh `sum(xs[i:j + 1])` per query is O(n·q) and will not finish the hidden case. Print `queries <q>`, `sum of answers <total>`, `max answer <largest>` and `min answer <smallest>`.

Input: `n q seed`. **Output:** four lines.

```
6 3 1
```

prints

```
queries 3
sum of answers 2278
max answer 1207
min answer 336
```

INPUT 1

```
6 3 1
```

OUTPUT 1

```
queries 3
sum of answers 2278
max answer 1207
min answer 336
```

REFERENCE SOLUTION

```python
import random

n, q, seed = map(int, input().split())
rng = random.Random(seed)
xs = [rng.randrange(-1000, 1000) for _ in range(n)]
queries = []
for _ in range(q):
    i = rng.randrange(n)
    j = rng.randrange(i, n)
    queries.append((i, j))

prefix = [0]
for x in xs:
    prefix.append(prefix[-1] + x)

answers = [prefix[j + 1] - prefix[i] for i, j in queries]
print("queries", q)
print("sum of answers", sum(answers))
print("max answer", max(answers))
print("min answer", min(answers))
```

QUIZ

1. What is the first question when a program is too slow?

A. Which loop to unroll

B. Is the algorithm right — what is the operation count as a function of n?

C. Which C extension to use

D. Whether to disable the GC

2. What does a prefix-sum array buy?

A. Faster sorting

B. Any range sum in O(1) after one O(n) pass

C. Less memory

D. Exact floats

3. How should a hundred thousand integers be read from standard input?

A. `input()` in a loop

B. `sys.stdin.buffer.read().split()` once, then `map(int, ...)`

C. `readline` per number

D. `eval(input())`

4. Which micro-optimisation belongs only in a profiled hot loop?

A. Using a set for membership

B. Binding `append = out.append` before the loop

C. Using `join` for strings

D. Choosing the right algorithm

5. When should optimisation stop?

A. Never

B. When the target number is met — every further change costs readability

C. When the code has no loops

D. At 100 % coverage

ANSWER KEY · WRITING FAST PYTHON — THE CHECKLIST, FROM ALGORITHM TO MICRO-OPTIMISATION

1. B Each level of the checklist dwarfs the next; a quadratic algorithm is slow in any language.

2. B `prefix[j + 1] - prefix[i]`.

3. B One read and one split; per-line calls each cost a system call and a decode.

4. B Hoisting lookups is worth 10–40 % on the loop that matters and nothing elsewhere; the others are always right.

5. B The readable version is what gets maintained.

CHECKPOINT · 25 MIN

Checkpoint — Memory, performance and the interpreter

This checkpoint covers the whole module: the object model with reference counting, the cycle collector, weak references and the memory levers; the cost model of the built-in operations and the loop shapes that hide a quadratic; bytecode, code objects, the four kinds of name load and the 3.11 specialising interpreter; measuring with `perf_counter`, `timeit`, `cProfile` and `tracemalloc` under proper hygiene; numeric performance with `array`, `bytes` and NumPy vectorisation; and the checklist for writing fast Python from the algorithm down to the hot loop.

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 `del x` do to the object `x` referred to, and when is an object actually freed?
- Why can reference counting not free two objects that refer to each other?
- What is the cost of `x in lst` versus `x in s`, and of `lst.pop(0)` versus `deque.popleft()`?
- Why is `LOAD_FAST` cheaper than `LOAD_GLOBAL`?
- Which statistic does `timeit` report, and why the minimum?
- What is the difference between `tottime` and `cumtime` in a profile?
- What does vectorisation remove from a numeric loop?

The three programs are a top-k selection over a large generated stream with a heap, an order-preserving de-duplication of a large generated list that only passes with constant-time membership, and a word-frequency tally over generated tokens with a stable tie order — each sized so that the wrong data structure exceeds the time limit.

EXERCISE 1

Top-k with a heap

Read `n k seed`. Generate `xs = [rng.randrange(10**9) for _ in range(n)]` with `rng = random.Random(seed)` and find the `k` largest values with `heapq.nlargest` — O(n log k), not a full sort. Print `top <the k values, largest first, space-separated>` and `threshold <the k-th largest>`.

Input: `n k seed`. **Output:** two lines.

```
10 3 1
```

prints

```
top 909925047 861425548 820096753
threshold 820096753
```

INPUT 1

```
10 3 1
```

OUTPUT 1

```
top 909925047 861425548 820096753
threshold 820096753
```

REFERENCE SOLUTION

```python
import heapq
import random

n, k, seed = map(int, input().split())
rng = random.Random(seed)
xs = [rng.randrange(10**9) for _ in range(n)]

top = heapq.nlargest(k, xs)
print("top", " ".join(map(str, top)))
print("threshold", top[-1])
```

EXERCISE 2

Order-preserving de-duplication

Read `n seed`. Generate `words = [f"w{rng.randrange(20000)}" for _ in range(n)]`. Remove duplicates keeping the first occurrence of each word, in O(n) with a `seen` set — a list membership test per word is O(n·d) and will not finish the hidden case. Print `distinct <count>`, `first <the first five distinct words>` and `last <the last five distinct words>` (fewer when there are fewer).

Input: `n seed`. **Output:** three lines.

```
12 1
```

prints

```
distinct 12
first w4402 w18651 w2067 w8358 w3863
last w15474 w12439 w6879 w3075 w15986
```

INPUT 1

```
12 1
```

OUTPUT 1

```
distinct 12
first w4402 w18651 w2067 w8358 w3863
last w15474 w12439 w6879 w3075 w15986
```

REFERENCE SOLUTION

```python
import random

n, seed = map(int, input().split())
rng = random.Random(seed)
words = [f"w{rng.randrange(20000)}" for _ in range(n)]


def dedupe(items):
    seen = set()
    result = []
    for item in items:
        if item not in seen:
            seen.add(item)
            result.append(item)
    return result


distinct = dedupe(words)
print("distinct", len(distinct))
print("first", " ".join(distinct[:5]))
print("last", " ".join(distinct[-5:]))
```

EXERCISE 3

Frequencies with stable ties

Read `n seed`. Generate `tokens = [f"t{rng.randrange(50)}" for _ in range(n)]`. Count them with `collections.Counter` and print the five most frequent (fewer if there are fewer distinct) as `<token> <count>`, ordered by count descending and then by token ascending — `most_common`

alone does not define the tie order, so sort explicitly. Finish with `distinct <number of distinct tokens>`.

Input: `n seed`. **Output:** five lines, then the count.

```
30 1
```

prints

```
t48 3
t24 2
t28 2
t31 2
t6 2
distinct 24
```

INPUT 1

```
30 1
```

OUTPUT 1

```
t48 3
t24 2
t28 2
t31 2
t6 2
distinct 24
```

REFERENCE SOLUTION

```
import random
from collections import Counter

n, seed = map(int, input().split())
rng = random.Random(seed)
tokens = [f"t{rng.randrange(50)}" for _ in range(n)]

counts = Counter(tokens)
for token, count in sorted(counts.items(), key=lambda item: (-item[1], item[0]))[:5]
    print(token, count)
print("distinct", len(counts))
```

CHECKPOINT QUESTIONS

1. When is an object freed by reference counting?

A. At the next `gc.collect()`

B. The moment its last reference goes

C. At function return only

D. At interpreter exit

2. Which pair of objects needs the cycle collector?

A. Two ints

B. A list and a dict that reference each other

C. A string and its slice

D. Two tuples of ints

3. `x in xs` for a list of a million items inside a loop of a million queries costs about…

A. 10^6 operations

B. 10^{12} operations — a set makes it 10^6

C. 10^9 operations

D. Nothing

4. Which is the O(1) queue in the standard library?

A. `list` with `pop(0)`

B. `collections.deque` with `popleft()`

C. `heapq`

D. `tuple`

5. Why is `LOAD_GLOBAL` slower than `LOAD_FAST`?

A. It is not

B. It is a dictionary lookup (globals then builtins) rather than an indexed slot read

C. It allocates

D. It calls a function

6. What did the 3.11 specialising interpreter change for code you write?

A. Nothing observable

B. Stable types in hot loops and `try/except` for rare cases are rewarded; no source changes needed

C. Loops must be annotated

D. Globals are free

7. What is the right tool to find which function a slow program spends its time in?

A. `timeit`

B. `cProfile` sorted by `tottime`

C. `time.time()` around `main`

D. `dis`

8. Why measure the second run rather than the first?

A. The first is faster

B. The first pays for imports, compilation, caches and the specialiser warming up

C. There is no difference

D. The clock is wrong at first

9. What is the memory of a Python list of a million floats, roughly?

A. 8 MB

B. 8 MB of pointers plus 24 MB of float objects

C. 4 MB

D. 1 MB

10. Which replaces a per-element Python loop over a NumPy array?

A. A `while` loop

B. Ufuncs, boolean masks and reductions — one C loop over the array

C. `map` with a lambda

D. Recursion

11. What does `@functools.cache` do for a recursive function with overlapping subproblems?

A. Nothing

B. Each distinct argument tuple is computed once — exponential becomes polynomial

C. Makes it iterative

D. Reduces recursion depth

12. In what order should performance fixes be attempted?

A. Micro-optimisations first

B. Algorithm, data structure, work in C, I/O, hot loop, escape hatches

C. C extension first

D. Random order

ANSWER KEY · CHECKPOINT — MEMORY, PERFORMANCE AND THE INTERPRETER

1. B Immediate and predictable; only cycles wait for the collector.

2. B Their counts never reach zero on their own; atoms are not even tracked.

3. B Linear membership inside a linear loop.

4. B A list shifts every element on a front pop.

5. B Hence binding a global to a local before a hot loop.

6. B Specialised instructions and zero-cost exceptions.

7. B `timeit` compares candidates; the profiler attributes.

8. B Warm-up is part of benchmarking hygiene.

9. B A NumPy float64 array of the same values is 8 MB total.

10. B Vectorisation is the whole benefit; an element loop throws it away.

11. B The recursion depth is unchanged; `sys.setrecursionlimit` may still be needed.

12. B Each level dwarfs the next; stop when the target is met.