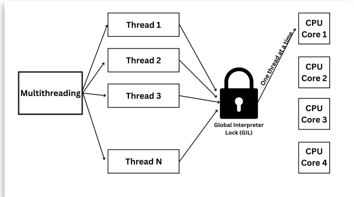
Part IV

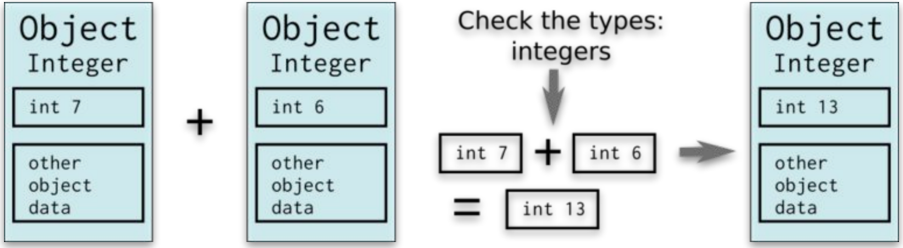
Why Python is considered slow?

Why Python is considered slow?

- The theoretical complexity in Python is similar to other languages:
  - Summing list: $O(n)$
  - Nested loops: $O(n^2)$
  - Naive matrix multiplication: $O(n^3)$
- BUT: Actual per-operation cost is much higher for Python!
  - Python introduces several overheads
  - A simple operation, such as “add”, includes multiple
    - Can be 10^2–$10^3\times$ slower than C, depending on the operation



Source: codeacademy
More info: Link1, Link2



Source: futurelearn

Let’s try it out!

Python is an Interpreted Language

- Every line and every operation (e.g. sum += x) is executed by the CPython interpreter, not by native machine code -> overhead
- For each iteration Python has to repeatedly:
    - Parse bytecode instructions
    - Perform type checks
    - Allocate memory
    - Manage references
    - Call C functions internally
- **C**, in comparison, translates the loop directly into CPU instructions during the compilation

Python bytecode call stack

```
↳ ceval.c: case NB_ADD
  ↳ PyNumber_Add(a, b)
    ↳ type(a)->tp_as_number->nb_add(a, b)
      ↳ int_add() / float_add() / custom __add__()
        ↳ Allocate new PyObject
        ↳ Refcount increments/decrements
```

Python executes *hundreds* of instructions to do what C does in three!

Function Pointer Dispatch

- Almost every operation eventually turns into a function pointer call at **C** level
- This creates a significant overhead compared to compiled languages
- Examples:
  - a + b: nb_add (*PyNumberMethods*)
  - a[b]: mp_subscript (*PyMappingMethods*)
  - len(a): sq_length (*PySequenceMethods*)
  - methods, e.g. a.append(), found via the type method table (*tp_methods*)
- Especially for complex/self-defined objects, the look-up can be costly

Look up the type of a => Fetch the correct function pointer (e.g. a->ob_type->tp_as_number->nb_add) => Call that function

Dynamic Typing

- Python determines the variable types at runtime
  - +: We do not need to care, very beginner-friendly
  - -: Performance…
- In C, the type of each variable is known at compile time
- Python allows you to crazily define whatever you want
- Can we avoid it? *No, unfortunately not in plain Python*
- Best way to mitigate: Use **NumPy**!

```
def add_2(x):
    return x+2
x=1
print(add_2(x))
type(x)   # <class 'int'>
```

```
import numpy as np
```

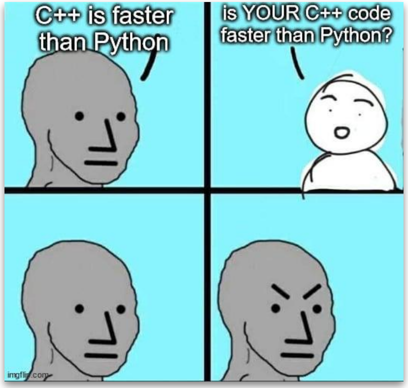
```
arr = np.arange(1000000,dtype=np.float32)
result = arr * 2.5
```

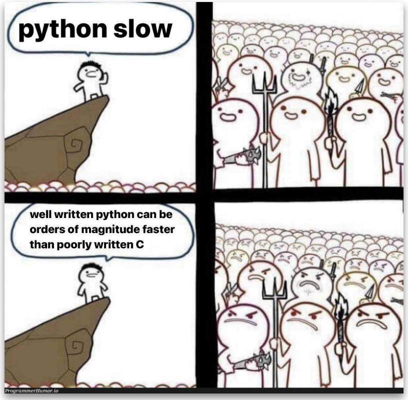
- Note: **Type hints** do NOT fix the types, but are helpful anyway!

Is it then a problem?

- Of course, but **only** for time and performance critical applications!
    - **Easy solution**: Don’t use it, if not applicable
- It can be **mitigated** with good practices and some tricks
    - **Avoid loops**, use **NumPy** and **vectorization** when possible
    - Directly use C implementation for performance-critical parts

Should I care?
My recommendation is: start right directly from the beginning! Even if you are expectedly not performance bound for easy computations, such as plotting, it is good to maintain a good coding style and in the end a good practice to get used to the standard optimizations.





Should I use Python for my Projects?

- This of course depends on many factors!
- What is the Nature of Your Project?
  - A large simulation framework for gravitational waves -> rather not
  - Time or performance critical? -> rather not, but could work
  - Data science and ML -> highly depends on the problem size
  - Data analysis and visualization of my measurements -> definitely!
  - Task automation -> Yes
- Can I make use of existing tools?
- How much time do I have?

In most scientific fields, Python is close to always a good choice! However, the more specific the task is, the more you may gain from a more specialized solution.

Summary: Performance Potholes

Interpreter loop overhead	A giant `switch` loop in `ceval.c` fetches and executes each bytecode, so every simple step involves opcode decoding and dispatch.
Dynamic type deduction	Every operation must inspect the runtime type (`a->ob_type`) to decide which addition function (`int`, `float`, custom `__add__`) to call.
Function pointer dispatch	Instead of a fixed machine instruction, Python calls a C function via a pointer from the object’s type table (`nb_add`), adding an indirect branch.
Object allocation	Each new result (even `1+1`) requires creating a new heap object with metadata, not just writing to a register.
Reference counting	CPython increments and decrements object reference counters on every assignment and temporary value, constantly updating bookkeeping.

This is not nice, but keep in mind that most of the limitations are compromises that simplify the language a lot!

Why so negativ?

- To make the best use Python, it is most important to know how **NOT** to use it!
- Philosophy: Better know your enemy
- If used **correctly**, many disadvantages can be **mitigated**
- **This is in my opinion the better way (instead of only learning the good things) and provides a good basis for better code quality and performance**