Efficient Python Programming

Dr Vassil Vassilev, Princeton/CERN
Tutor: Aaron Jomy, CERN

Prerequisites

- Be interactive, ask questions!
- Please share your own experience!
- Have a laptop
- Install the software required: https://compiler-research.org/tutorials/efficient-python-programming/
- Should you have any problems with the installation find Aaron during the break

Physical Warmup

- How many of you have used Python before?
  - How many of you have a project > 1000 lines of code?
  - > 5000 lines?
  - > 15000 lines?
  - How many of you worked in a team of 2 on such project?
    - > 5
    - > 10
- How many of you coded in a different language than Python?
  - In a low-level language such as C++, C, or assembly?
- What’s your average data set? MB, GB, TB, PB?

Goals

- Understand the general principles behind high-performance programming
- Recognize and explore performance optimization opportunities
- Build intuition about computer program execution
- Practice

Just Enough Performance

```
def f(N = 100, M = 1000, L = 10000):
    for i in range(N):
        for j in range(M):
            for k in range(L):
                g(i, j, k)
```

What Is Python?

- Delivers just enough performance when relying on bare-metal technologies
- NumPy is an enabler for an entire data science ecosystem
- NumPy is very good but sometimes far from bare metal, accelerators and across nodes (means to address the problem such as CuPy or Dask)
- Lets learn what else could be done

“This is why I love C and use Python for most of the work I do...”, a happy user on the internet

Outline

What is my problem? Can I classify it?	Problem	~10x-1000x
What is the step-by-step solution to my problem?	Design, Algorithm, Data	
What is the step-by-step solution in code?	Source Code	~2x-10x
What other tools and technologies I can use?	Compilers, libraries	~10%-20% (w/o vectorization)
What's my hardware? How much resources I have?	System Architecture	~5%-20%
What are the commands that my chip understands?	Instruction Set	
What are the implementation details of my chip?	Microarchitecture	~10%-30%
Which commands are implemented with transistors?	Integral Schemes	

Outline

Problem	~10x-1000x
Design, Algorithm, Data	
Source Code	~2x-10x
Compilers, libraries	~10%-20% (w/o vectorization)
System Architecture	~5%-20%
Instruction Set	
Microarchitecture	~10%-30%
Integral Schemes	

If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about the solutions

- *Albert Einstein*

Problem Defining

- Is a problem worth solving?
- What would be the impact of having a solution?
- Is the problem new to the domain?
- Does the problem exist anywhere else? How it is solved in other domains?
- What domain knowledge it requires and can it define away some issues with existing solutions?
- Can I decompose the it
  - Pontryagin's suboptimality principle
  - Morphology analysis

Solution Designing

- What’s the mathematical foundation of my solution?
- What are the implementation requirements?
  - Do I need 3 years to implement it?
  - What skills?
  - How many people?

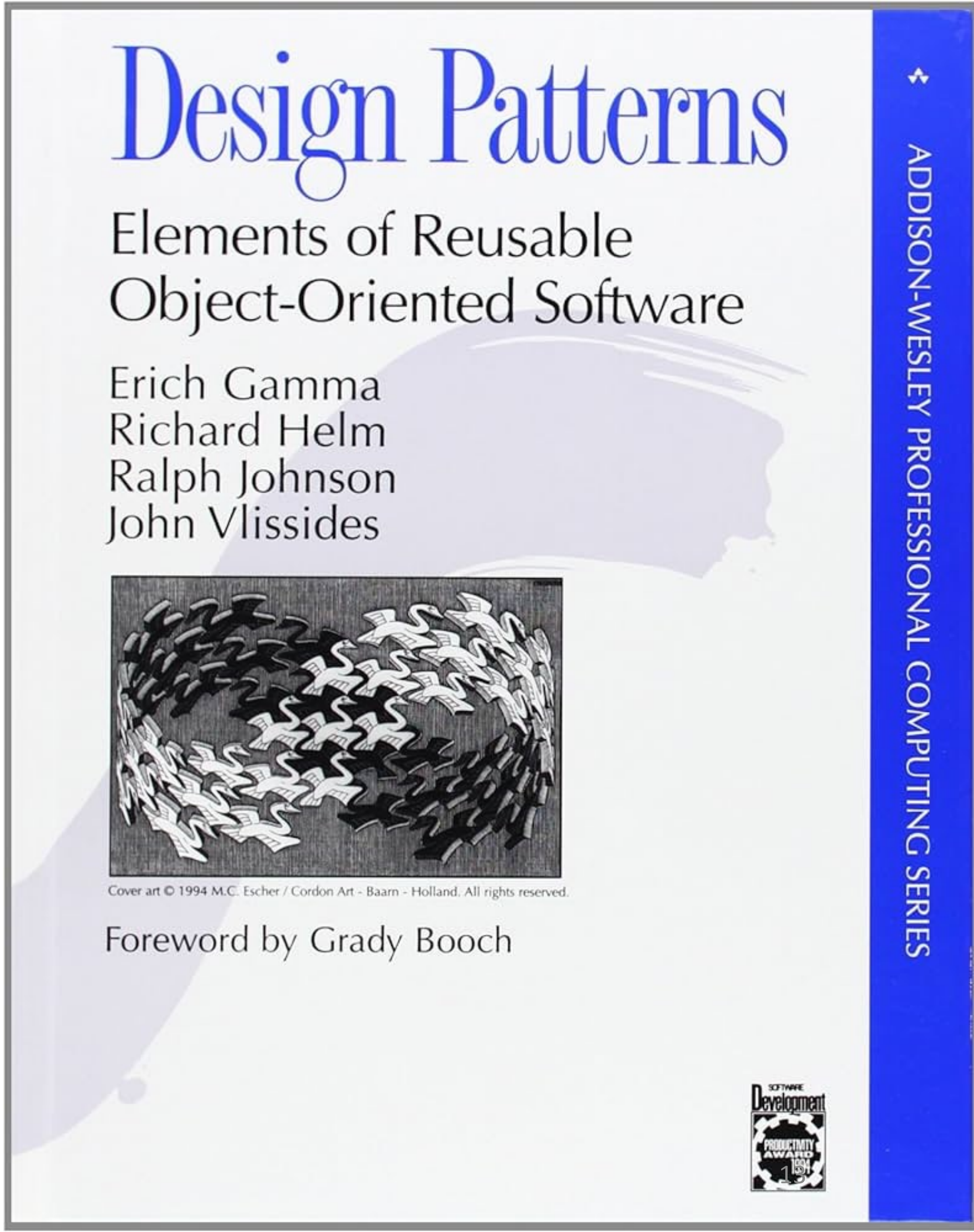
Solution Designing

- How do I translate the solution into code?
  - What's my input data?
  - What algorithms I should use? What's their complexity?
  - What are my data structures?
  - How to scale?

Debugging is 3 times more difficult than coding the algorithm.
What happens if coded a solution at the limit of our skills?

Design (Anti-)Patterns

- In software engineering, a design pattern describes a relatively small, well-defined aspect (i.e. functionality) of a computer program in terms of how to write the code.
- Using a pattern is intended to leverage an existing concept rather than re-inventing it. This can decrease the time to develop software and increase the quality of the resulting program.

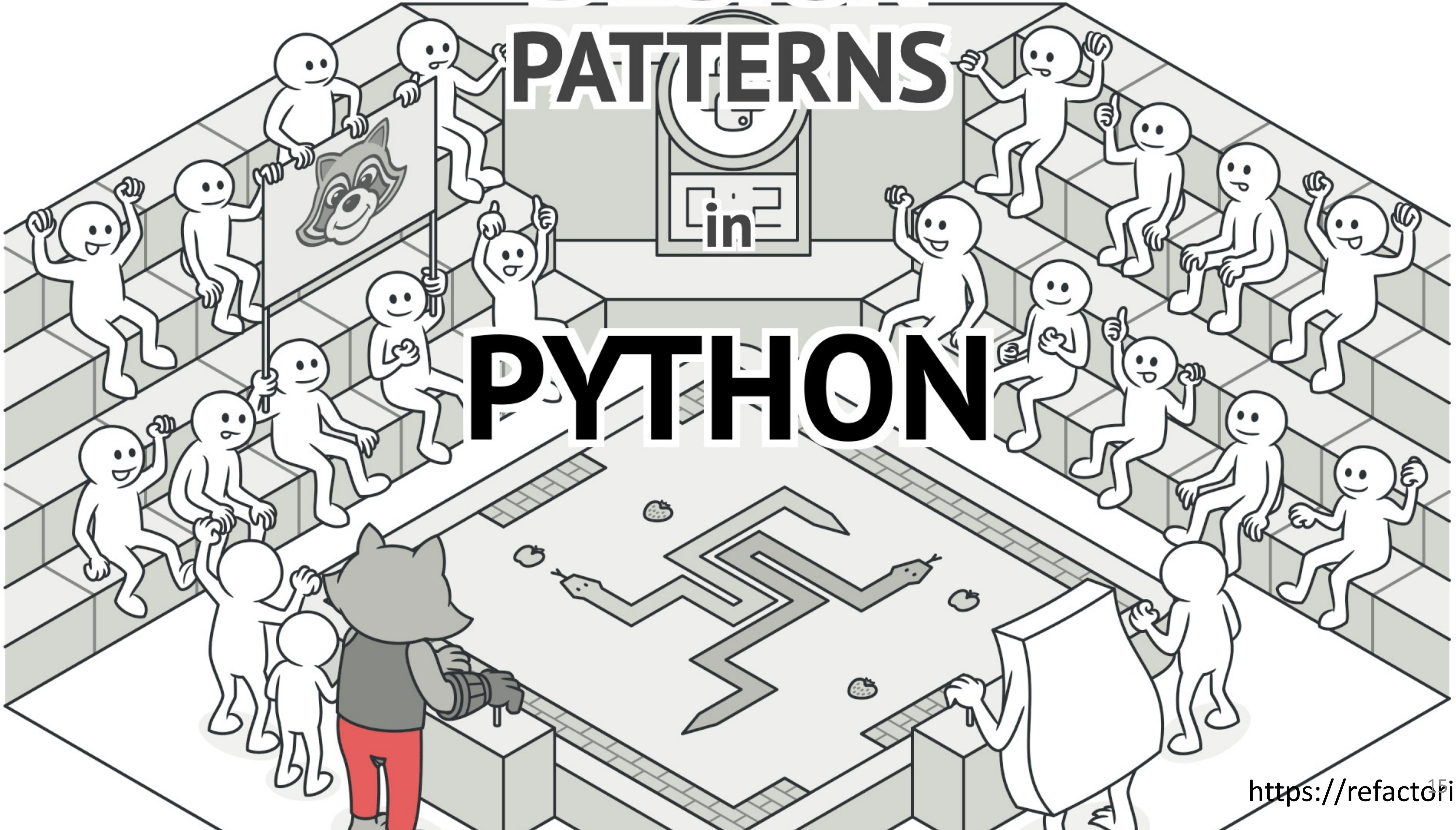


Design Pattern Description

Each pattern is classified as either creational, structural or behavioral and described by a section on:

- Intent – What is the overall idea of the pattern
- Motivation – What is the problem that's being solved
- Applicability – What are the common scenarios to use the pattern
- Structure – A high-level UML description of the entity relationships of the pattern
- Participants – What is the responsibility of the UML entities
- Collaborations – What's the impact on the client
- Consequences – What are the new trade-offs
- Implementation – High-level description of the implementation ideas
- Sample Code – How the implementation can be used in code illustratively
- Known Uses – What are the uses in well-known software
- Related Patterns – What are the other competing patterns and how they can be combined

DESIGN PATTERNS in PYTHON



https://refactoring.guru/

Algorithm

What makes one program better than another?

- ***<u>Correctness</u>***
- Speed
- Resources it takes
- What else?

Algorithm Analysis. Example

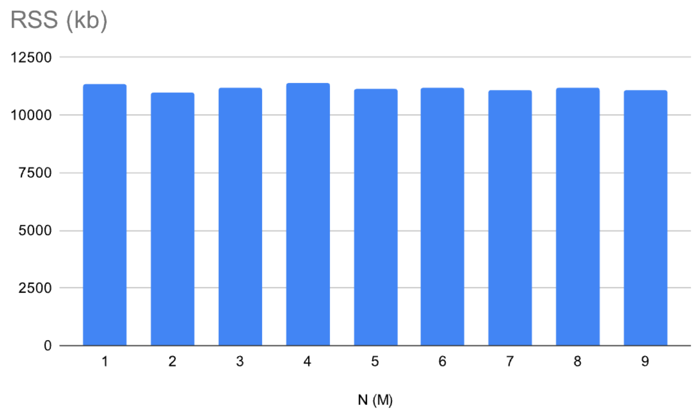
Find the sum of the first N numbers.

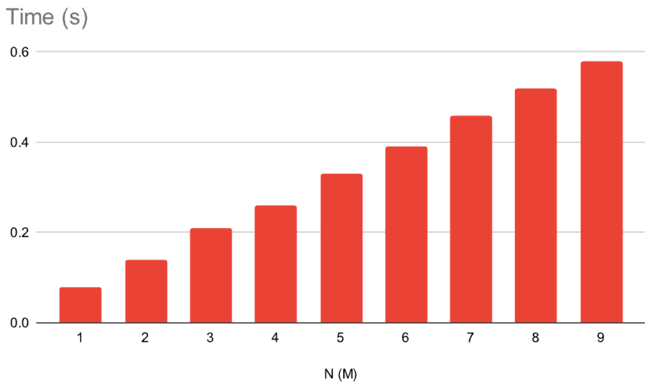
$$\sum_{i=0}^{n} i$$

```python
def sum_to_n(n):
    total = 0
    for i in range(n + 1):
        total += i
    return total
```

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

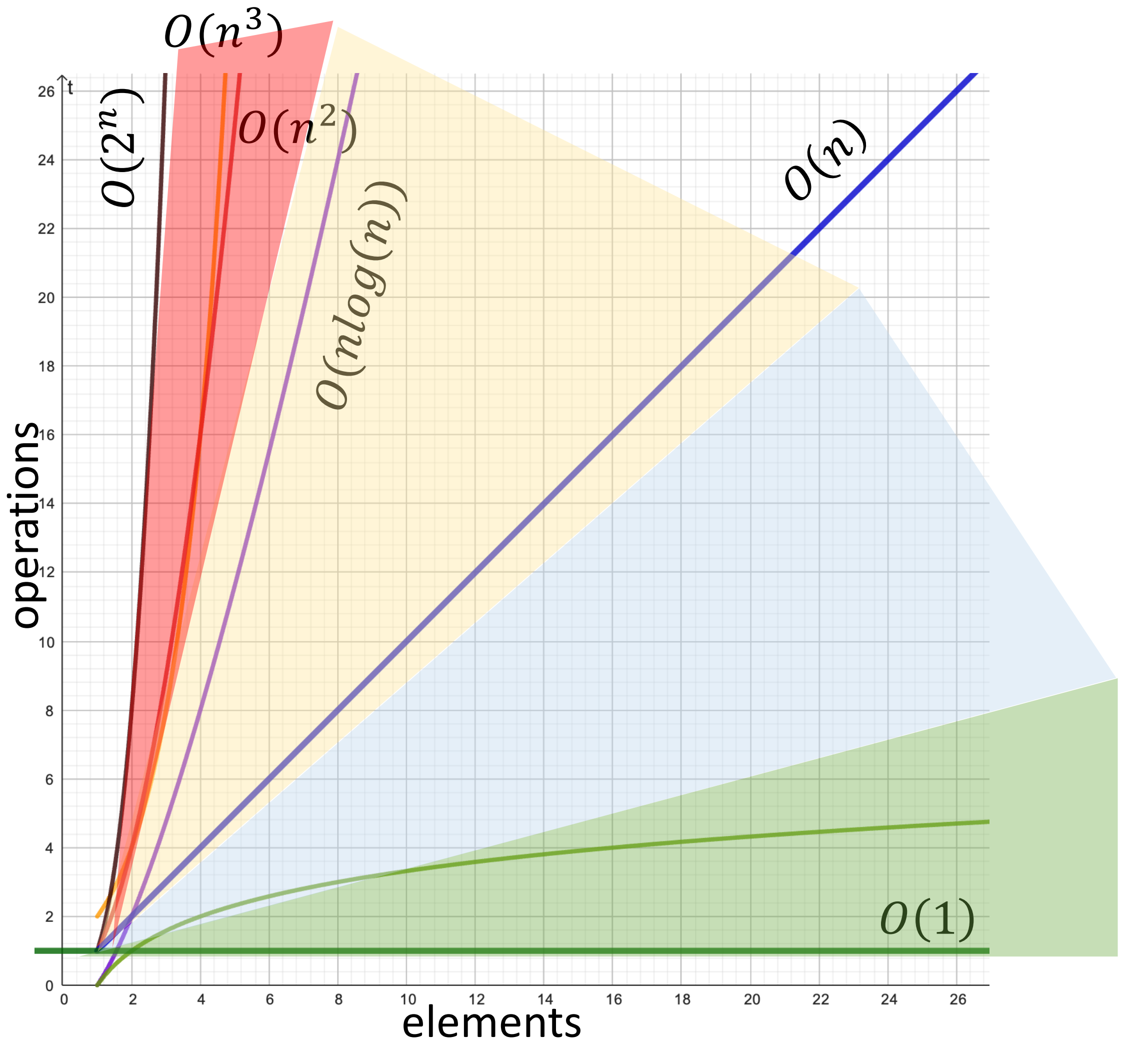
N	Time (s)	RSS (kb)
1M	0.08	11320
2M	0.14	10952
3M	0.21	11172
4M	0.26	11416
5M	0.33	11156
6M	0.39	11180
7M	0.46	11056
8M	0.52	11188
9M	0.58	11088





The Big Picture With Big O

Worst-case algorithm complexity analysis.



Big O is an asymptotic notation which is used limited behavior of function.

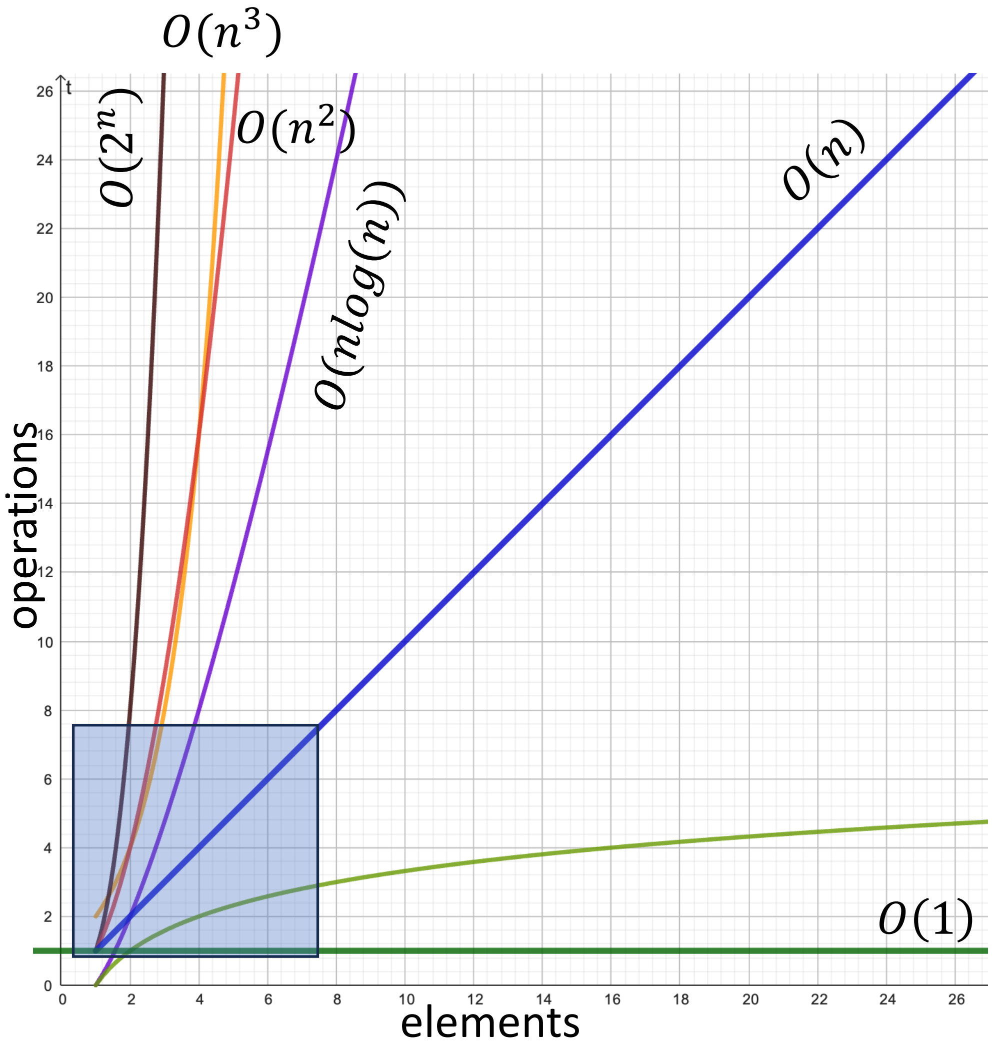
In CS is used to classify algorithms according to how their run time or space requirements grow as the input size grows.

A very important tool in the programmer's toolbox.

Handout time

The Big Picture With Big O

Worst-case algorithm complexity analysis. O



f(n)	Name	Example
1	Constant	
logn	Logarithmic	
n	Linear	
nlogn	Log Linear	
n^2	Quadratic	
n^3	Cubic	
2^n	Exponential	Halting problem/Generalized Chess, Go
n!	Factorial	Brute forcing travel salesman

Big O. Limitations.

Amortized O(1)

```
list = [1, ..., 63]
list.append(64) # Would that be O(1)?
```

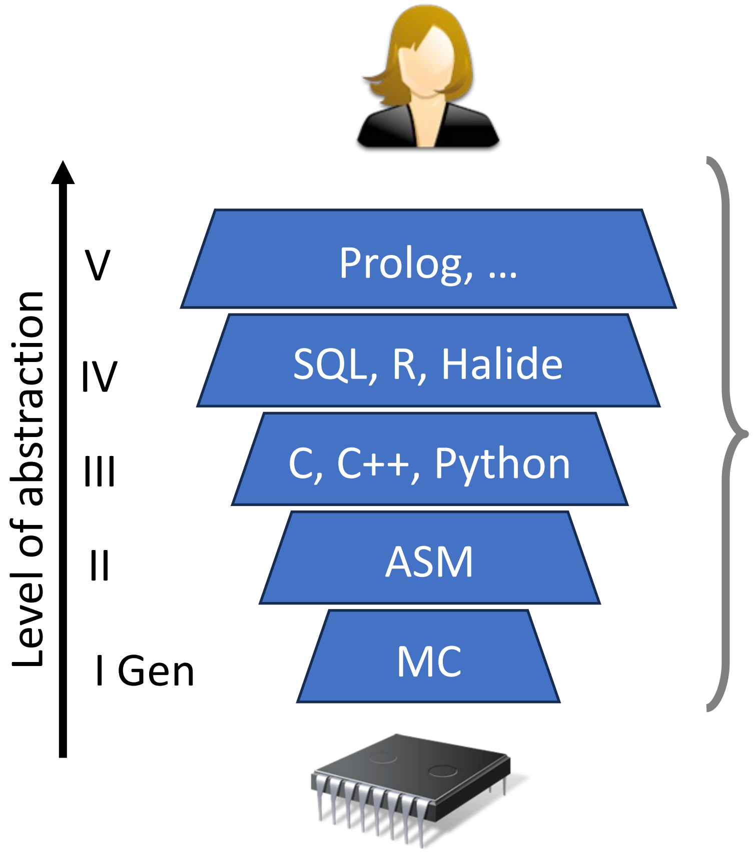
- $O(M*N^2)$ can matter
- Worst-case scenarios happen rarely
- Depending on the properties of the input data the algorithmic complexity can vary

Programming is the art of replacing old bugs with new ones.

- *My personal experience*

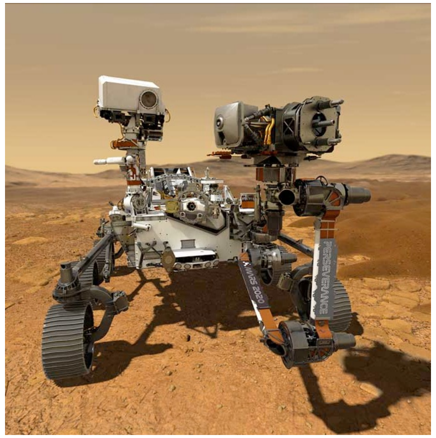
Brief History of Programming Languages

- **1957** Fortran — first compilers (arithmetic expressions, statements, procedures)
- **1960** Algol — first formal definition of PL (BNF grammars, block structure, recursion)
- **1970** Pascal — user-defined types, virtual machines
- **1972** C — structured programming, static type system
- **1985** C++ — object-oriented, exceptions, templates
- **1991** Python — duck typing, ease of use



- ****Important steps in imperative PL***

Language Design Principles



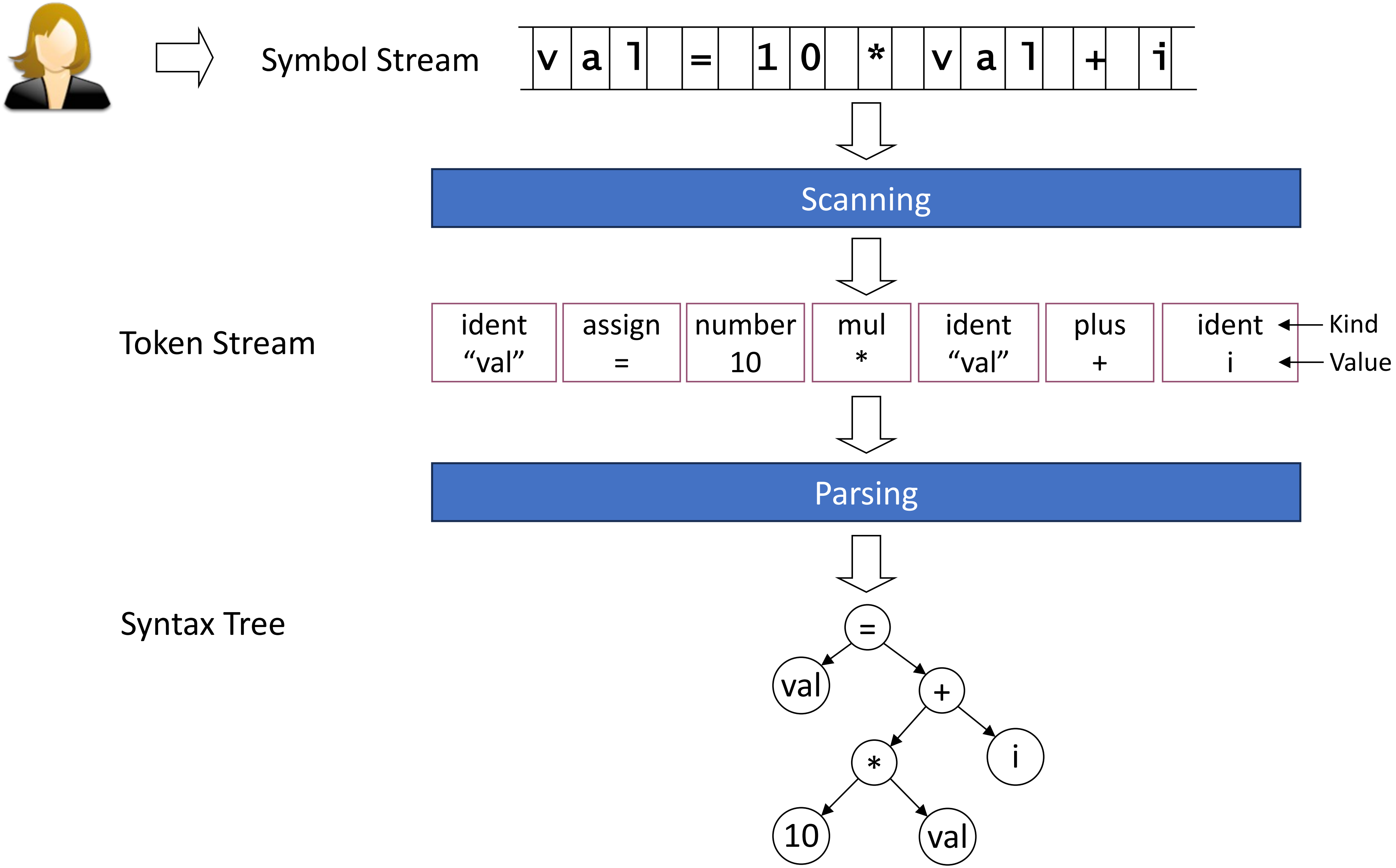
C++
- Efficiency
- Stability
- Backward compatibility

“Prioritizes Performance over Surprise which is sometimes surprising“ T. Winters [Link]

Python
- Readability
- Simplicity
- Flexibility

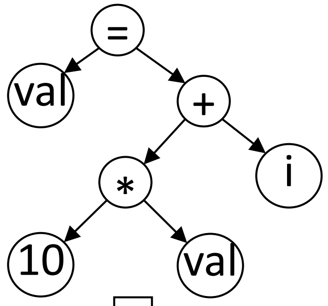
“Special cases aren't special enough to break the rules” Zen of Python [Link]

Translating Programming Languages



Translating Programming Languages

Syntax Tree



Semantic Analysis

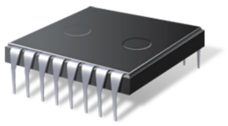
Intermediate representation

Syntax tree, symbol table, ...

Optimization

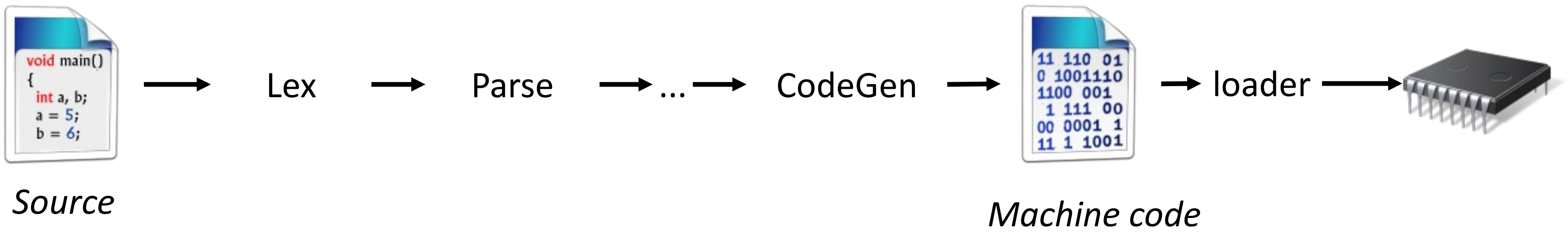
Code Generation

```
push rbp
mov rbp, rsp
mov qword ptr [rbp - 8], rdi
mov dword ptr [rbp - 12], esi
mov rax, qword ptr [rbp - 8]
imul ecx, dword ptr [rax], 10
...
ret
```

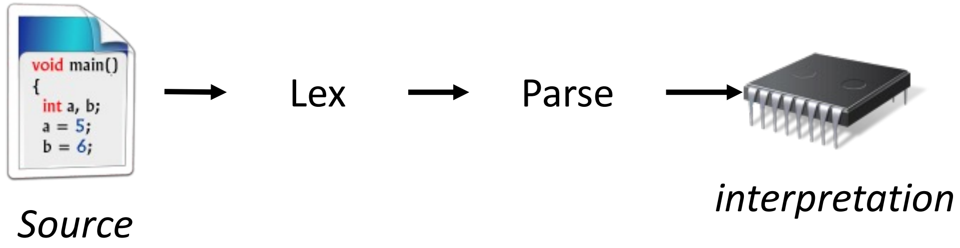


Translator Classification

Compiler translates to machine code

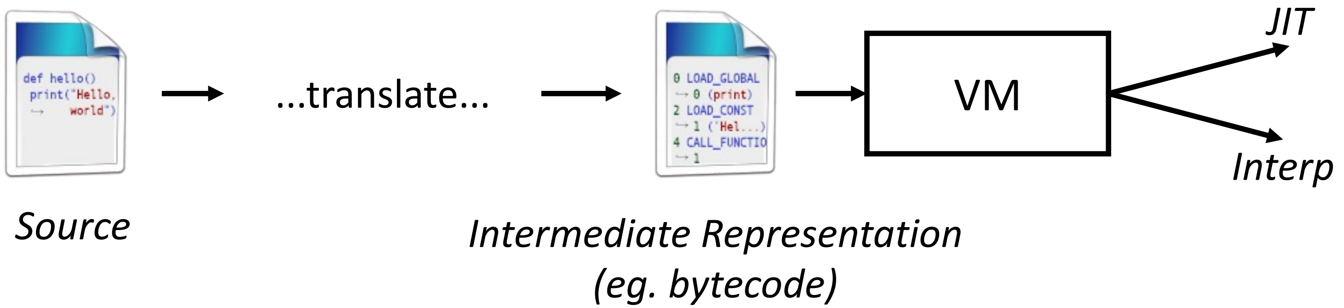


Interpreter executes the source "directly"



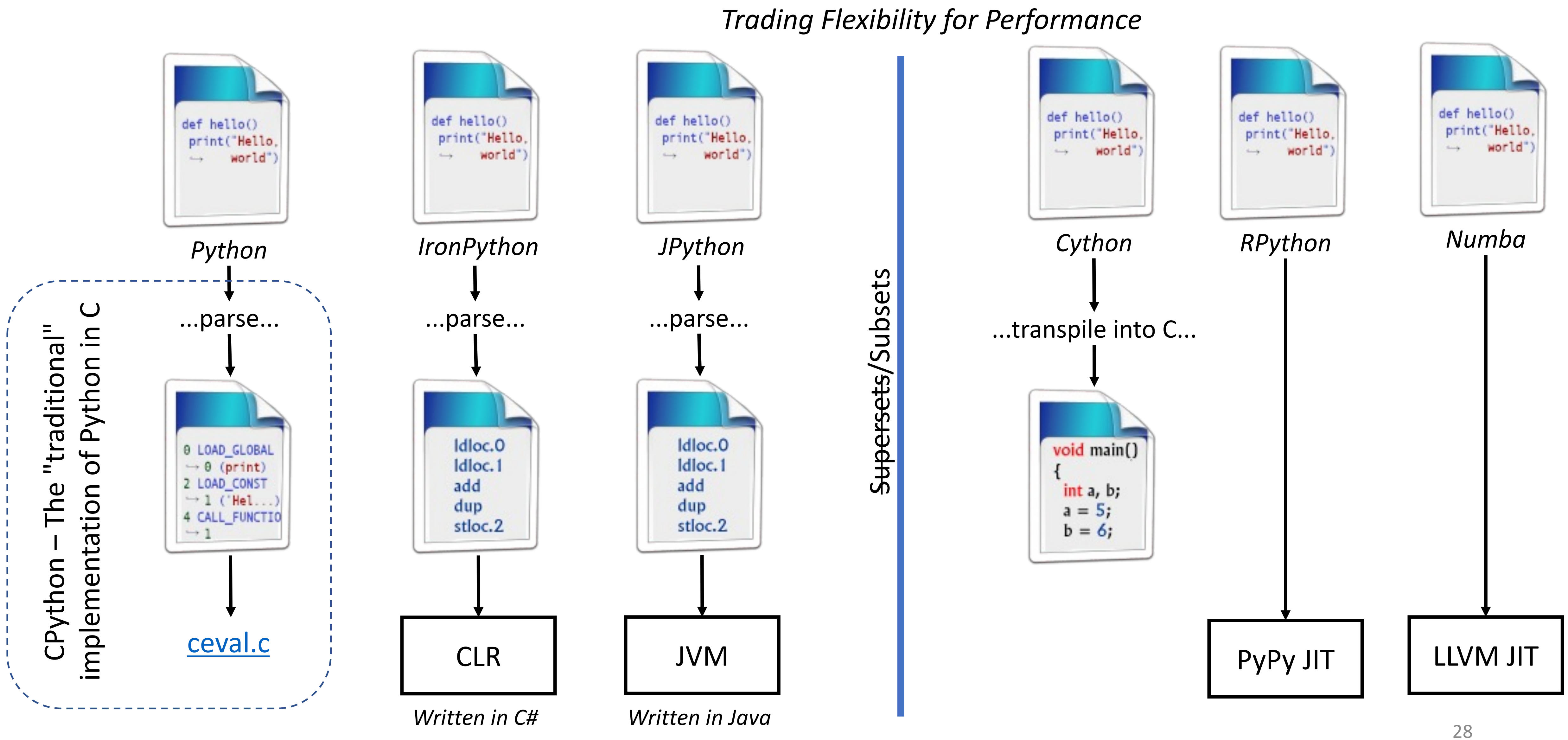
- The operators in loops are analyzed again and again

Hybrid compiler interprets intermediate representation

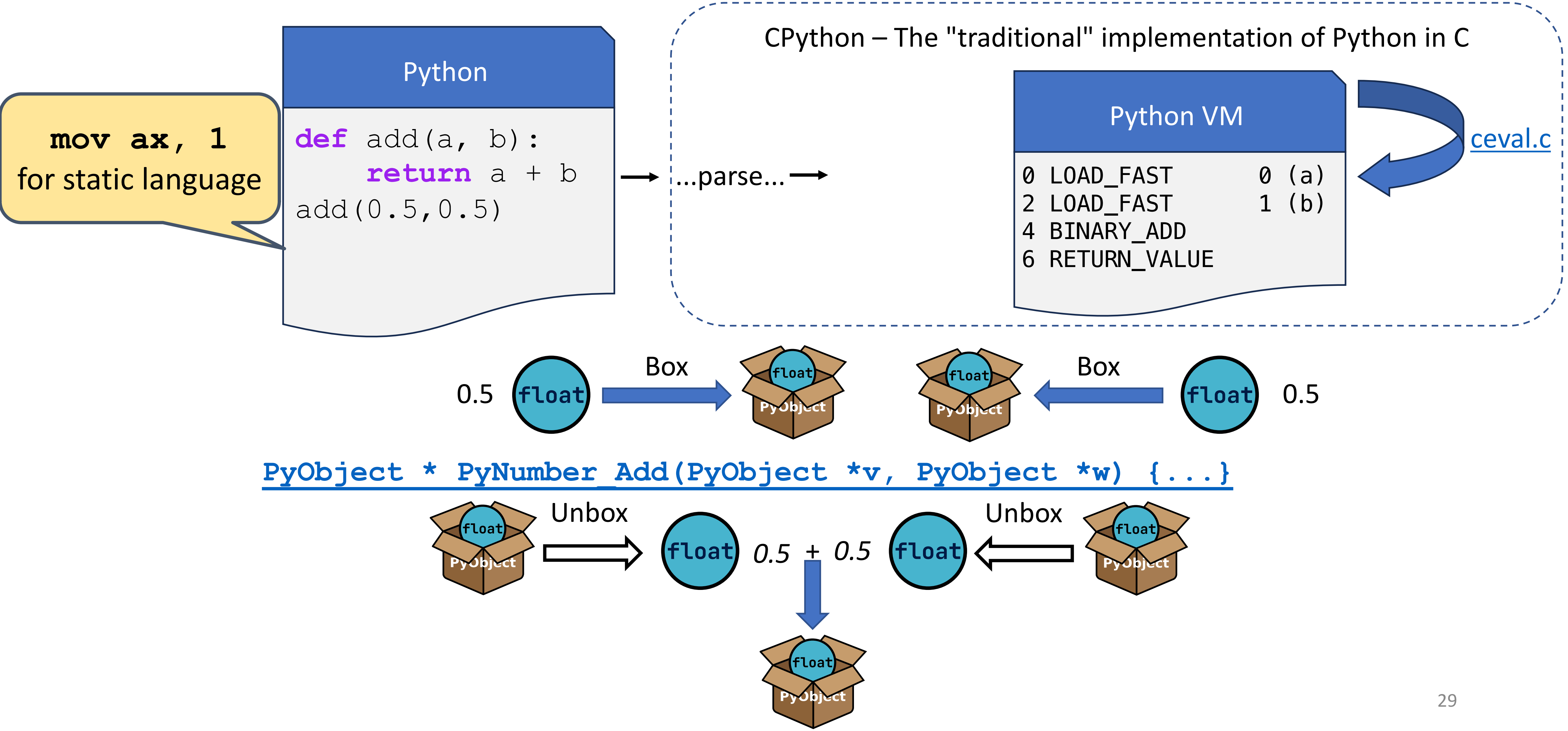


- The source translates to code for virtual machine (VM)
- The VM runtime interprets the code simulating real machine
- The Just-In-Time compiler translates the representation at startup time.

Python Jungle



Trading Performance For Flexibility



“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; *premature optimization is the root of all evil (or at least most of it) in programming*.”

- Donald Knuth

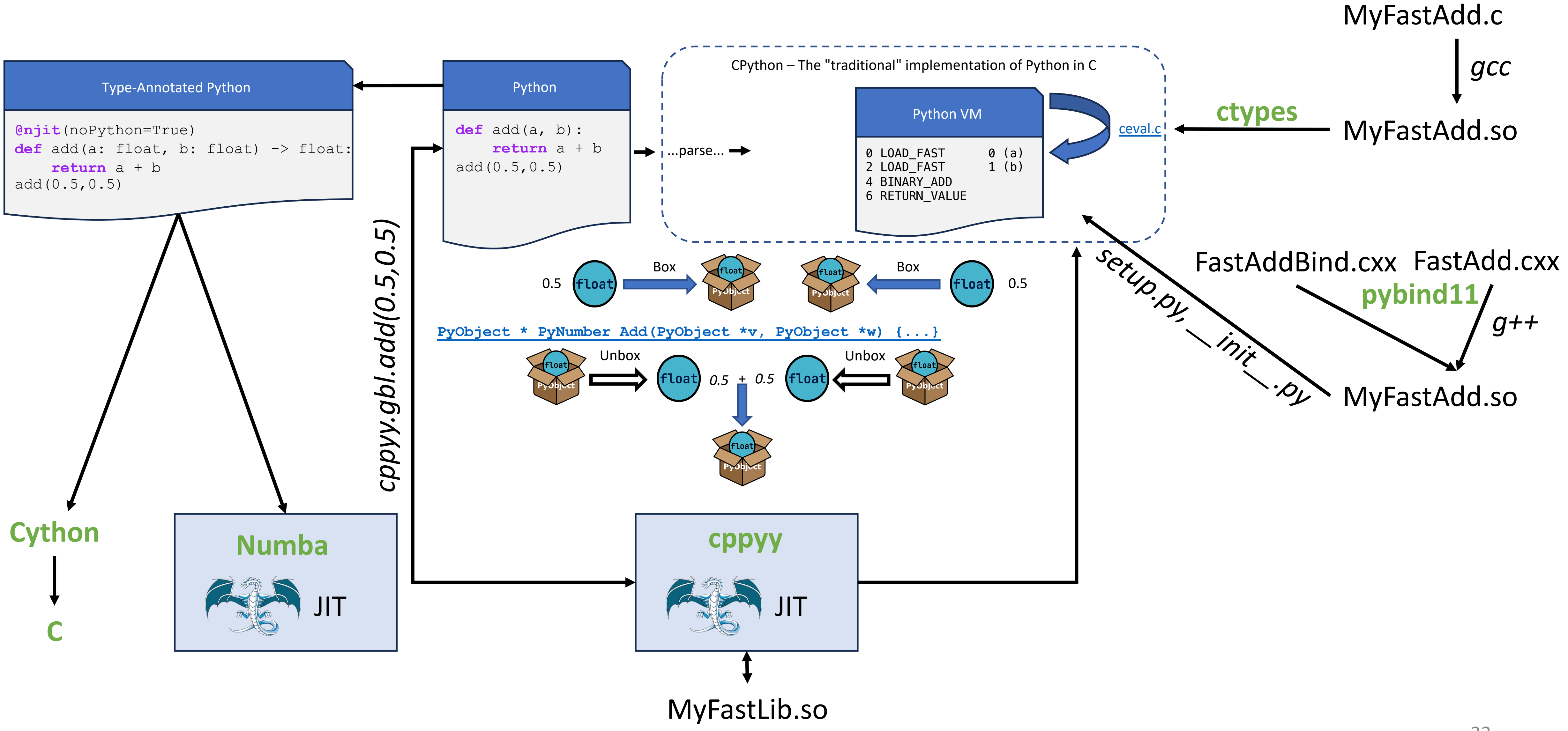
Expressing Optimization Assumptions

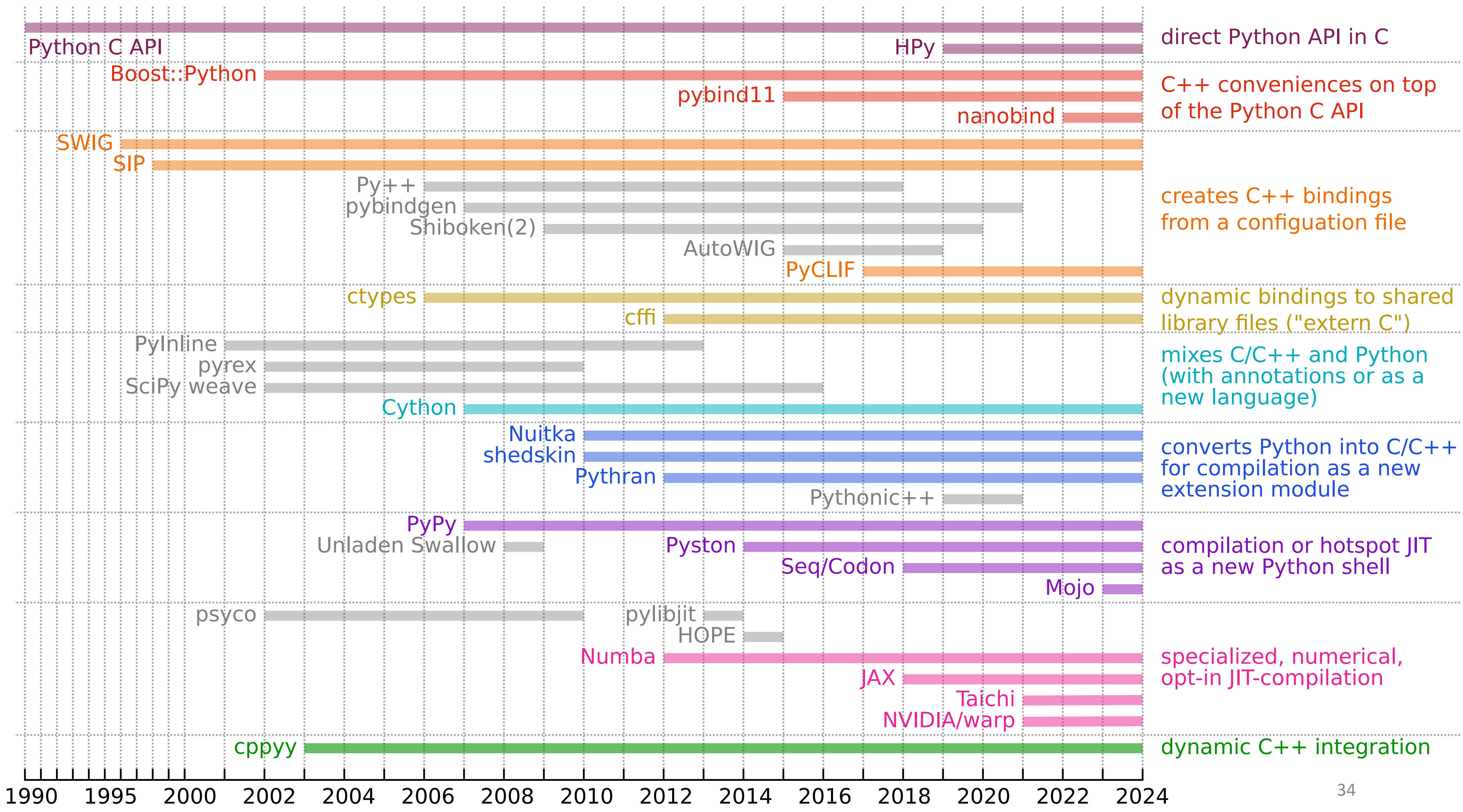
- Use built-in functions and libraries
  They are heavily optimized and implemented in C. Eg. use sum() instead of manually iterating
- Avoid global state
  Global variables are slower to access and hinder optimizations
- Minimize the use of loops
  Use map, filter, reduce from libraries such as NumPy
- Use proper data structures
  Choose the correct data structure for the task. Prefer immutable types such as tuple and frozenset. Prefer pre-allocated, pre-resized types to avoid amortization effects
- Be mindful with string operations such as concatenation

Expressing Optimization Assumptions

- Avoid excessive object creation
  Think about the created objects especially in tight loops
- Avoid memory leaks in resource release
  Use the `with` construct to manage resources such as files or network connections
- Avoid using try/except constructs for control flow
  Exceptions are designed for error handling and less so for other things
- Avoid unnecessary abstractions
  Abstractions introduce often indirection which can be inefficient
- Use declarative style
  List comprehensions and generator expressions allow sometimes better performance.
- Use memoisation techniques to avoid recomputation
  Cache results with `functools.lru_cache`

From Bindings to Full Language InterOp





Python C API
HPy
direct Python API in C
Boost::Python
pybind11
nanobind
C++ conveniences on top of the Python C API
SWIG
SIP
Py++
pybindgen
Shiboken(2)
AutoWIG
PyCLIF
creates C++ bindings from a configuation file
ctypes
cffi
dynamic bindings to shared library files ("extern C")
PyInline
pyrex
SciPy weave
Cython
mixes C/C++ and Python (with annotations or as a new language)
Nuitka
shedskin
Pythran
Pythonic++
converts Python into C/C++ for compilation as a new extension module
PyPy
Unladen Swallow
Pyston
Seq/Codon
Mojo
compilation or hotspot JIT as a new Python shell
psyco
pylibjit
HOPE
Numba
JAX
Taichi
NVIDIA/warp
specialized, numerical, opt-in JIT-compilation
cppyy
dynamic C++ integration
1990 1995 2000 2002 2004 2006 2008 2010 2012 2014 2016 2018 2020 2022 2024

ctypes

```c
// Add.c
int add(int a, int b) {
    return a + b;
}
```

gcc -shared -o libAdd.so Add.c

libAdd.so

Both sides manual by the user

```python
import ctypes

# Load the shared library
lib = ctypes.CDLL('./libAdd.so')

# Specify the argument types and return type
for the C function
lib.add.argtypes = (ctypes.c_int, ctypes.c_int)
lib.add.restype = ctypes.c_int


# Call the function
result = lib.add(3, 4)
```

pybind11

```cpp
// Add.cpp
#include <pybind11/pybind11.h>

int add(int a, int b) {
    return a + b;
}

PYBIND11_MODULE(example, m) {
    m.def("add", &add, "...");
}
```

Manual by the library author, doesn't work well with templates

```python
from setuptools import setup, Extension
import pybind11


setup(name="adder",...)
```

python setup.py build_ext --inplace

```python
import adder
result = adder.add(3, 4)
```

cython

Manual by the library author

```python
# add.py
import cython

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

cythonize --annotate -3 --inplace add.py

```python
import add
result = add.add(3, 4)
```

numba

Automatic, no C++ support

```python
import numba
@numba.jit(noPython=True)
def add(a,b):
    return a+b

# Call the function
result = add(3, 4)
```

```llvm
; LLVM IR
define i32 @_ZN8__main__...dEdd(...) {
entry:
   %.6 = fadd double %arg.a, %arg.b
   store double %.6, double* %retptr, align 8
   ret i32 0
}
```

JIT

cppyy

Automatic with C++ support

#include <Eigen>

```python
import cppyy

cppyy.cppdef("
 int add(int a, int b) {
   return a+b;
 }")

# Call the function
result = cppyy.gbl.add(3, 4)
```

```llvm
; LLVM IR
define i32 @_ZN8__main__...dEdd(...) {
entry:
   %.6 = fadd double %arg.a, %arg.b
   store double %.6, double* %retptr, align 8
   ret i32 0
}
```

JIT

On-Demand Language Interoperability

Crossing the language barrier is expensive

Our Compiler-As-A-Service Approach solves that

```
In [1]: struct S { double val = 1.; };
```

```
In [2]: from libInterop import std
        python_vec = std.vector(S)(1)
```

```
In [3]: print(python_vec[0].val)

        1
```

```
In [4]: class Derived(S)
            def __init__(self):
                self.val = 0
        res = Derived()
```

```
In [5]: __global__ void sum_array(int n, double *x, double *sum) {
          for (int i = 0; i < n; i++) *sum += x[i];
        }
        // Init N=1M and x[i] = 1.f. Run kernel on 1M elements on the GPU.
        sum_array<<<1, 1>>>(N, x, &res.val);
```

Instead of Conclusion

Optimal performance is a continuous process in building trust in your:

- Developers
- Compilers
- Libraries
- Hardware

Just like trust, performance is hard-earned, easily lost, difficult to re-establish