Efficient Python Programming

Tutorial and Hands On

Aaron Jomy, CERN
Dr Vassil Vassilev, Princeton/CERN

Prerequisites:

All of the code in the tutorial and exercises can be run by pip installing the relevant packages

Windows users: (WSL2/Ubuntu) recipe:

Install Ubuntu on WSL from the [Microsoft Store]()

Linux/MacOS users can proceed to pip install the packages in a virtual environment

Further recipe and list of pip packages in the [README]():

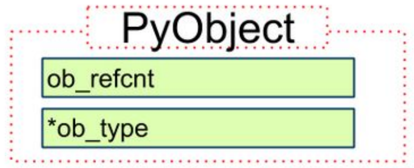
https://github.com/aaronj0/efficient-python-tutorials/

Recap

Why is python slow?

Dynamic typing

- Variables get a type only during runtime: Values(PyObjects) assigned to them.

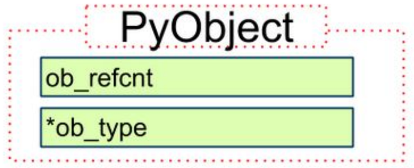


Python has only one data type, PyObject with a pointer to its runtime type, which is yet another PyObject*.*

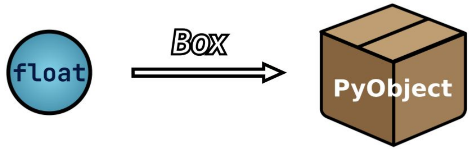
- Hard for the interpreter to optimize the execution.
- Compilers can make extensive analysis and optimization before the execution.

Why is python slow?

Dynamic typing

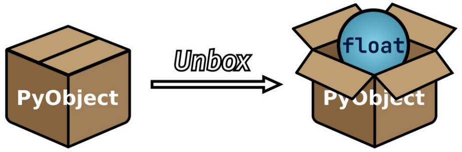


Python has only one data type, PyObject with a pointer to its runtime type, which is yet another PyObject*.*



```
>> f = 0.5
```

Python is a dynamically typed language(Duck Typing). It wraps and later unwraps objects (referred to as boxing/unboxing)



```
>> f = f ** 2
```

Why is python slow?

Flexible Data Structures

- Python builtins (lists, dictionaries, etc) -> Flexible

  Downsides: Very generic, hence not well suited for extensive numerical computations.
- Data structure implementations efficient when processing diverse data
- Heavy overhead when processing only a single type of data.

$$\begin{tikzcd}
 & \text{Flexible nature of Python} \arrow[dl] \arrow[dr] & \\
\text{Enhances programmer productivity} & & \text{Causes performance problems}
\end{tikzcd}$$

Why is python slow?

Flexible Data Structures

- Python builtins (lists, dictionaries, etc) -> Flexible
  Downsides: Very generic, hence not well suited for extensive numerical computations.
- Data structure implementations efficient when processing diverse data
- Heavy overhead when processing only a single type of data.

$$\begin{tikzcd}
& \text{Flexible nature of Python} \arrow[dl] \arrow[dr] & \\
\text{Enhances programmer productivity} & & \text{Causes performance problems}
\end{tikzcd}$$

Although just-in-time (JIT) compilation allow programs to be optimized at runtime, the inherent, dynamic nature of the Python programming language remains one of its main performance bottlenecks.

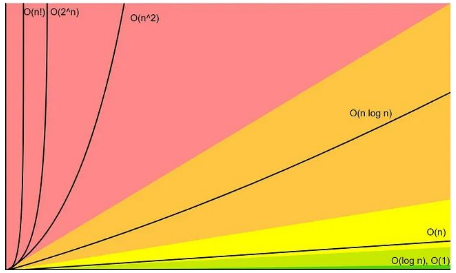
Before optimising our code:

- How do we measure performance?

Before optimising our code:

- How do we measure performance?

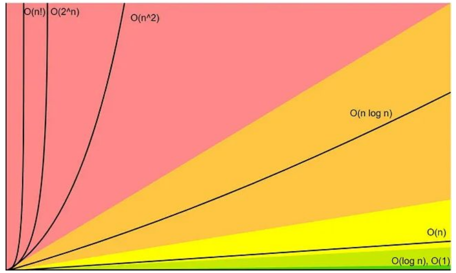
- Navigating algorithmic complexity



Before optimising our code:

- How do we measure performance?

- Navigating algorithmic complexity



- Tools that help achieve this

Performance Analysis

Profiling and optimising go hand in hand.

Performance Analysis

Profiling and optimising go hand in hand.

Objectives

- Learn how to benchmark and profile Python code
- Understand how optimization can be algorithmic(software design) or resource based(CPU/Memory)

Performance Analysis

Profiling and optimising go hand in hand.

Objectives

- Learn how to benchmark and profile Python code
- Understand how optimization can be algorithmic(software design) or resource based(CPU/Memory)

Often resource usage can be optimised by improving software design

Performance Analysis - Time

A simple example:

```
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```

```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```

Performance Analysis - Time

A simple example:

Better!

```
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```

```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```

timeit:

```
9 µs ± 405 ns per loop (mean ± std. dev.
of 7 runs, 100000 loops each)
```

```
15.7 µs ± 427 ns per loop (mean ± std.
dev. of 7 runs, 100000 loops each)
```

Performance Analysis - Time

A simple example:

~~Better!~~

```
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```

```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```

timeit:

```
9 µs ± 405 ns per loop (mean ± std. dev.
of 7 runs, 100000 loops each)
```

```
15.7 µs ± 427 ns per loop (mean ± std.
dev. of 7 runs, 100000 loops each)
```

execution time → not a good metric of algorithmic complexity

Performance Analysis - Time

A simple example:

~~Better!~~

```
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```

```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```

timeit:

```
9 µs ± 405 ns per loop (mean ± std. dev.
of 7 runs, 100000 loops each)
```

```
15.7 µs ± 427 ns per loop (mean ± std.
dev. of 7 runs, 100000 loops each)
```

execution time $\longrightarrow$ not a good metric of algorithmic complexity

We need a more objective complexity analysis metric for an algorithm

Performance Analysis - Time

A simple example:

~~Better!~~

```
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```

```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```

timeit:

```
9 µs ± 405 ns per loop (mean ± std. dev.
of 7 runs, 100000 loops each)
```

```
15.7 µs ± 427 ns per loop (mean ± std.
dev. of 7 runs, 100000 loops each)
```

execution time $\rightarrow$ not a good metric of algorithmic complexity

We need a more objective complexity analysis metric for an algorithm

line-by-line evaluation?

Performance Analysis - Line-by-line evaluation

```
def bubble_sort(a):
  n = len(a)
  for i in range(n):
      for j in range(n - i - 1):
        if a[j] > a[j + 1]:
          a[j], a[j + 1] = a[j + 1], a[j]
  return a
```

`line_profiler`

Useful to identify hotspots in code

(Try it yourself : Use line-profiler on a code example)

kernprof -h

Tip: Use verbosity (-lv)

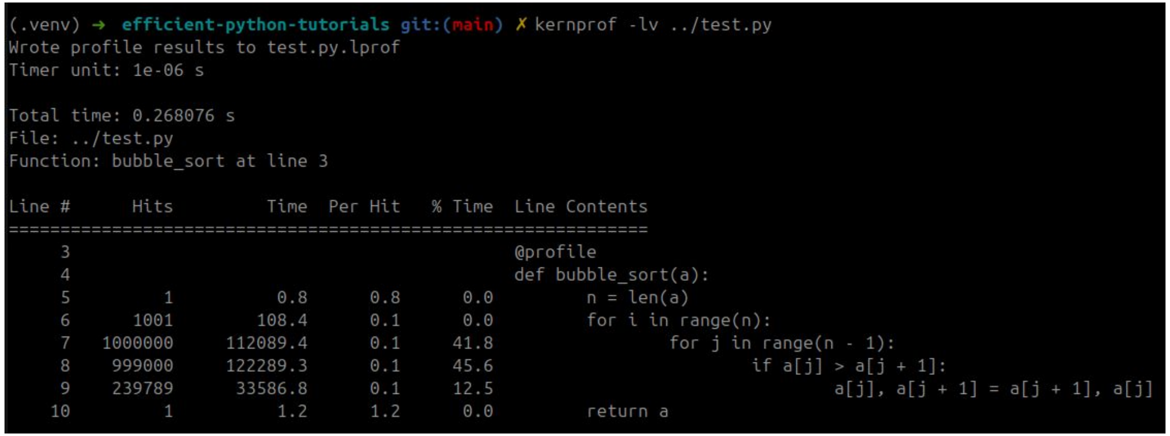
Performance Analysis - Line-by-line evaluation

```
def bubble_sort(a):
  n = len(a)
  for i in range(n):
      for j in range(n - i - 1):
        if a[j] > a[j + 1]:
          a[j], a[j + 1] = a[j + 1], a[j]
  return a
```

`line_profiler`

Useful to identify hotspots in code

Result:



```
(.venv) ➜  efficient-python-tutorials git:(main) ✗ kernprof -lv ../test.py
Wrote profile results to test.py.lprof
Timer unit: 1e-06 s

Total time: 0.268076 s
File: ../test.py
Function: bubble_sort at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def bubble_sort(a):
     5         1          0.8      0.8      0.0      n = len(a)
     6      1001        108.4      0.1      0.0      for i in range(n):
     7   1000000     112089.4      0.1     41.8          for j in range(n - 1):
     8    999000     122289.3      0.1     45.6              if a[j] > a[j + 1]:
     9    239789      33586.8      0.1     12.5                  a[j], a[j + 1] = a[j + 1], a[j]
    10         1          1.2      1.2      0.0      return a
```

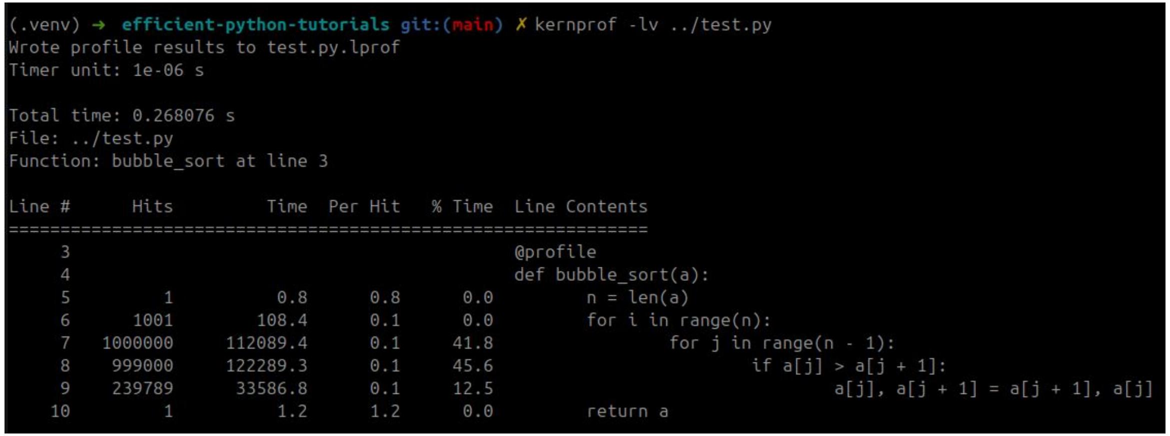
Performance Analysis - Line-by-line evaluation

```
def bubble_sort(a):
  n = len(a)
  for i in range(n):
      for j in range(n - i - 1):
        if a[j] > a[j + 1]:
          a[j], a[j + 1] = a[j + 1], a[j]
  return a
```

`line_profiler`

Useful to identify hotspots in code

How to optimise loop with highest number of hits?



```
(.venv) ➜  efficient-python-tutorials git:(main) ✗ kernprof -lv ../test.py
Wrote profile results to test.py.lprof
Timer unit: 1e-06 s

Total time: 0.268076 s
File: ../test.py
Function: bubble_sort at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def bubble_sort(a):
     5         1          0.8      0.8      0.0      n = len(a)
     6      1001        108.4      0.1      0.0      for i in range(n):
     7   1000000     112089.4      0.1     41.8          for j in range(n - 1):
     8    999000     122289.3      0.1     45.6              if a[j] > a[j + 1]:
     9    239789      33586.8      0.1     12.5                  a[j], a[j + 1] = a[j + 1], a[j]
    10         1          1.2      1.2      0.0      return a
```

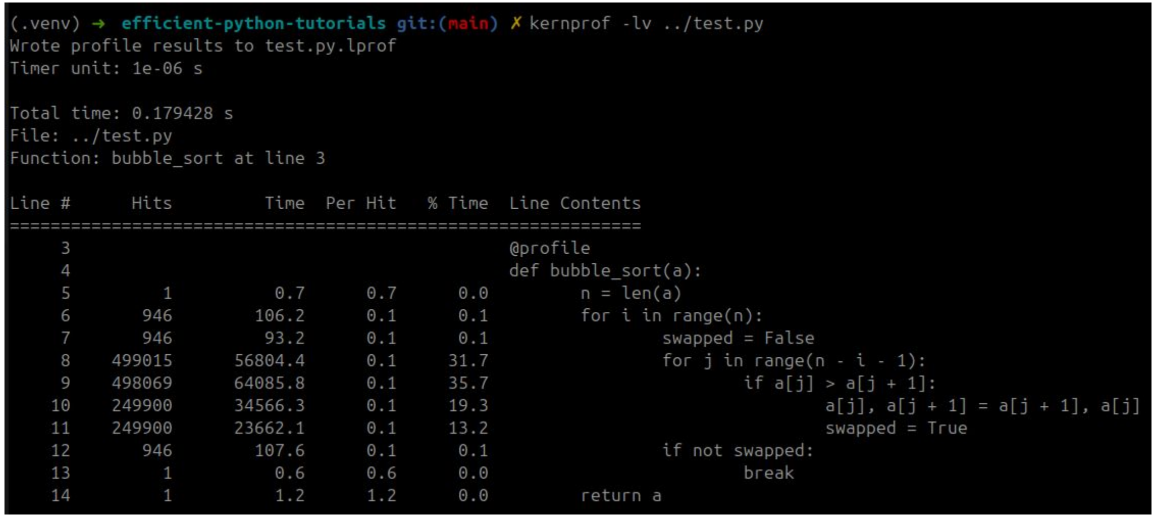
Performance Analysis - Line-by-line evaluation

```
def bubble_sort(a):
  n = len(a)
  for i in range(n):
      swapped = False
      for j in range(n - i - 1):
        if a[j] > a[j + 1]:
          a[j], a[j + 1] = a[j + 1], a[j]
          swapped = True
      if not swapped:
        break
  return a
```

`line_profiler`

Useful to identify hotspots in code

Let’s end the sort early



```
(.venv) → efficient-python-tutorials git:(main) ✗ kernprof -lv ../test.py
Wrote profile results to test.py.lprof
Timer unit: 1e-06 s

Total time: 0.179428 s
File: ../test.py
Function: bubble_sort at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def bubble_sort(a):
     5         1          0.7      0.7      0.0      n = len(a)
     6       946        106.2      0.1      0.1      for i in range(n):
     7       946         93.2      0.1      0.1          swapped = False
     8    499015      56804.4      0.1     31.7          for j in range(n - i - 1):
     9    498069      64085.8      0.1     35.7              if a[j] > a[j + 1]:
    10    249900      34566.3      0.1     19.3                  a[j], a[j + 1] = a[j + 1], a[j]
    11    249900      23662.1      0.1     13.2                  swapped = True
    12       946        107.6      0.1      0.1          if not swapped:
    13         1          0.6      0.6      0.0              break
    14         1          1.2      1.2      0.0      return a
```

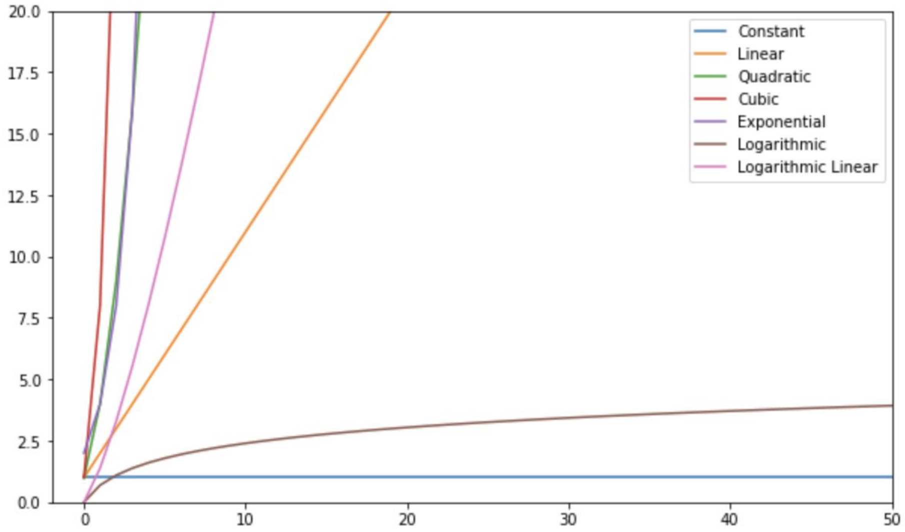
Performance Analysis - Big O

Big-O notation -> relationship between the input to the algorithm and the steps required to execute the algorithm.

Big-O isn't interested in a particular instance in which you run an algorithm, such as fact(50)*, but rather, in how well it* ***scales*** *given:*

a) Increasing input
b) Type of input

This is a much better evaluation metric than concrete time for a concrete instance!



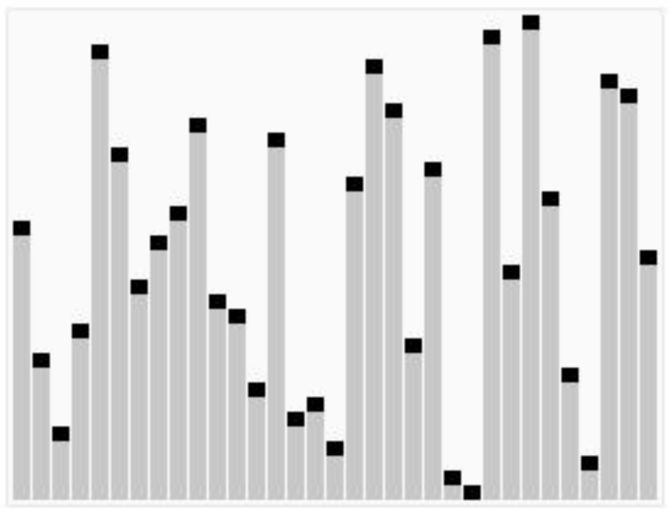
Performance Analysis 1

```python
def ex2(arr):
  if len(arr) <= 1:
      return arr
  pivot = arr[len(arr) // 2]
  left = [x for x in arr if x < pivot]
  middle = [x for x in arr if x ==
pivot]
  right = [x for x in arr if x > pivot]
  return ex2(left) + middle +
ex2(right)
```

Recap : what is the space and time complexity of this sorting algorithm?

Performance Analysis 1 - Quick Sort

```python
def ex2(arr):
  if len(arr) <= 1:
      return arr
  pivot = arr[len(arr) // 2]
  left = [x for x in arr if x < pivot]
  middle = [x for x in arr if x ==
pivot]
  right = [x for x in arr if x > pivot]
  return ex2(left) + middle +
ex2(right)
```



from https://github.com/the-akira

Performance Analysis 2

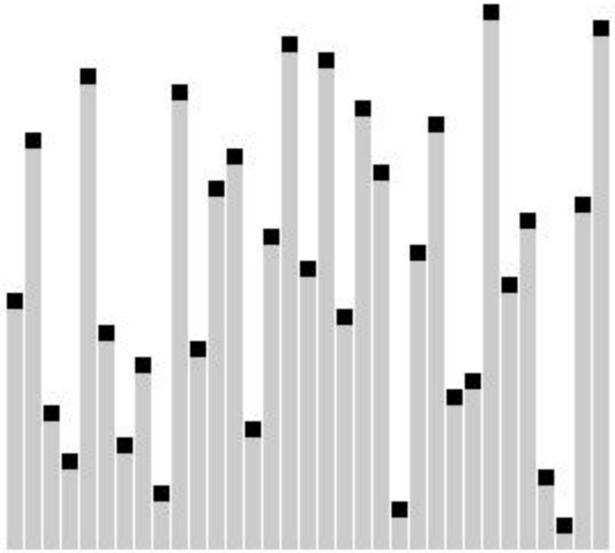
```python
def ex3(arr):
  if len(arr) > 1:
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    ex3(left)
    ex3(right)

    i = j = k = 0

    while i < len(left) and j <
len(right):
      if left[i] < right[j]:
        arr[k] = left[i]
        i += 1
      else:
        arr[k] = right[j]
        j += 1
      k += 1
```

```python
    while i < len(left):
      arr[k] = left[i]
      i += 1
      k += 1

    while j < len(right):
      arr[k] = right[j]
      j += 1
      k += 1
```

Performance Analysis 2 - Merge Sort

```python
def ex3(arr):
  if len(arr) > 1:
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    ex3(left)
    ex3(right)

    i = j = k = 0

    while i < len(left) and j <
len(right):
      if left[i] < right[j]:
        arr[k] = left[i]
        i += 1
      else:
        arr[k] = right[j]
        j += 1
      k += 1
```

```python
    while i < len(left):
      arr[k] = left[i]
      i += 1
      k += 1

    while j < len(right):
      arr[k] = right[j]
      j += 1
      k += 1
```



from https://github.com/the-akira

Is the Algorithm optimal?

Understanding of the maths behind the algorithm helps.

For certain algorithms, many of the bottlenecks will be linear algebra computations. In these cases, using the right function to solve the right problem is key.

Example:

An eigenvalue problem with a symmetric matrix is easier to solve than with a general matrix.

Moreover, most often, you can avoid inverting a matrix and use a less costly (and more numerically stable) operation.

However, it can be as simple as moving computation or memory allocation outside a loop, and this happens very often as well.

Performance Analysis - **CPU Usage**

```
sum = 0
for i in range(b.shape[0]):
     sum += i
```

vs

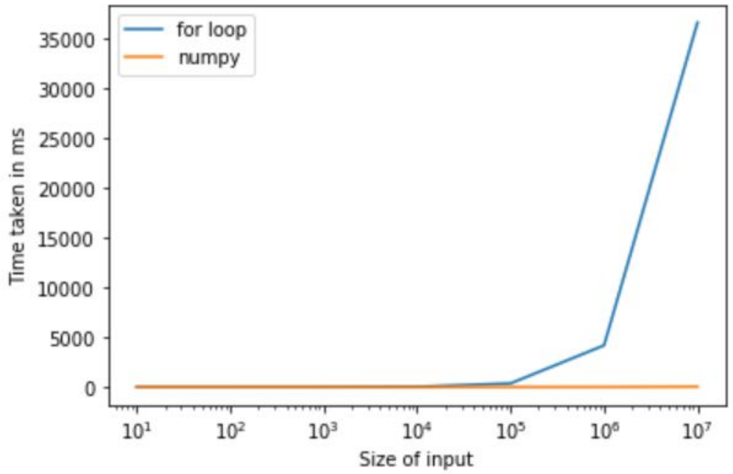
```
sum = np.sum(b)
```

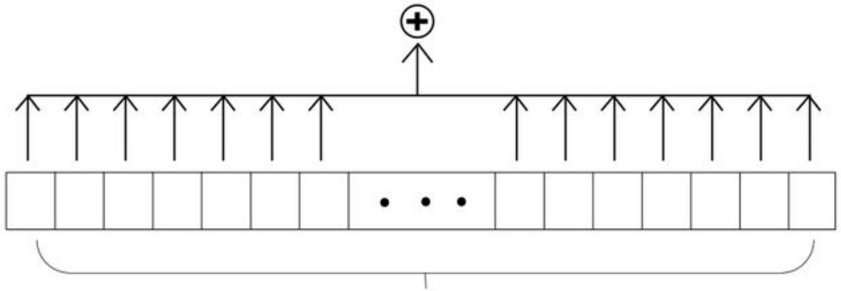
Performance Analysis - **CPU Usage**

```
sum = 0
for i in range(b.shape[0]):
     sum += i
```

vs

```
sum = np.sum(b)
```



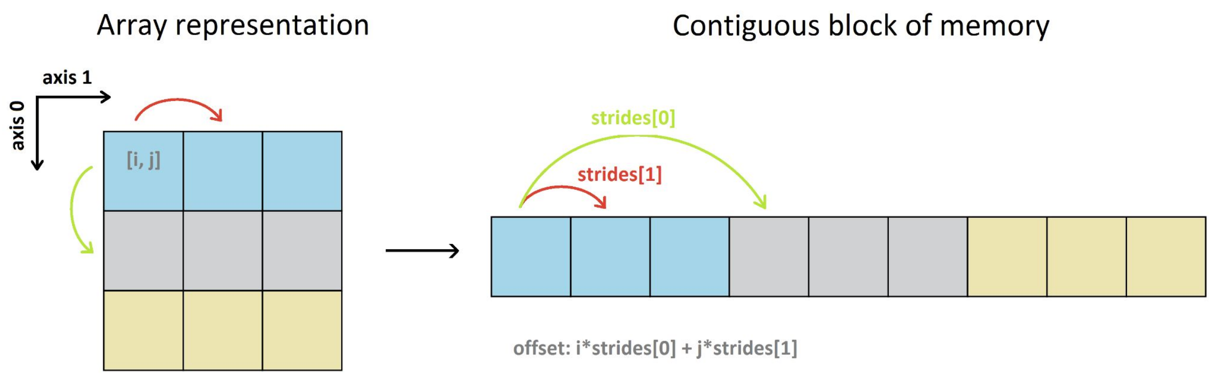


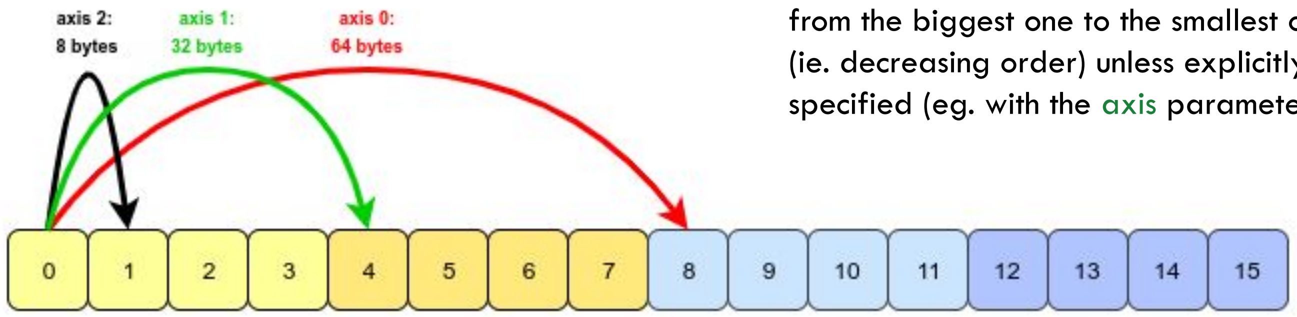
Vectorized implementation

Performance Analysis - **CPU Usage**

Notebook

Performance Analysis - **Memory**


Array representation
Contiguous block of memory
axis 1
axis 0
[i, j]
strides[0]
strides[1]
offset: i*strides[0] + j*strides[1]

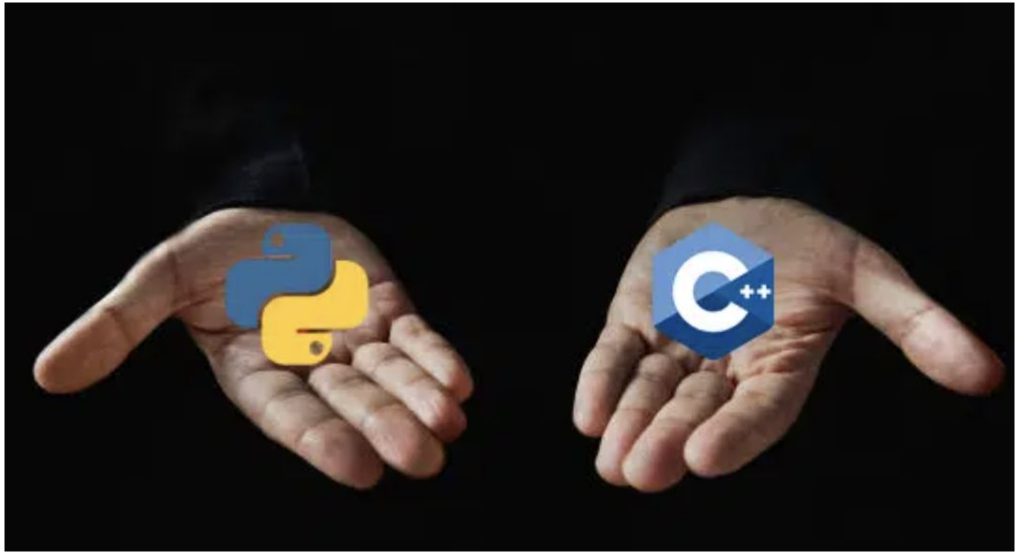


Numpy always iterates through the axis from the biggest one to the smallest one (ie. decreasing order) unless explicitly specified (eg. with the axis parameter)

Performance Analysis - **Memory**

Notebook

Beyond Python….



Beyond Python…. In Python

Beyond Python

A robust approach to speedups is by binding to faster compiled extensions (whether manual or automatic)

Beyond Python

A robust approach to speedups is by binding to faster compiled extensions (whether manual or automatic)

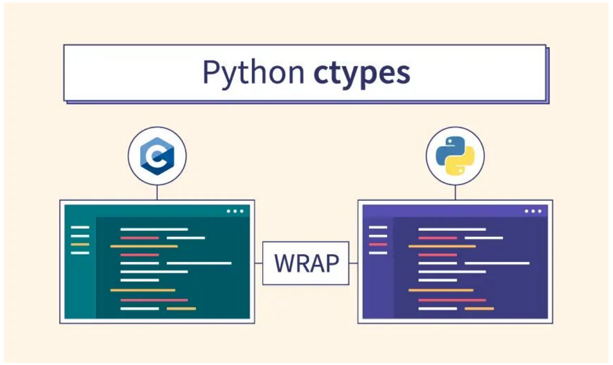
How do we do that?

Beyond Python - Bindings

A robust approach to speedups is by binding to faster compiled extensions (whether manual or automatic)

How do we do that?

`ctypes`

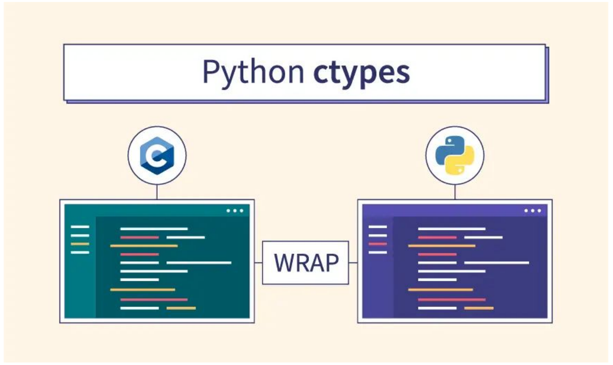


Ctypes

ctypes is a foreign function library for Python

It provides C compatible data types, and allows calling functions in DLLs or shared libraries

Allows wrapping these libraries in pure Python

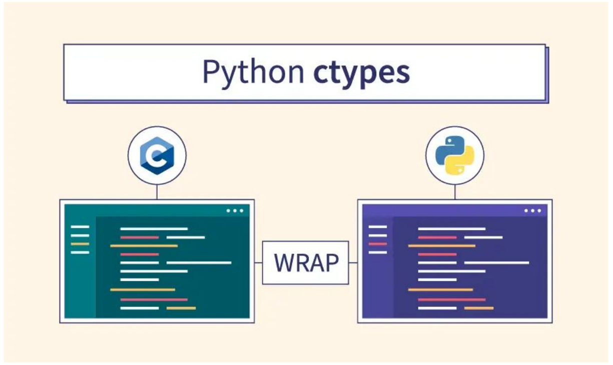


Ctypes

ctypes is a foreign function library for Python

It provides C compatible data types, and allows calling functions in DLLs or shared libraries

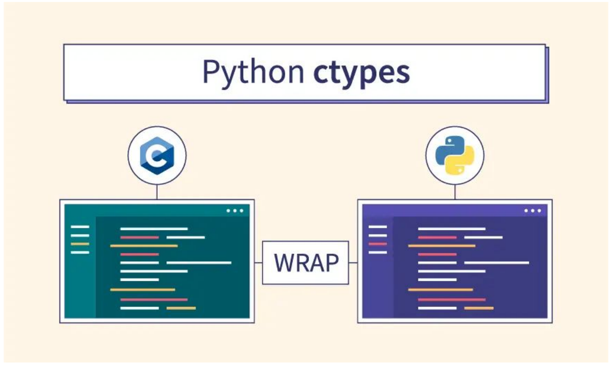
Allows wrapping these libraries in pure Python



Let's take a look at how we can compile, store and access these extensions:

Ctypes

What are the “compiled extensions” we are referring to?



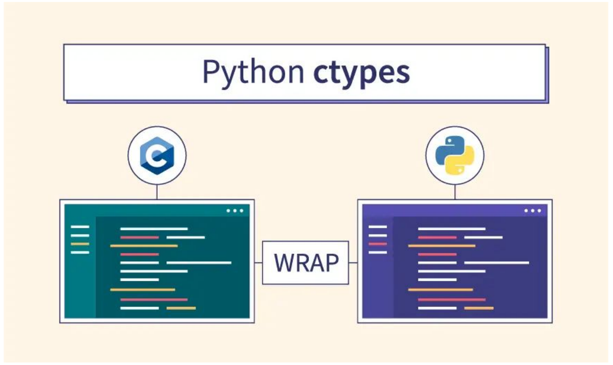
Ctypes

What are the “compiled extensions” we are referring to?

.so/.dll files

Dynamically linked libraries

Can be loaded at runtime by the OS



Hands On 2: Shared Objects and Bindings using Ctypes

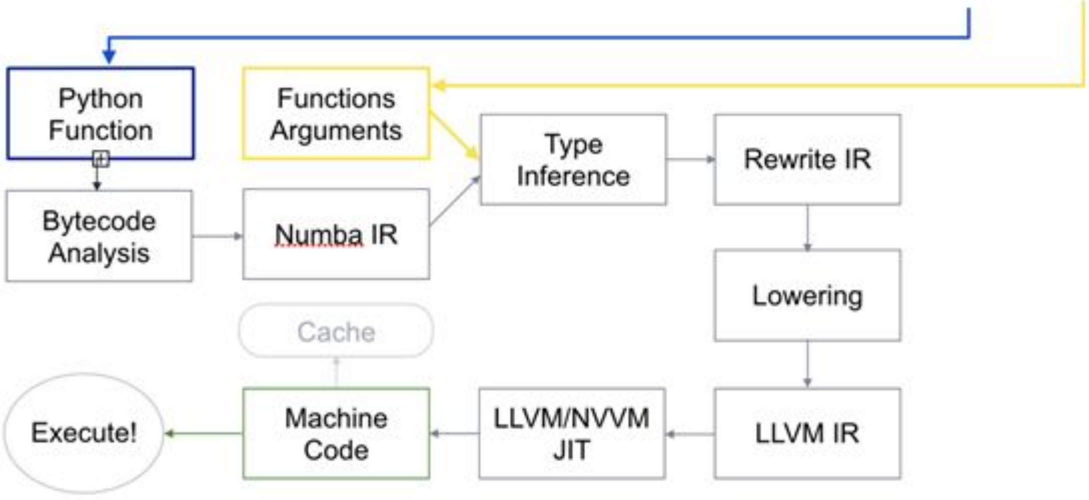
Notebook

Numba

Numba allows pure Python function to be JIT compiled to native machine instructions.

Numba

```
@jit
def do_math(a, b):
    ...
>>> do_math(x, y)
```



Numba

Numba allows pure Python function to be JIT compiled to native machine instructions.

Similar in performance to C, C++ and Fortran.

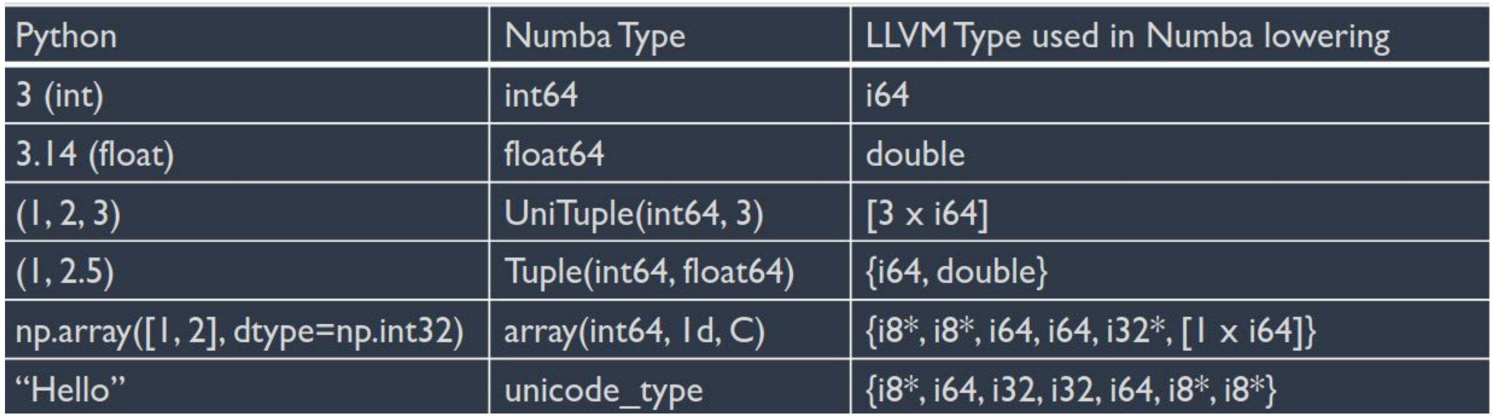
@jit compilation adds a one-time compilation overhead to the runtime of the function.

Once the function is JIT compiled and cached, subsequent calls will be fast.

The optimized machine code is generated by LLVM

Typing

Numba core has a type inference algorithm which assigns a nb_type for a variable



Python	Numba Type	LLVM Type used in Numba lowering
3 (int)	int64	i64
3.14 (float)	float64	double
(1, 2, 3)	UniTuple(int64, 3)	[3 x i64]
(1, 2.5)	Tuple(int64, float64)	{i64, double}
np.array([1, 2], dtype=np.int32)	array(int64, 1d, C)	{i8*, i8*, i64, i64, i32*, [1 x i64]}
“Hello”	unicode_type	{i8*, i64, i32, i32, i64, i8*, i8*}

Typing
Numba core has a type inference algorithm which assigns a nb_type for a variable

Lowering
High-level Python operations into low-level LLVM code.
Exploits typing to map to LLVM type

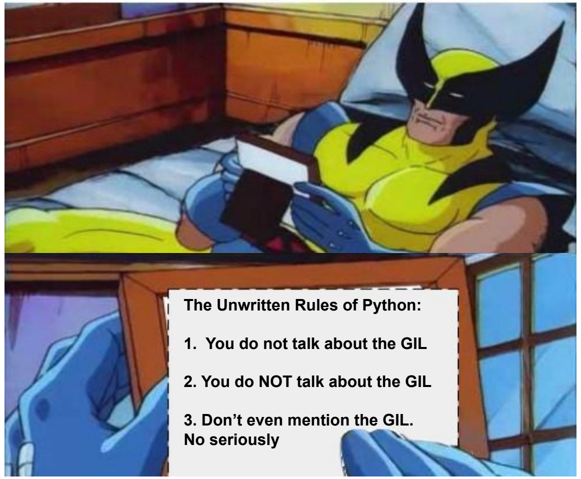
Boxing and unboxing
convert PyObject* ‘s into native values, and vice-versa.

Exercise 3: Accelerating Python with Numba

Notebook

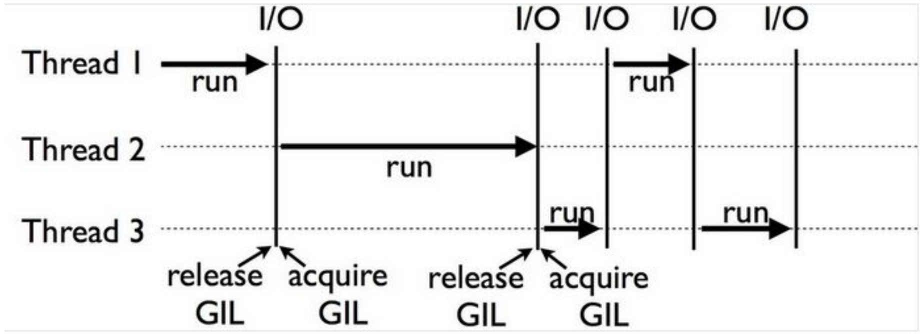
The GIL

Global Interpreter Lock: a mutex that protects access to Python objects


The Unwritten Rules of Python:
1. You do not talk about the GIL
2. You do NOT talk about the GIL
3. Don't even mention the GIL. No seriously

The GIL

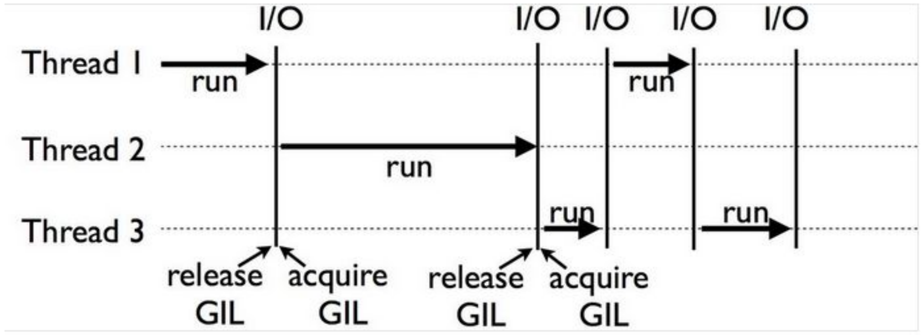
Global Interpreter Lock: a mutex that protects access to Python objects



The GIL

Global Interpreter Lock: a mutex that protects access to Python objects

Prevents multiple threads from executing Python bytecodes at once.

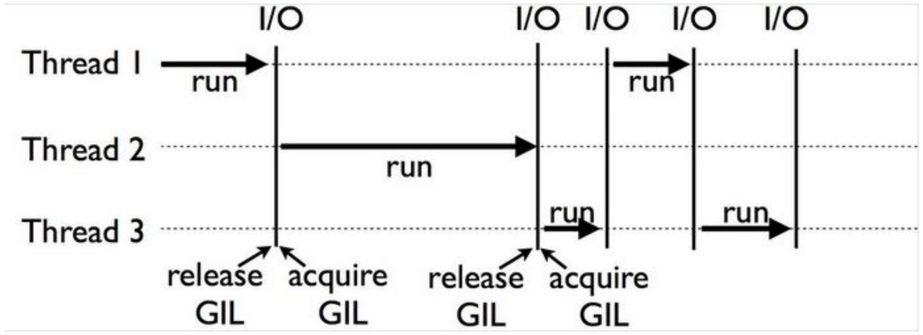


The GIL

Global Interpreter Lock: a mutex that protects access to Python objects

Prevents multiple threads from executing Python bytecodes at once.

Why? CPython's memory management is not thread-safe.

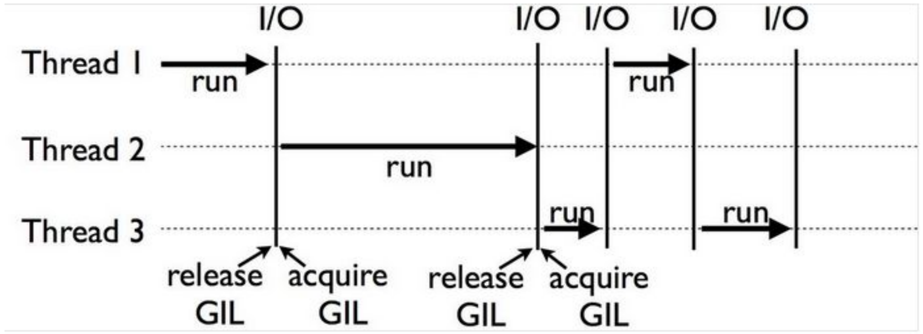


The GIL

Global Interpreter Lock: a mutex that protects access to Python objects

Prevents multiple threads from executing Python bytecodes at once.

Why? CPython’s memory management is not thread-safe.



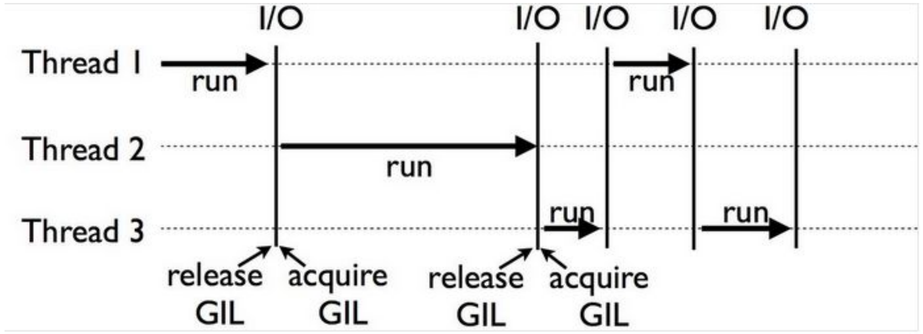
~~multithreading~~ cooperative multitasking

The GIL

Global Interpreter Lock: a mutex that protects access to Python objects

Prevents multiple threads from executing Python bytecodes at once.

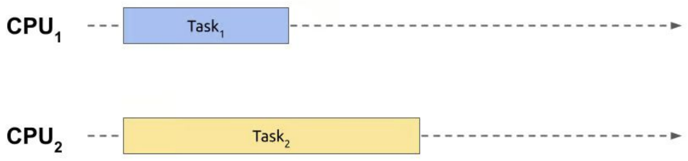
Why? CPython's memory management is not thread-safe.

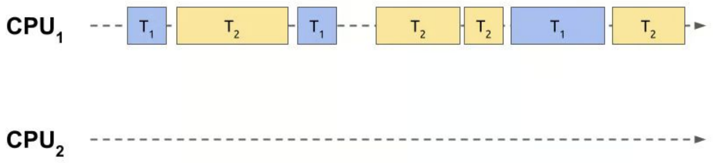


~~multithreading~~ *cooperative multitasking*

The GIL does not prevent the execution of C extensions that do not need the Python C API and can release the GIL.

The GIL - Parallel Processing





Release the GIL yourself?

This comes with some risk and is generally not a good idea

```
static PyObject* fibmodule_fib(PyObject* self, PyObject* args)
{

    int n, result;
    if (!PyArg_ParseTuple(args, "i", &n)) {
    return NULL;
    }

    Py_BEGIN_ALLOW_THREADS
    result = fib(n);
    Py_END_ALLOW_THREADS

    return Py_BuildValue("i", result);
}
```

Side note: Python >3.13 makes releasing the GIL optional

Solution?

Use GIL-Immune Libraries - NumPy (BLAS, SIMD)

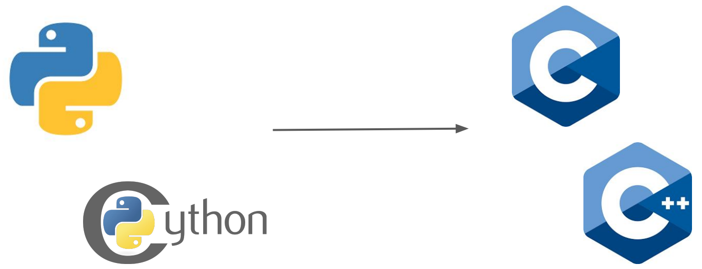
Use Cython + threading

Cython

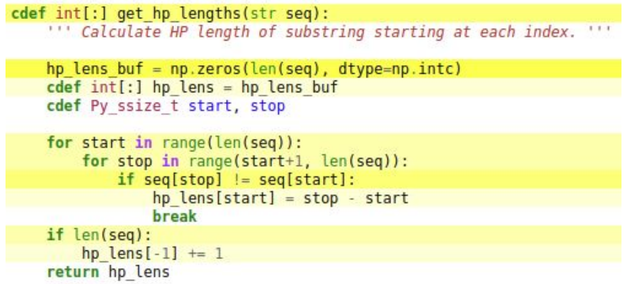
A superset of the Python programming language

Python-like syntax and C/C++-like static typing

Optimizing static compiler



Annotates how much of your code is C-like and what parts have pythonic interactions



```
cdef int[:] get_hp_lengths(str seq):
    ''' Calculate HP length of substring starting at each index. '''

    hp_lens_buf = np.zeros(len(seq), dtype=np.intc)
    cdef int[:] hp_lens = hp_lens_buf
    cdef Py_ssize_t start, stop

    for start in range(len(seq)):
        for stop in range(start+1, len(seq)):
            if seq[stop] != seq[start]:
                hp_lens[start] = stop - start
                break
    if len(seq):
        hp_lens[-1] += 1
    return hp_lens
```

Cython

Annotates how much of your code is C-like and what parts have pythonic interactions

Utilises static typing

Supports structs, unions, enums

C-style pointers

Notebook

[Notebook]

Exercise 4: Multithreading with Cython

Exercise 4: Multithreading with Cython

Recall the previous note:

The GIL does not prevent the execution of C extensions that do not need the Python C API and can release the GIL.

Can we get away with pure C?

Try it yourself:

- Instead of using Cython, can you multithread the fibonacci example in Ex 4 with a shared library and ctypes?

A Final Solution?

Should you use Numba or Cython? Does it matter?

- Performance is usually very similar and exact results depend on package versions.
- Numba is generally easier to use (just add `@jit`)
- Cython is more stable and mature, Numba developing faster
- Numba also works for GPUs
- Cython can compile arbitrary Python code and directly call C libraries, Numba has restrictions(subset of Python, object mode)
- Numba requires LLVM toolchain, Cython only C compiler.

pybind11

Lightweight header-only library that exposes C++ types in Python and vice versa

Creates Python bindings to existing C++ code

Provides core C++ features in Python:

- Functions with value, reference, or pointer parameters and return types
- Instance methods and static methods
- Overloaded functions
- Callbacks

pybind11

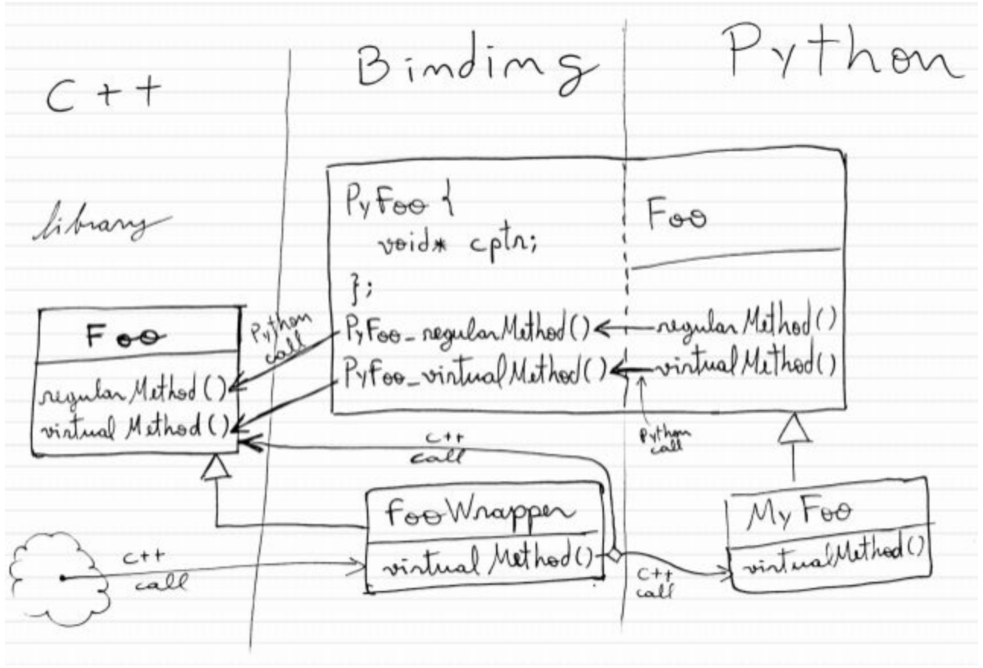
Provides core C++ features in Python:

- Functions with value, reference, or pointer parameters and return types
- Instance methods and static methods
- Overloaded functions
- Callbacks
- Iterators and ranges
- Single and multiple inheritance
- STL data structures
- Smart pointers with reference counting like std::shared_ptr
- And more…

pybind11

Notebook

Automatic bindings - **cppyy**



Automatic bindings - **cppyy**

cppyy: Yet another Python – C++ binder?

Yes, but bindings are **runtime**

Python is all runtime, so runtime is more natural

C++-side runtime-ness is provided by Cling

root-project/**cling**

The cling C++ interpreter

Allows for the **interactive** exploration of C++ libraries in Python

Automatic bindings - cppyy

Run-time generation enables:

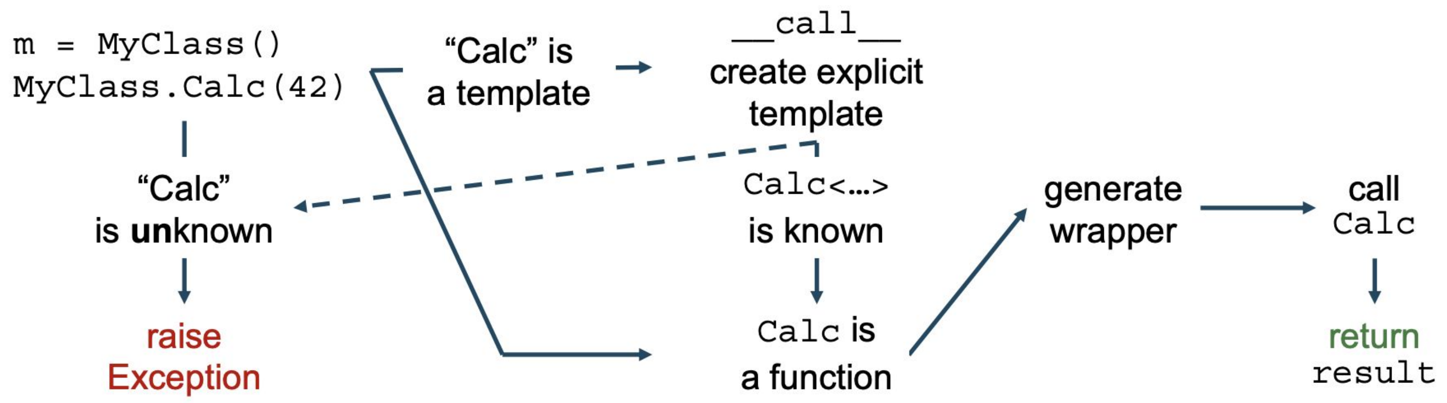
- Detailed specialization for higher performance
- Lazy loading for reduced memory use in large scale projects
- Python-side cross-inheritance and callbacks for working with C++ frameworks
- Run-time template instantiation
- Automatic object downcasting
- Exception mapping

Automatic bindings - cppyy

Notebook

Automatic bindings - **cppyy**

The how?



Wrappers of generated (and JITted) C++ are used to easily cover a range of C++isms, such as linkage of inline functions, overloaded operator new, default arguments, operator

Automatic bindings - cppyy

Hands on task:

- Write a header file using the Eigen C++ library (https://eigen.tuxfamily.org/dox/group__TutorialMatrixArithmetic.html)
- Interactively explore the C++ `VectorXd` class to create vectors in Python
- Call templated operations from Python like vector normalisation, vector squared norm
- Generate a plot using matplotlib

Is the Algorithm optimal?

Understanding of the maths behind the algorithm helps.

For certain algorithms, many of the bottlenecks will be linear algebra computations. In these cases, using the right function to solve the right problem is key.

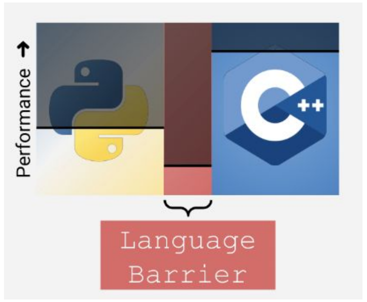
Is the Algorithm optimal?

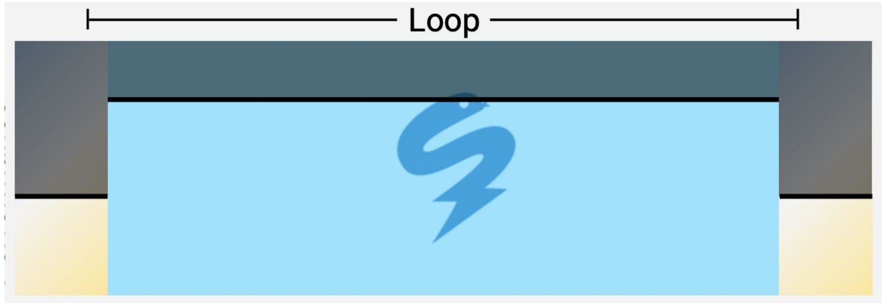
Understanding of the maths behind the algorithm helps.

For certain algorithms, many of the bottlenecks will be linear algebra computations. In these cases, using the right function to solve the right problem is key.

Example:

Bonus: Cppyy + Numba(experimental)





Notebook

Putting it all together

Cython pybind11 NumPy

cppyy Numba

Notebook