Data Science Survival Skills

Improving your code using code profiling and smart processing

Benchmarking my code

What is going wrong?

- It is not fast enough?
- It does not scale well
  - Across data
  - Across resources
- It is not accurate enough

Efficiency and scalability vs. bugs

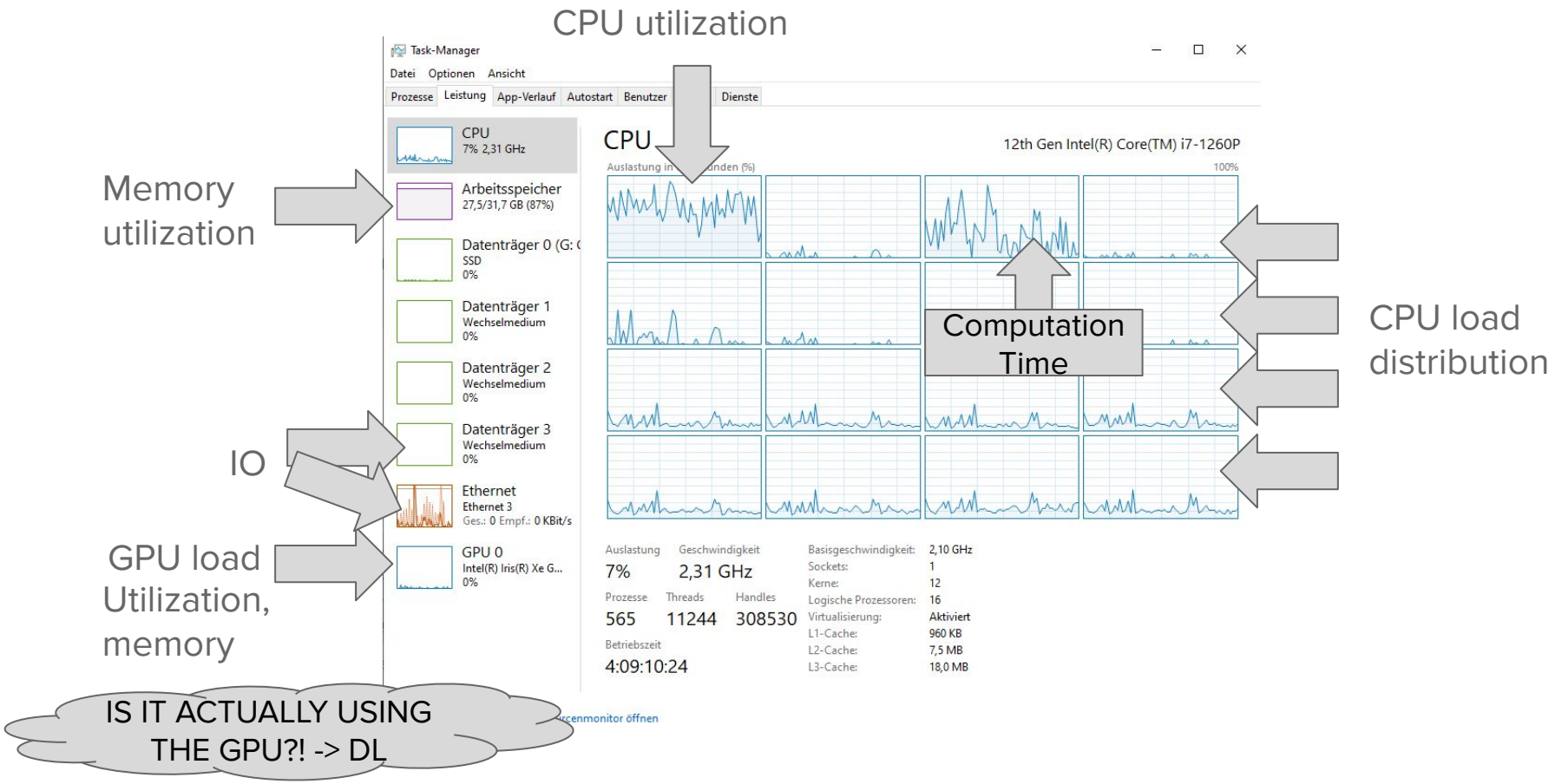
Is my code running well?

In software engineering, **profiling** ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance engineering.

https://en.wikipedia.org/wiki/Profiling_(computer_programming)

We are doing an experiment - what is the question and what can we measure?

What can we measure?



Profilers are

- A program by themselves ➔ are not perfect in efficiency
- Do have an influence on the program runtime
  - Create overhead because collecting data
  - Saving collected data

Types of profilers

- EVENT BASED
  - Only run on specific events, such as exceptions, statements, functions, ...
  - Start or end of a function
- SAMPLING BASED
  - Run in a discrete time difference

When do you do profiling?

Generally applicable in all production stages

Development: Require a lot of detail

Production: Require only little overhead

Built-in functions

- Modul 'time':

```
1: start = time.time()
2:
3: [...CODE HERE ...]
4:
5: end = time.time()
6: total = end - start
```

Modul 'timeit':
aus der Shell:
$python -m timeit "1+2"
10000000 loops, best of 3: 0.0196 usec per loop

im Skript:

```
1: import timeit
2: timeit.timeit(*METHODNAME*)
```

Tools

Tool	Description	Platforms	Profile level	Avg. overhead *
Intel® VTune™ Amplifier	• Rich GUI viewer • Mixed C/C++/Python code	Windows Linux	Line	~1.1-1.6x
cProfile (built-in)	• Text interactive mode: “pstats” (built-in) • GUI viewer: RunSnakeRun (Open Source) • PyCharm	Any	Function	1.3x-5x
Python Tools	• Visual Studio (2010+) • Open Source	Windows	Function	~2x
line_profiler	• Pure Python • Open Source • Text-only viewer	Any	Line	Up to 10x or more

* Measured against Grand Unified Python Benchmark

https://devopedia.org/profiling-python-code

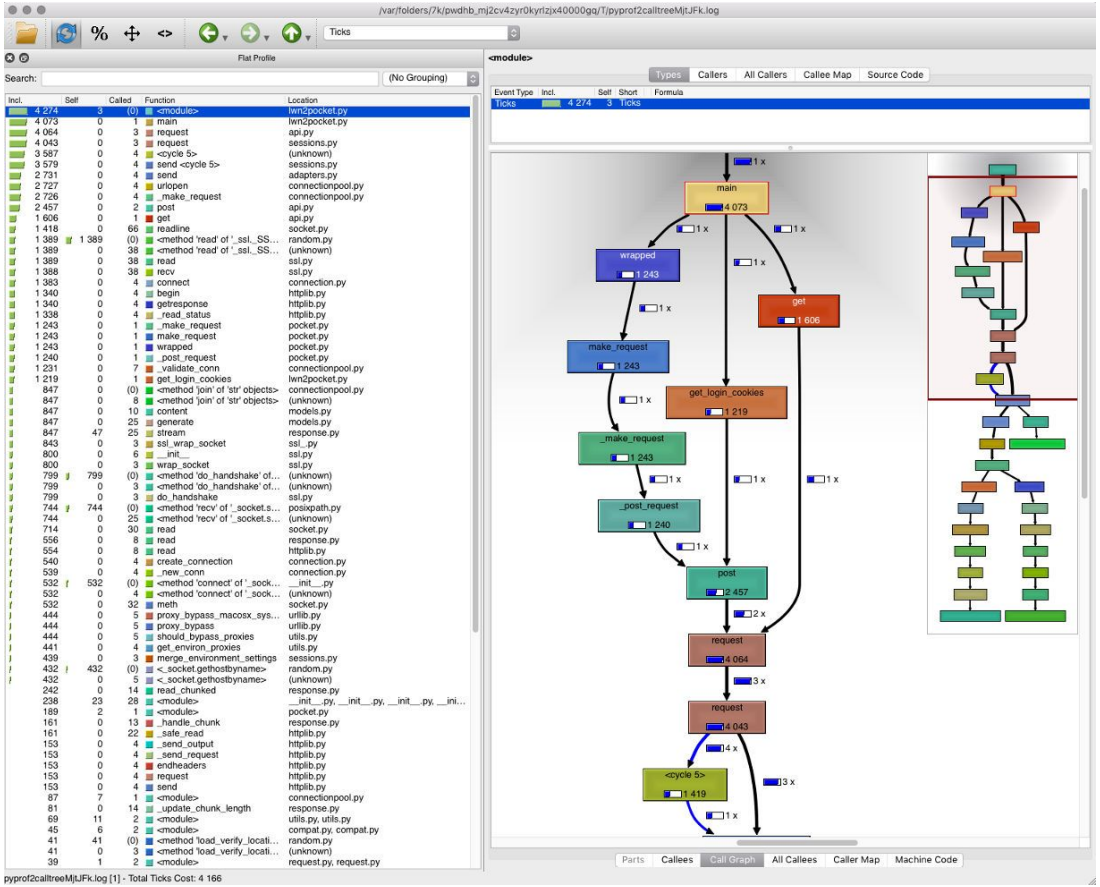
From IPython

💬 What are the IPython magic commands that help with profiling?

IPython has the following magic commands for profiling:*

- `%time` : Shows how long a one or more lines of code take to run.
- `%timeit` : Like `%time` but gives an average from multiple runs. Option `-n` can be used to specify the number of runs. Depending on how long the program takes, the number of runs is limited automatically. This is unlike the `timeit` module.
- `%prun` : Shows time taken by each function.
- `%lprun` : Shows time taken line by line. Functions to profile can be specified with `-f` option.
- `%mprun` : Shows how much memory is used.
- `%memit` : Like `%mprun` but gives an average from multiple runs, which can be specified with `-r` option.

Visualization using KCachegrid

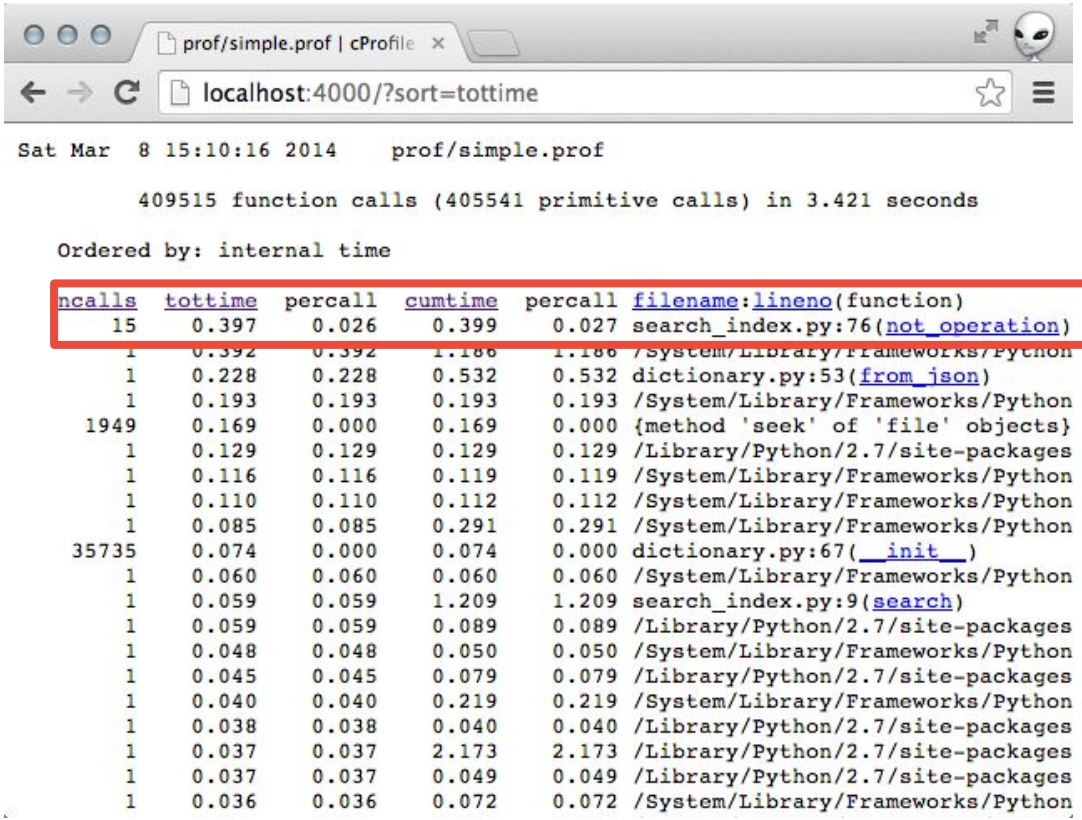


CProfiler example

- How long each call took (percall, inclusive and exclusive)
- How many times it was called (ncalls)
- How long it took (cumtime: includes the times of other functions it calls)
- How long it actually took (tottime: excludes the times of other functions)
- What functions it called (callers)
- What functions called it (callees)

https://ymichael.com/2014/03/08/profiling-python-with-cprofile.html

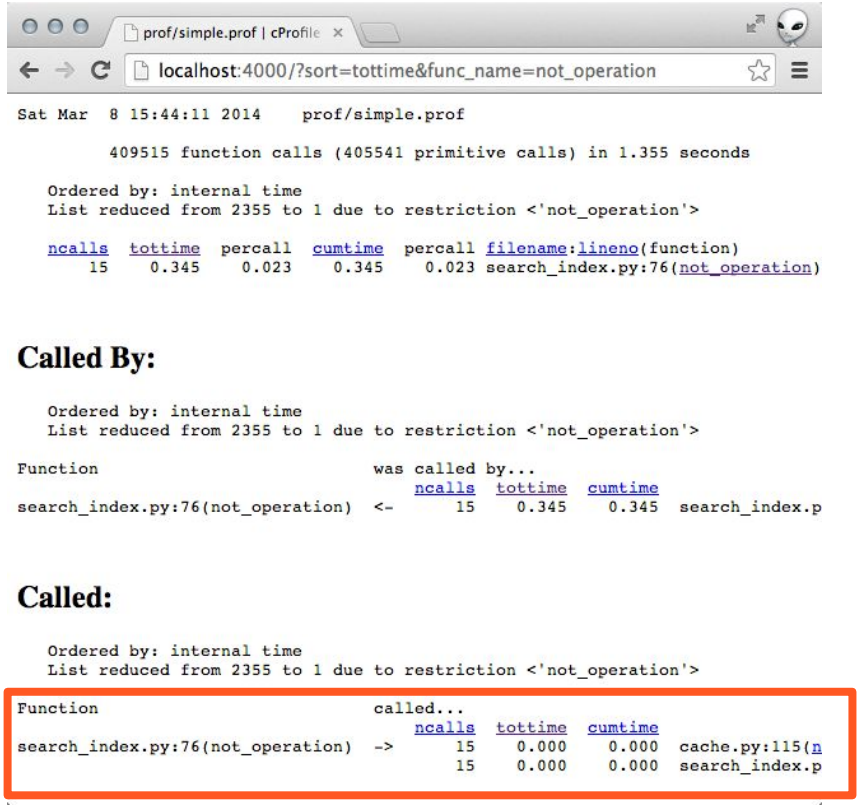
An example: finding inefficient code



A. It is most useful for handling multiple I/O operations.

https://ymichael.com/2014/03/08/profiling-python-with-cprofile.html

Functions called by not_operations are fast..



https://ymichael.com/2014/03/08/profiling-python-with-cprofile.html

Optimizing list comprehension

```
# line 76 of search_index.py
def not_operation(operand, dictionary, pfile):
    """Performs the operation `NOT operand`."""

    # A list of all the documents (sorted)
    all_docs = dictionary.all_docs()

    # A list of the documents matching `operand` (sorted)
    results = get_results(operand, dictionary, pfile, force_list=True)

    return [doc for doc in all_docs if doc not in results]
```

```
$ time python search.py
# real    0m1.521s
# user    0m1.250s
# sys     0m0.142s
```

```
# the fix.
def not_operation(operand, dictionary, pfile):
    """Performs the operation `NOT operand`."""

    # A list of all the documents (sorted)
    all_docs = dictionary.all_docs()

    # A list of the documents matching `operand` (sorted)
    results = get_results(operand, dictionary, pfile, force_list=True)

    return list_a_and_not_list_b(all_docs, results)


def list_a_and_not_list_b(a, b):
    """Returns `a AND NOT b`.

    Both a and b are expected to be sorted lists.

    """
    results = []
    idx_a = 0
    idx_b = 0
    while idx_a < len(a) and idx_b < len(b):
        if a[idx_a] < b[idx_b]:
            results.append(a[idx_a])
            idx_a += 1
        elif b[idx_b] < a[idx_a]:
            idx_b += 1
        else:
            idx_a += 1
            idx_b += 1

    while idx_a < len(a):
        results.append(a[idx_a])
        idx_a += 1

    return results
```

Replacement below

```
time python search.py
# real    0m1.160s
# user    0m1.018s
# sys     0m0.133s
```

https://ymichael.com/2014/03/08/profiling-python-with-cprofile.html

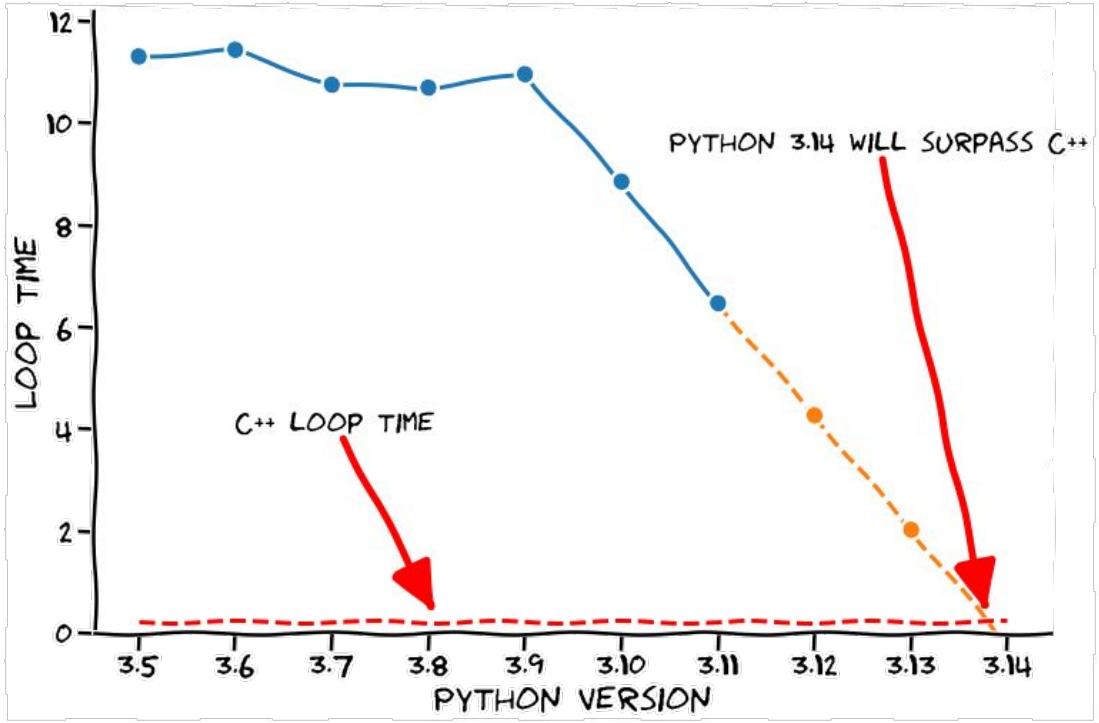
Tipps for profiling/optimization

Optimize only what matters:

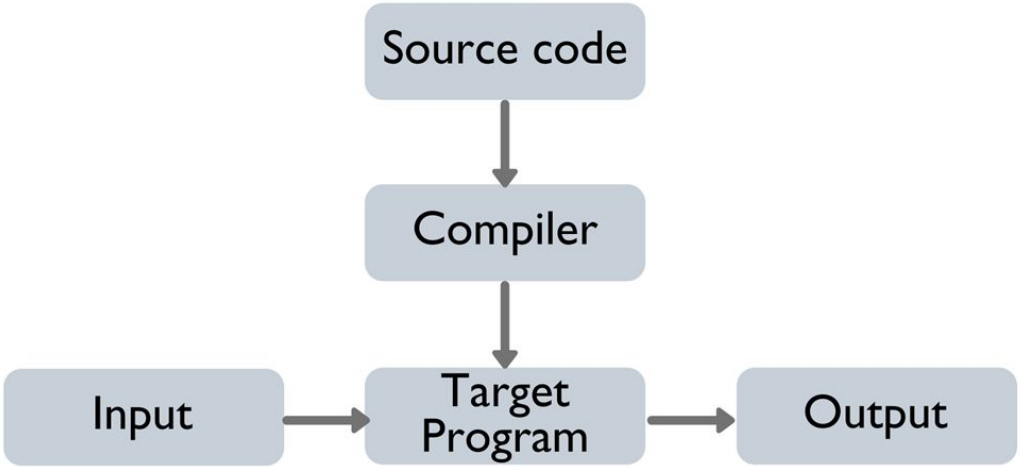
Keep in mind that speeding up a function 100 times is irrelevant if that function takes up only a few percent of the program's total execution.

- I/O is a bottleneck, **threading** can help.
- If your code uses a lot of regex, try to replace these with string equivalents, which will run faster.
- To avoid repeat computations, earlier results can be cached. This is called *memoization*.
- **If there are no obvious places to optimize the code, you can also consider an alternative runtime such as PyPy or moving critical parts of the code into Cython or C and calling the same from Python.**
- **Likewise, vectorize some operations using NumPy.**

Why is C faster than Python?



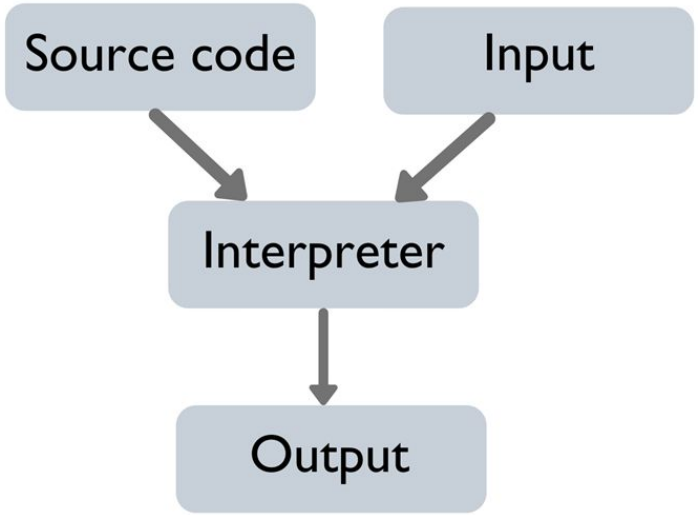
Compiling code



- Compilation is the process in which the source code is translated into the target language, where the target language is sometimes machine-readable code and in other times it serves as the input to another compiler or interpreter.
- It takes the entire program at a time as the input.
- The code is translated once into object code and can be run many times.

https://www.c-sharpcorner.com/article/why-learn-python-an-introduction-to-python/

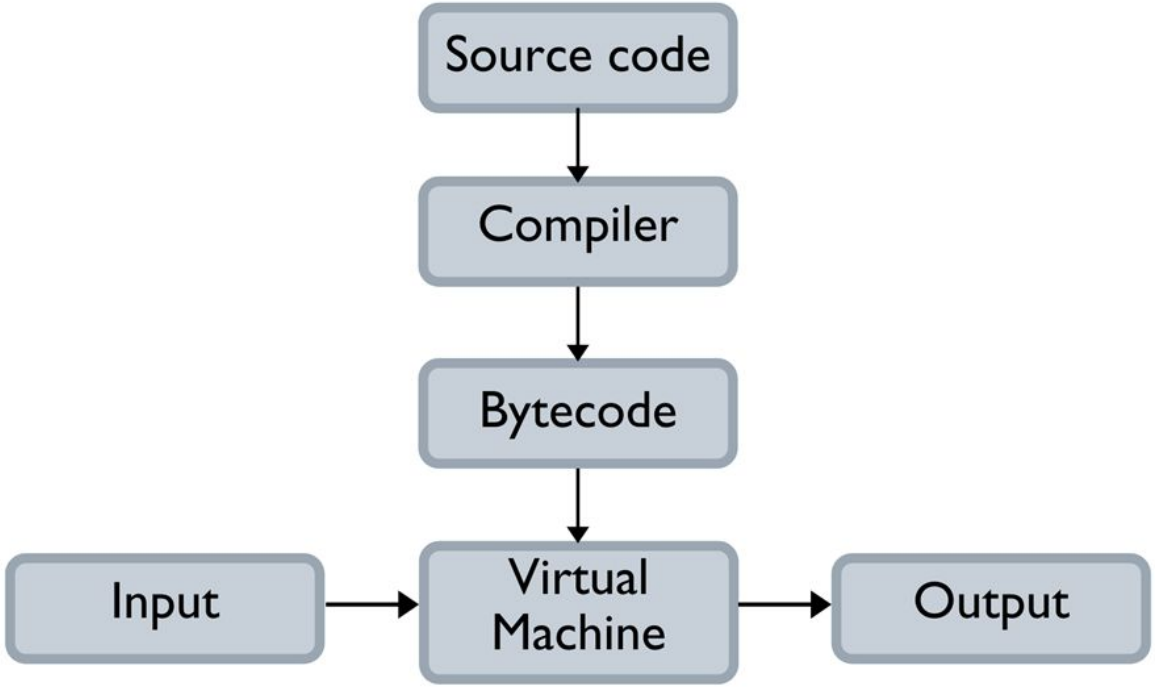
Interpreting code



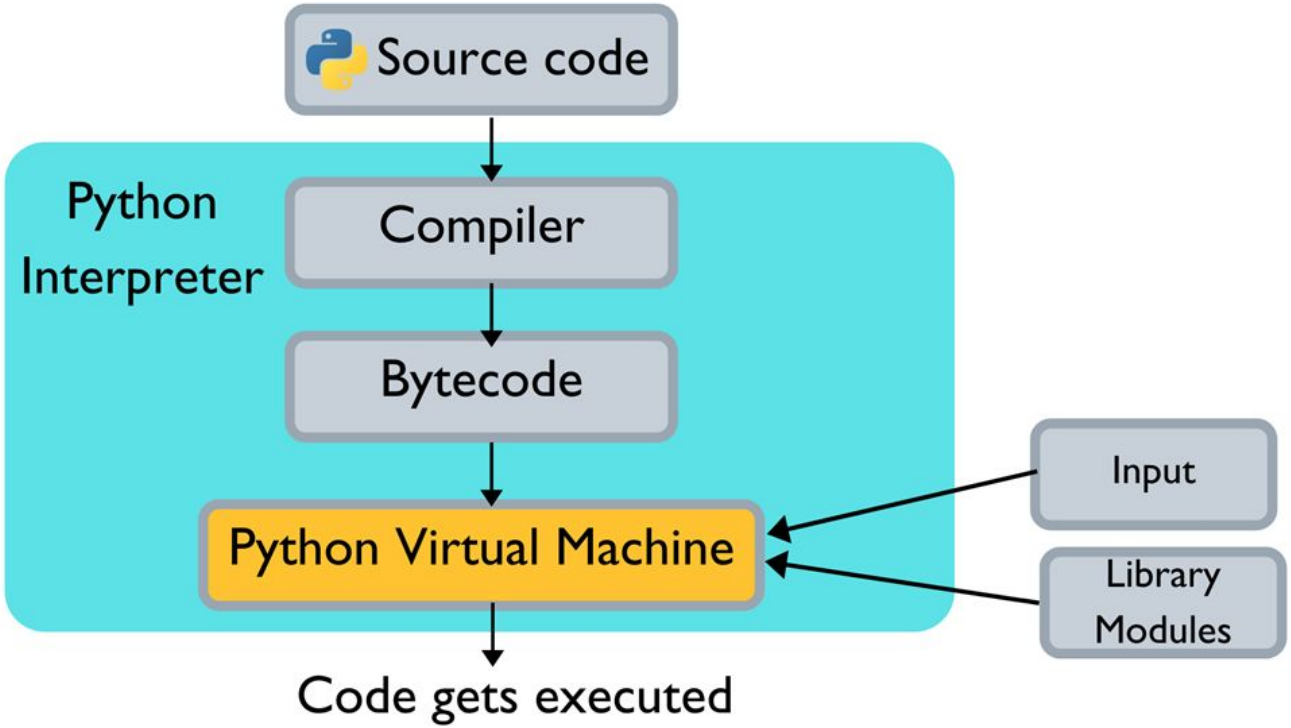
- Interpretation is the process in which it reads the source code and immediately reacts to them.
- Interpreter executes the instruction specified in the source code while the program gets executed.
- It takes a single line of code at a time and executes it.

- PARSING
- REACTING
- EXECUTING

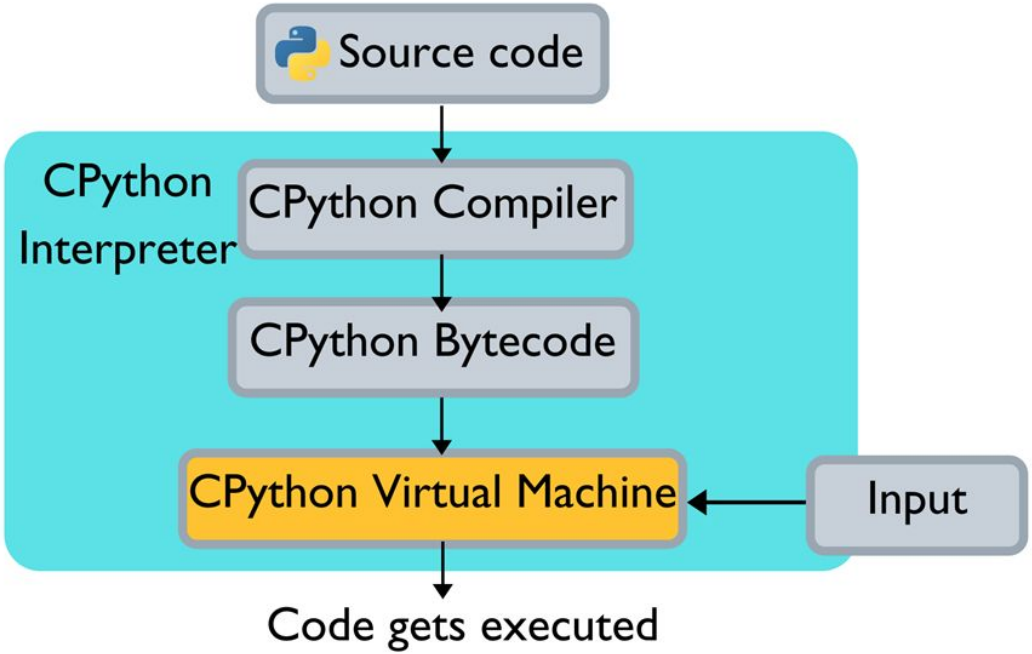
Intermediate step: bytecode



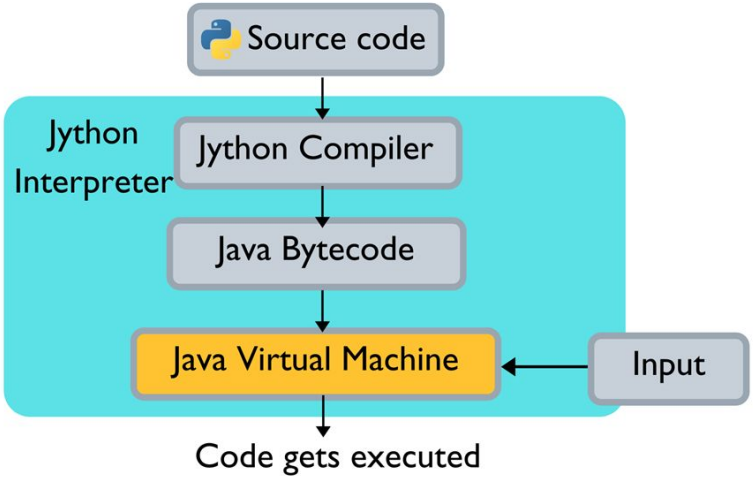
Code interpreting



CPython - the standard Python distribution



Jython, IronPython and PyPy

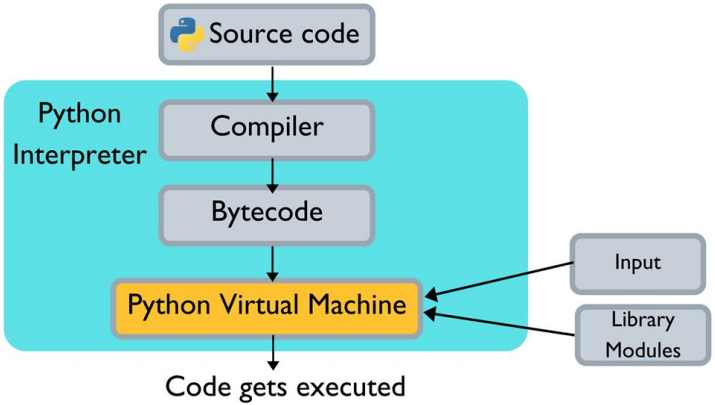


IronPython: C#
PyPy: Python

➜ still faster than CPython!

Steps involved when you run your Python code

1. The compiler receives the source code.
2. The compiler checks the syntax of each line in the source code.
3. If the compiler encounters an error, it halts the translation process with an error message (**Syntax error**).
4. Else if the instruction is well formatted, then it translates the source code to Bytecode.
5. Bytecode is sent to the Python Virtual Machine (PVM)
6. Bytecode along with the inputs and Library modules is given as the input to the PVM.
7. PVM executes the Bytecode and if any error occurs, it displays an error message (**Runtime error**).
8. Otherwise, if there is no error in execution, it results in the output.



Why is Python slow?

- Reason 1: Interpreted language (see slides before)
- Reason 2: Global interpreter lock (only one thread at a time)*
- Reason 3: Dynamic typing (% type hinting ➔ only readability)

Further watching: https://www.youtube.com/watch?v=bC428xPW1H8

What can we do about it?

- Utilize static typing and pre-compiled code (e.g. C in CPython)

Three ways:

- Write C-code, compile it and access it
- Write code in Cython and compile it
- Use LLVM compiler with numba

Writing C code

C++

```
// cmult.c
float cmult(int int_param, float float_param) {
    float return_value = int_param * float_param;
    printf("    In cmult : int: %d float %.1f returning  %.1f\n", int_param,
           float_param, return_value);
    return return_value;
}
```

https://realpython.com/python-bindings-overview/#ctypes

Compiling

- GNU C Compiler (GCC)
- MinGW (recommended for Windows)
  To interface GCC


```
$ gcc -o basic_function_win32.so -shared -fPIC -O2 basic_function.c # Windows

$ gcc -o basic_function_darwin.so -shared -fPIC -O2 basic_function.c # Mac

$ gcc -o basic_function_linux.so -shared -fPIC -O2 basic_function.c # Linux
```

Interfacing with ctypes

1. Load your library.
2. Wrap some of your input parameters.
3. Tell `ctypes` the return type of your function.

Python

```python
# ctypes_test.py
import ctypes
import pathlib

if __name__ == "__main__":
    # Load the shared library into ctypes
    libname = pathlib.Path().absolute() / "libcmult.so"
    c_lib = ctypes.CDLL(libname)
```

Ctypes loading shared library

16.16.2.2. Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

class `ctypes.`**`CDLL`**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

class `ctypes.`**`OleDLL`**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific `HRESULT` code. `HRESULT` values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an `OSError` is automatically raised.

Changed in version 3.3: `WindowsError` used to be raised.

class `ctypes.`**`WinDLL`**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

On Windows CE only the standard calling convention is used, for convenience the `WinDLL` and `OleDLL` use the standard calling convention on this platform.

The Python global interpreter lock is released before calling any function exported by these libraries, and reacquired afterwards.

class `ctypes.`**`PyDLL`**(*name*, *mode=DEFAULT_MODE*, *handle=None*)

Instances of this class behave like `CDLL` instances, except that the Python GIL is *not* released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

Releases GIL

Keeps GIL

Trying to get it to work...

Python

```
x, y = 6, 2.3
answer = c_lib.cmult(x, y)
```

```
ctypes.ArgumentError: argument 2:
<class 'TypeError'>: Don't know
how to convert parameter 2
```

=> Need to specify every non-int parameter

Set input datatype...

Python

```
# ctypes_test.py
answer = c_lib.cmult(x, ctypes.c_float(y))
print(f"    In Python: int: {x} float {y:.1f} return val {answer:.1f}")
```

Shell

```
$ invoke test-ctypes
    In cmult : int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 48.0
```

Oh wait...

Shell

```
$ invoke test-ctypes
    In cmult : int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 48.0
```

Wait a minute... `6` multiplied by `2.3` is not `48.0`!

Python assumes the returning value is also an integer, therefore, we need to change this to c_float

Python

```
# ctypes_test.py
c_lib.cmult.restype = ctypes.c_float
answer = c_lib.cmult(x, ctypes.c_float(y))
print(f"    In Python: int: {x} float {y:.1f} return val {answer:.1f}")
```

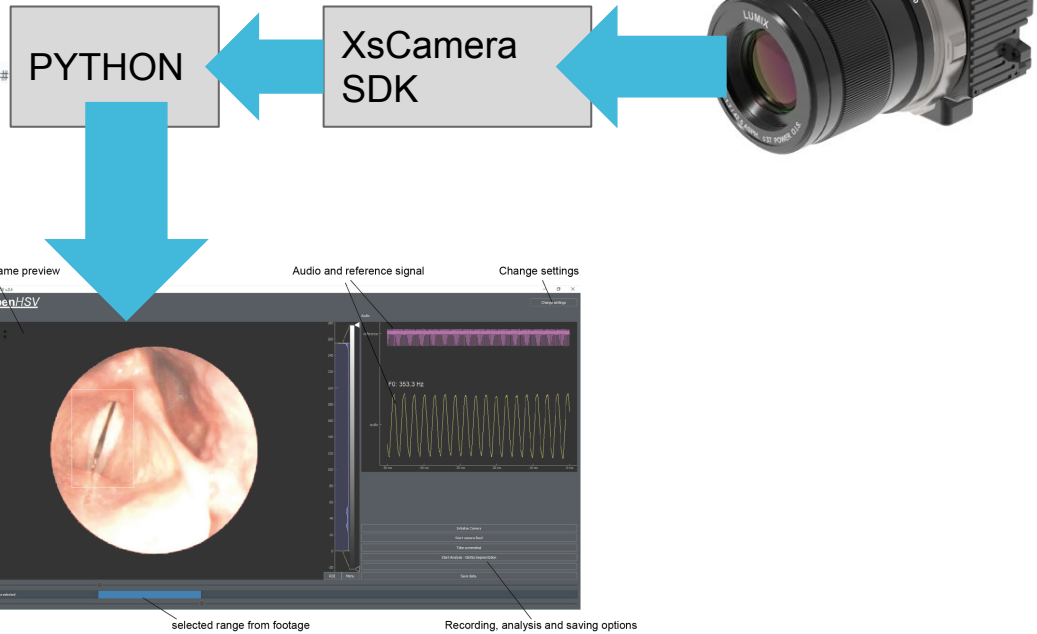
Shell

```
$ invoke test-ctypes
================================================
= Building C Library
* Complete
================================================
= Testing ctypes Module
    In cmult : int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 48.0

    In cmult : int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 13.8
```

You can access very complex DLL/SO libraries...

2523 lines (2337 sloc) | 106 KB

```
1  ##############################################################################
2  #   XsCamera.py - Python wrapper class for IDT cameras
3  #   Version 2.15.01
4  #   Copyright (C) 2000-2019 Integrated Design Tools, Inc.
5  #   ALL RIGHTS RESERVED
6  ##############################################################################
7
8  import ctypes
9  import sys
10 import types
11 import ctypes.util
12
13 ##############################################################################
14 # Loading library
15
16 if sys.platform.startswith('win'):
17     _library_loader = ctypes.windll
18     CALLBACK_TYPE = ctypes.WINFUNCTYPE
19 else:
20     _library_loader = ctypes.cdll
21     CALLBACK_TYPE = ctypes.CFUNCTYPE
22
23 def FindLibrary(library_name = "XStreamDrv", library_loader = _library_loader):
24     library_path = ctypes.util.find_library(library_name)
25
26     if library_path:
27         return library_loader.LoadLibrary(library_path)
28     else:
29         return library_loader.XStreamDrv
30
31 class LibraryNotLoadedException(Exception):
32     pass
33
34 class LibraryAlreadyLoadedException(Exception):
```



Cython

- Accessing c-functions
- Write c-functions

The most common form of declaring a module in `Cython` is to use a `.pyx` file:

Python

```python
# cython_example.pyx
""" Example cython interface definition """

cdef extern from "cppmult.hpp":
    float cppmult(int int_param, float float_param)

def pymult( int_param, float_param ):
    return cppmult( int_param, float_param )
```

Building Cython module

Python

```python
# tasks.py
def compile_python_module(cpp_name, extension_name):
    invoke.run(
        "g++ -O3 -Wall -Werror -shared -std=c++11 -fPIC "
        "`python3 -m pybind11 --includes` "
        "-I /usr/include/python3.7 -I .  "
        "{0} "
        "-o {1}`python3.7-config --extension-suffix` "
        "-L. -lcppmult -Wl,-rpath,.".format(cpp_name, extension_name)
    )

def build_cython(c):
    """ Build the cython extension module """
    print_banner("Building Cython Module")
    # Run cython on the pyx file to create a .cpp file
    invoke.run("cython --cplus -3 cython_example.pyx -o cython_wrapper.cpp")

    # Compile and link the cython wrapper library
    compile_python_module("cython_wrapper.cpp", "cython_example")
    print("* Complete")
```

Shell

```
$ invoke build-cython
========================================
= Building C++ Library
* Complete
========================================
= Building Cython Module
* Complete
```

Running module

Python

```python
# cython_test.py
import cython_example

# Sample data for your call
x, y = 6, 2.3

answer = cython_example.pymult(x, y)
print(f"    In Python: int: {x} float {y:.1f} return val {answer:.1f}")
```

Shell

```
$ invoke test-cython
================================================
= Testing Cython Module
    In cppmul: int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 13.8
```

Cython static typing

integrate.py

```
def f(x):
    return x ** 2 - x


def integrate_f(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

In Cython: +35% faster

integrate_cy.pyx

```
def f(double x):
    return x ** 2 - x


def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s
    cdef double dx
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

In Cython w/ static typing:
+400% faster

=> Leaving Python and static typing can greatly enhance code efficiency

Yellow: Python interaction

If you would like to release the GIL: No yellow lines!!

```
 01:
 02: from libc.math cimport sin
 03:
 04:
 05:
+06: cdef double f(double x) except *:
+07:     return sin(x**2)  # sin from libc
 08:
 09:
+10: def integrate(double a, double b, int N):
 11:     cdef int i
 12:     cdef double s
 13:     cdef double dx
+14:     s = 0
+15:     dx = (b - a) / N
+16:     for i in range(N):
   __pyx_t_2 = __pyx_v_N;
   __pyx_t_3 = __pyx_t_2;
   for (__pyx_t_4 = 0; __pyx_t_4 < __pyx_t_3; __pyx_t_4+=1) {
     __pyx_v_i = __pyx_t_4;
+17:         s += f(a + i * dx)
+18:     return s * dx
```

Pybind11

pybind11

Core features

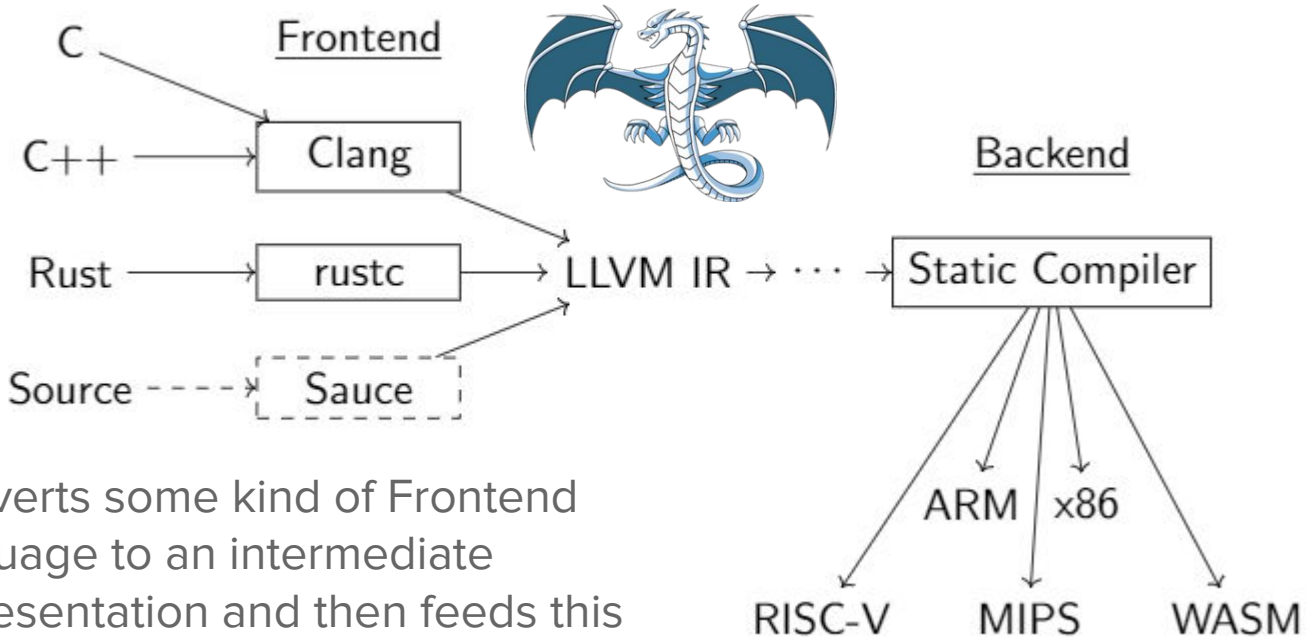
pybind11 can map the following core C++ features to Python:

- Functions accepting and returning custom data structures per value, reference, or pointe
- Instance methods and static methods
- Overloaded functions
- Instance attributes and static attributes
- Arbitrary exception types
- Enumerations
- Callbacks
- Iterators and ranges
- Custom operators
- Single and multiple inheritance
- STL data structures
- Smart pointers with reference counting like `std::shared_ptr`
- Internal references with correct reference counting
- C++ classes with virtual (and pure virtual) methods can be extended in Python

Instant Neural Graphics Primitives with a Multiresolution Hash Encoding

https://github.com/NVlabs/instant-ngp

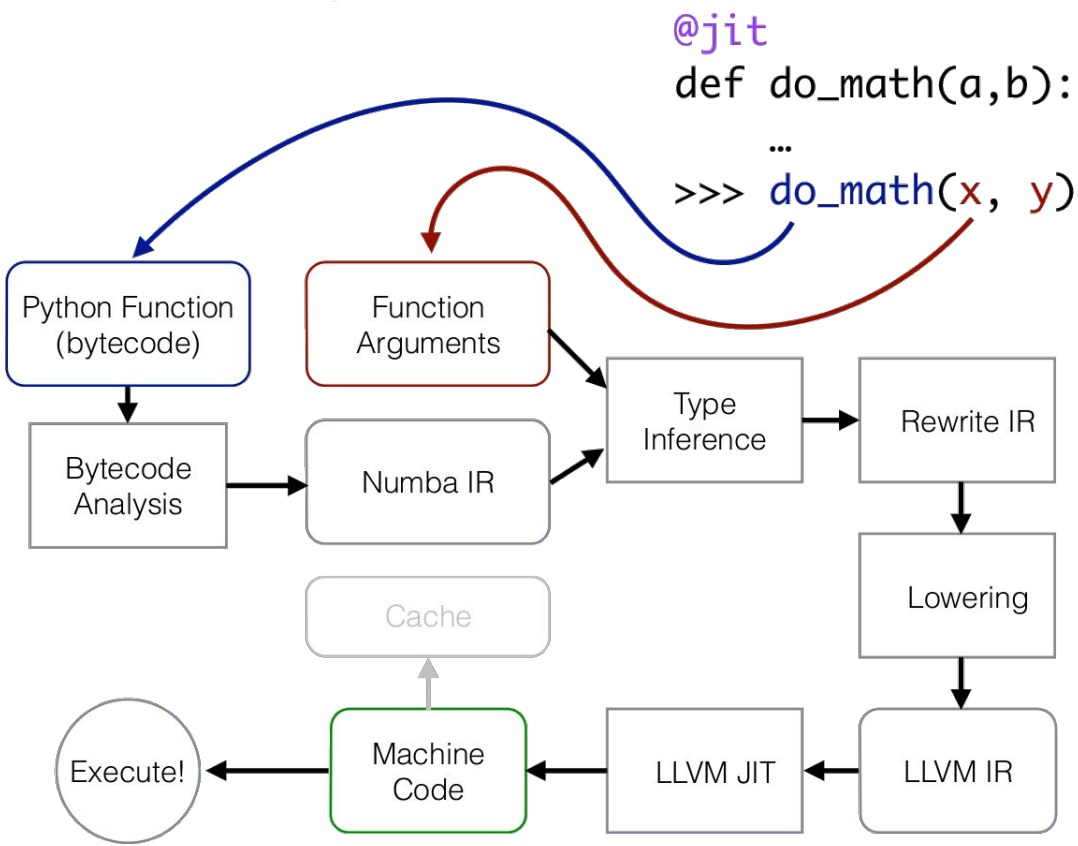
LLVM (low level virtual machine)



Converts some kind of Frontend language to an intermediate representation and then feeds this into any backend architecture

https://github.com/jiachen247/llvm-sauce

Numba



https://medium.com/rapids-ai/the-life-of-a-numba-kernel-a-compilation-pipeline-taking-user-defined-functions-in-python-to-cuda-71cc39b77625


```python
from numba import jit
import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Exiting Python mode

Accelerate Python Functions

Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.

You don't need to replace the Python interpreter, run a separate compilation step, or even have a C/C++ compiler installed. Just apply one of the Numba decorators to your Python function, and Numba does the rest.

Numba usage

```
from numba import jit

@jit
def f(x, y):
    # A somewhat trivial example
    return x + y
```

```
from numba import jit, int32

@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

```
@jit
def square(x):
    return x ** 2

@jit
def hypot(x, y):
    return math.sqrt(square(x) + square(y))
```

Have to use @jit in each function that is used, otherwise ➔ slow

Releasing the GIL:
Ready for concurrency

```
@jit(nogil=True)
def f(x, y):
    return x + y
```

“When using `nogil=True`, you’ll have to be wary of the usual pitfalls of multi-threaded programming (consistency, **synchronization**, **race conditions**, etc.).”

```
@jit(nopython=True, parallel=True)
def f(x, y):
    return x + y
```

Using numba in parallel

```
@jit(nopython=True, parallel=True)
def f(x, y):
    return x + y
```

```
from numba import njit, prange

@njit(parallel=True)
def prange_test(A):
    s = 0
    # Without "parallel=True" in the jit-decorator
    # the prange statement is equivalent to range
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

```
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def two_d_array_reduction_prod(n):
    shp = (13, 17)
    result1 = 2 * np.ones(shp, np.int_)
    tmp = 2 * np.ones_like(result1)

    for i in prange(n):
        result1 *= tmp

    return result1
```

Race conditions

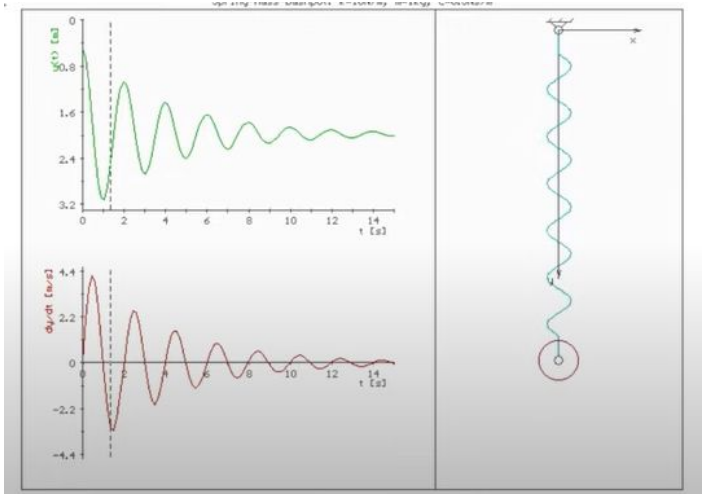
```
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def prange_wrong_result(x):
    n = x.shape[0]
    y = np.zeros(4)
    for i in prange(n):
        # accumulating into the same element of `y` from different
        # parallel iterations of the loop results in a race condition
        y[:] += x[i]

    return y
```

```
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def prange_wrong_result(x):
    n = x.shape[0]
    y = np.zeros(4)
    for i in prange(n):
        # accumulating into the same element of `y` from different
        # parallel iterations of the loop results in a race condition
        y[i % 4] += x[i]

    return y
```

```
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def prange_ok_result_whole_arr(x):
    n = x.shape[0]
    y = np.zeros(4)
    for i in prange(n):
        y += x[i]
    return y
```

```
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def prange_ok_result_outer_slice(x):
    n = x.shape[0]
    y = np.zeros(4)
    z = y[:]
    for i in prange(n):
        z += x[i]
    return y
```

Numba for scientific computing (1000x speed-up)



```
In [55]: %time _ = simulate_spring_mass_funky_damper(1)
CPU times: user 299 ms, sys: 0 ns, total: 299 ms
Wall time: 299 ms
```

njit

```
In [64]: %time _ = simulate_spring_mass_funky_damper(1)
CPU times: user 1.5 ms, sys: 122 µs, total: 1.62 ms
Wall time: 1.63 ms
```

https://www.youtube.com/watch?v=x58W9A2lnQc

```
In [67]: %%time
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(8) as ex:
    ex.map(simulate_spring_mass_funky_damper, np.arange(0.1, 1000
CPU times: user 14.8 s, sys: 229 ms, total: 15.1 s
Wall time: 14.7 s
```

nogil=True

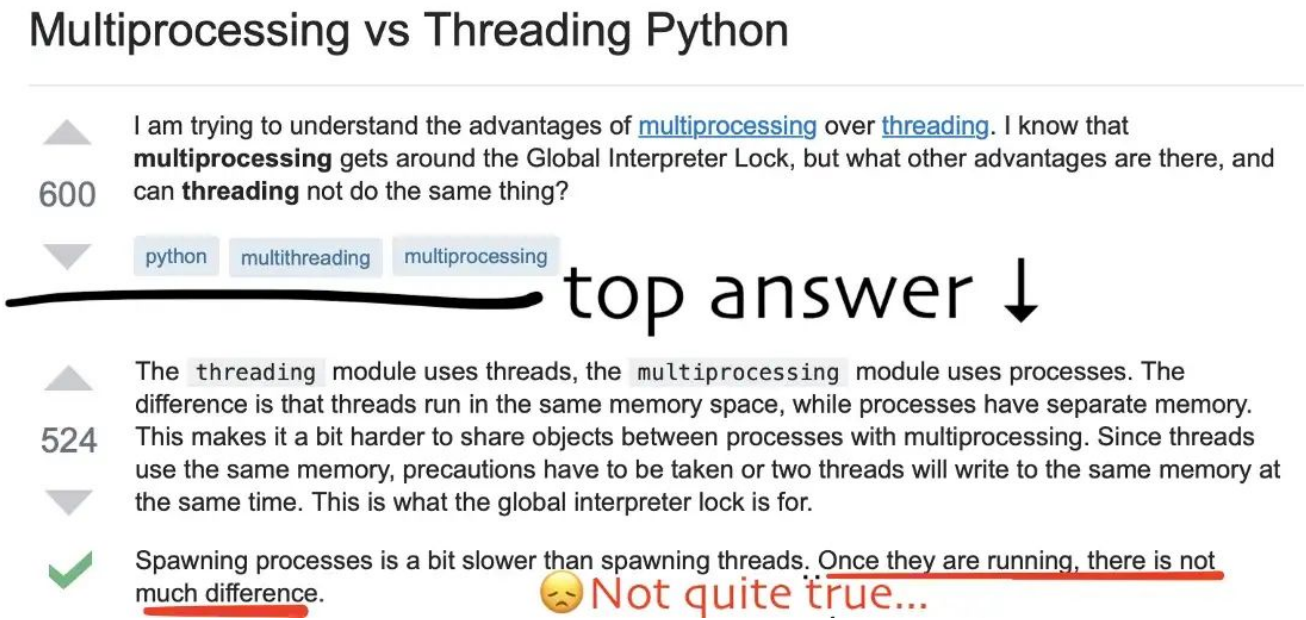
```
In [69]: %%time
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(8) as ex:
    ex.map(simulate_spring_mass_funky_damper, np.arange(0.1, 1000, 0.1))
CPU times: user 13.9 s, sys: 70 ms, total: 13.9 s
Wall time: 2.77 s
```

Native parallel

```
In [73]: from numba import prange

@njit(parallel=True)
def run_sims(end=1000):
    for ii in prange(int(end/0.1)):
        if ii == 0:
            continue
        simulate_spring_mass_funky_damper(ii*0.1)

run_sims(10)
```

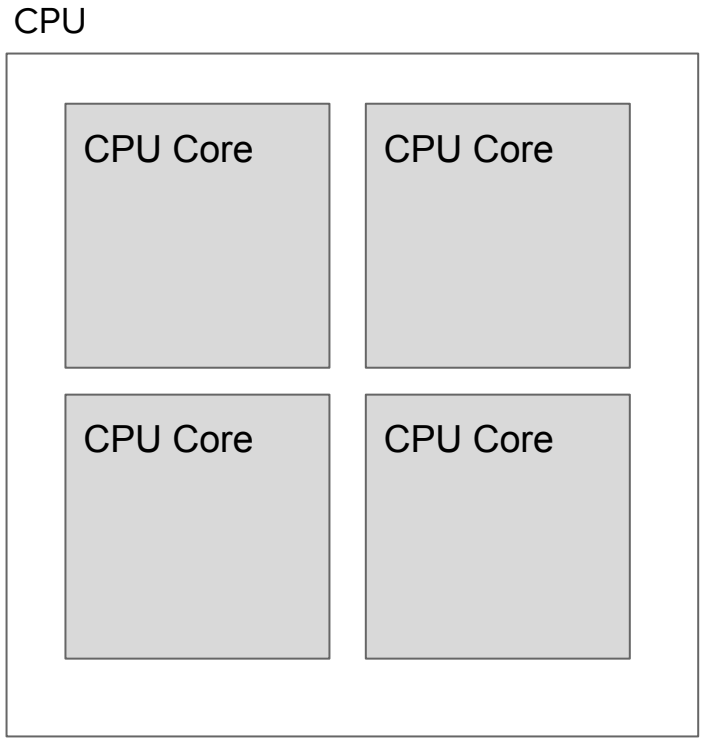
```
In [74]: %time run_sims(1000)
CPU times: user 12.8 s, sys: 84.9 ms, total: 12.9 s
Wall time: 2.05 s
```

Multithreading and multiprocessing is *not* the same!

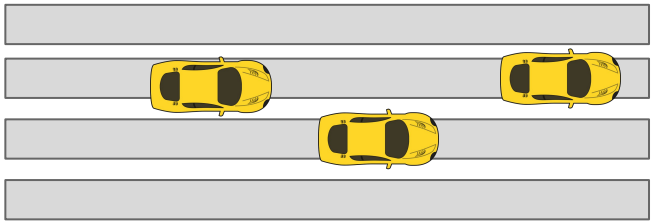


https://medium.com/contentsquare-engineering-blog/multithreading-vs-multiprocessing-in-python-ece023ad55a

CPU, Cores and Threads

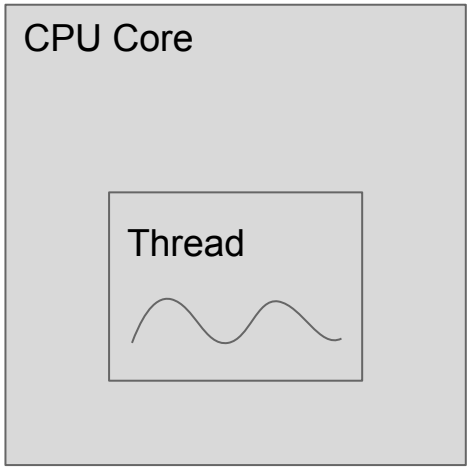


4 lane highway



Clock rate: 3 GHz
3.000.000.000 clock cycles (signal by internal oscillator) / s
Maybe many instructions per cycle, or one instruction over multiple cycles

How does a CPU execute instructions?

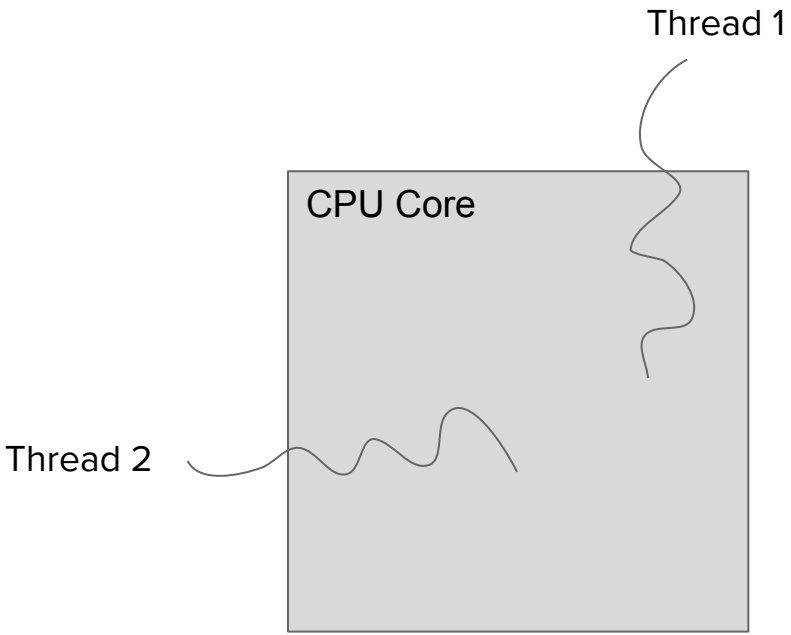


A **thread (of execution)** is the most high-level instruction of a CPU execution cycle.

1. Instruction theme is fetched into CPU's front end via the L1 cache in batches
2. Decoding of instructions into processor's internal instruction format
3. Decoded instructions are dispatched to the execution hardware
4. Execution hardware does arithmetics etc
5. Results are written to the CPU's register
6. New thread? Old thread's state is written to main memory and old thread instructions are removed from pipeline. Start #1 with new thread.

⇒ 1 Thread is able to run simultaenously

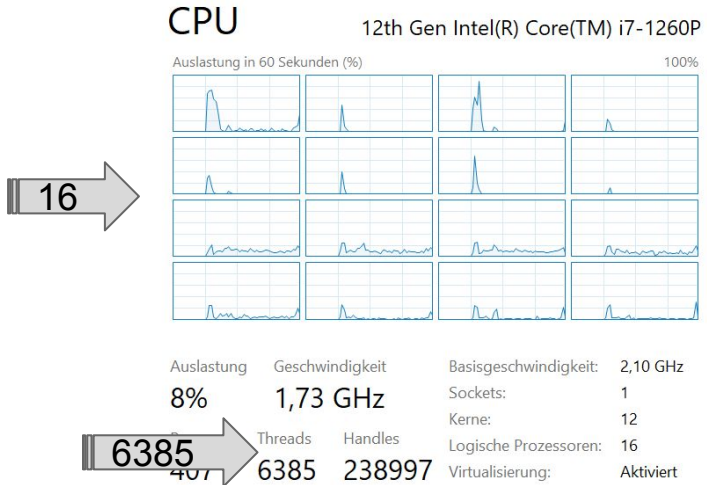
Threading is changing the attention



CPU can only run 1 thread at a time/core.

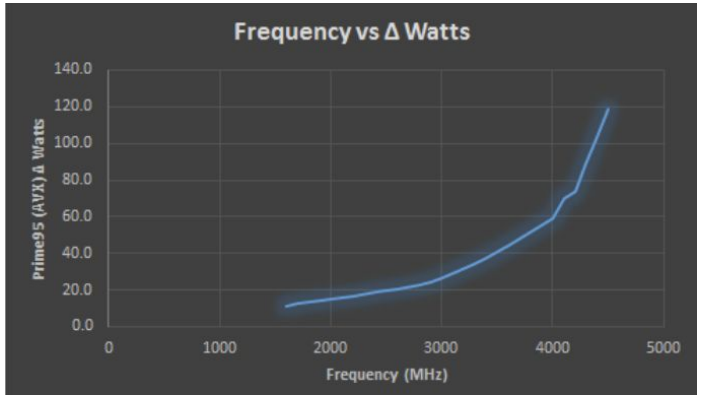
CPU needs to figure out when to run which thread.

If a thread has no urgent reason to be executed (e.g. **waiting** for a response from the user or some data to be loaded), it is in **idle** mode and another thread can run



Ways to go faster

Increase the clock frequency of CPUs!

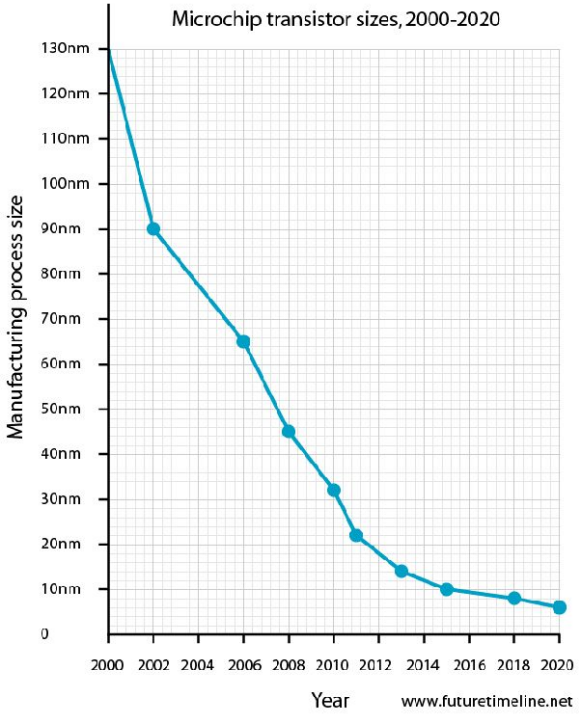


https://forums.tomshardware.com/threads/higher-clock-speed-vs-higher-load-power-consumption.2652062/

⇒ Power draw is becoming very large!

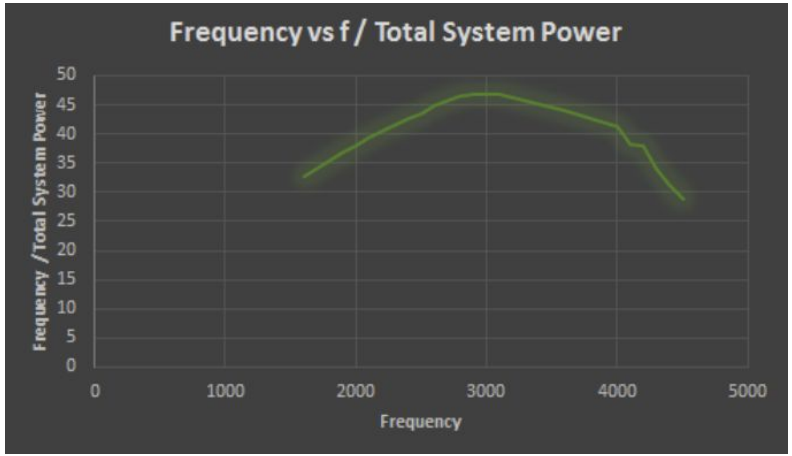
Optimize TDP to allow longer battery life times!

What is with the transistors?



Transistors became SMALLER, but NOT FASTER!

Sweet spot of efficiency



https://forums.tomshardware.com/threads/higher-clock-speed-vs-higher-load-power-consumption.2652062/

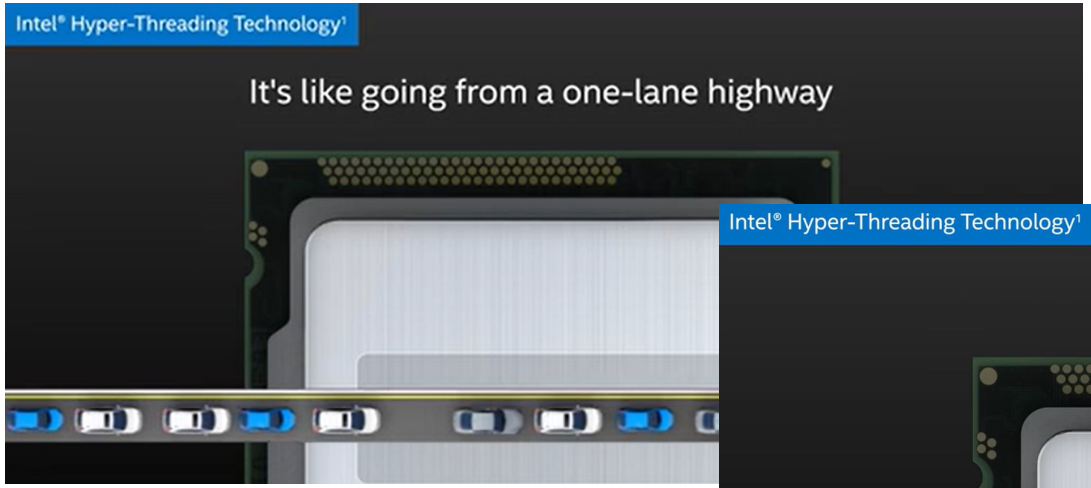
⇒ multiple Cores allow also multiple threads!

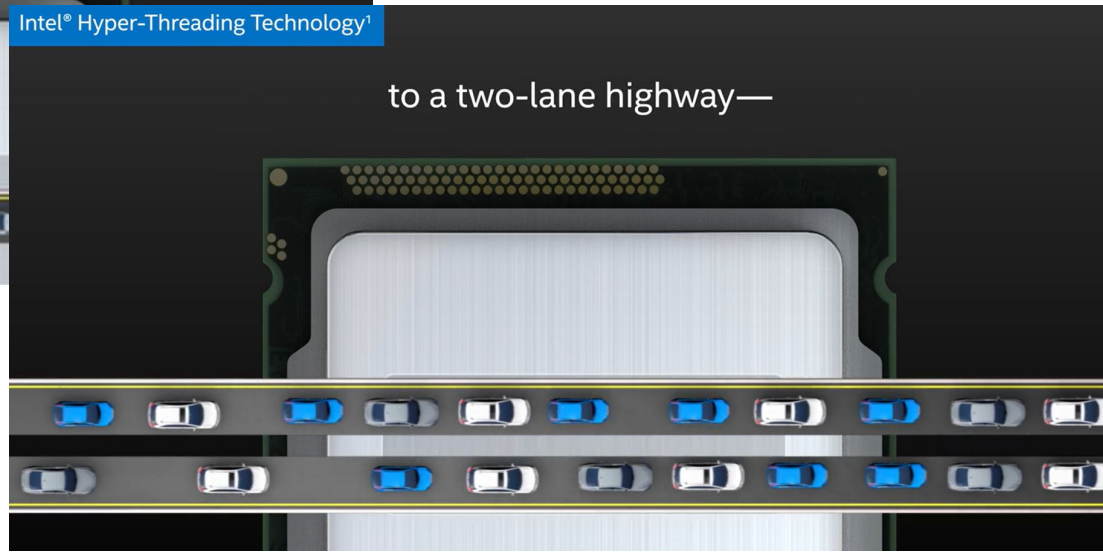
IBM introduced dual cores in its Power 4 chips in 2000. In 2004, Sun and HP introduced their first dual core CPUs.

The first dual core chips for x86-based PCs and servers were introduced in 2005 and included the Pentium D and Pentium Processor Extreme Edition 840 from Intel and the Opteron 800 and Athlon 64 X2 from AMD. A year later, Intel added dual cores to its Itanium line.

https://www.pcmag.com/encyclopedia/term/dual-core#:~:text=The%20first%20dual%20core%20chips,Athlon%2064%20X2%20from%20AMD.

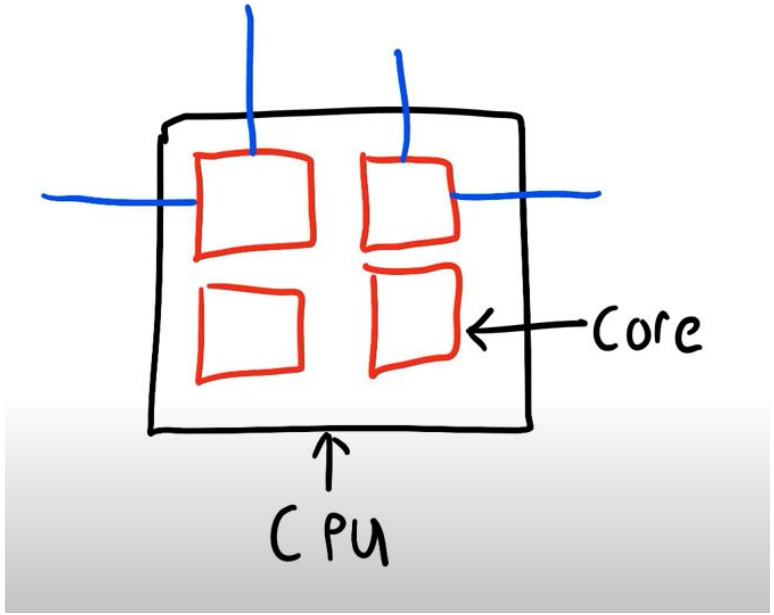
Simultaneous multithreading (SMT)/ Hyperthreading (HT)





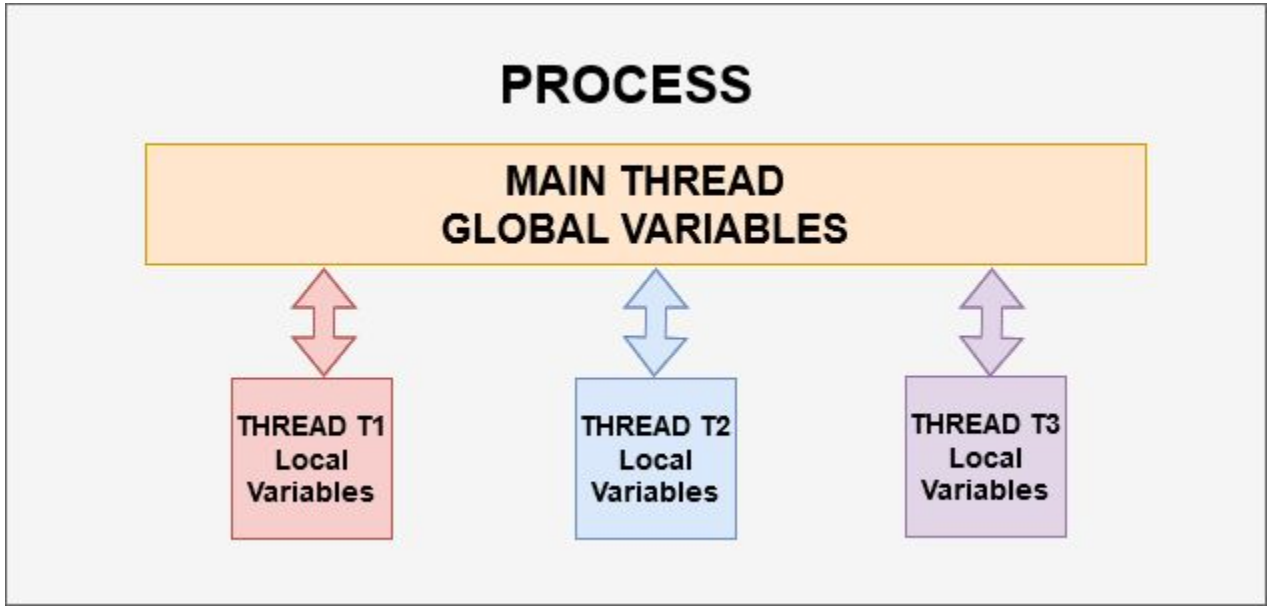
CPU creates for each core a second, virtual core. Real and virtual core can optimize instructions are therefore faster. 5% more die surface, 15-30% faster

Threads

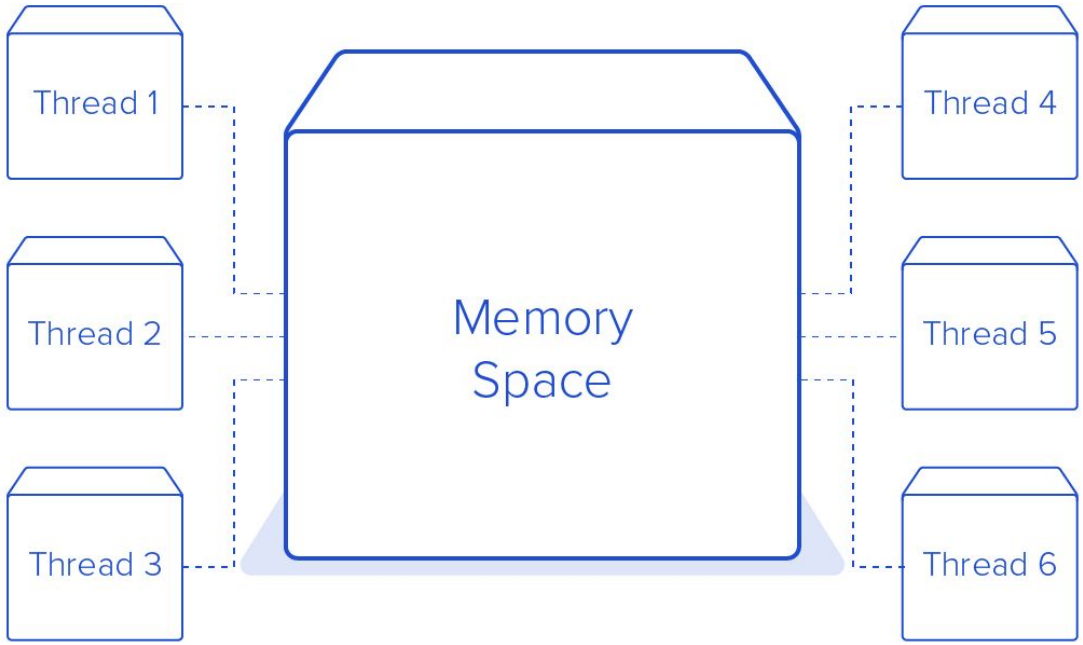


https://www.youtube.com/watch?v=olYdb0DdGtM

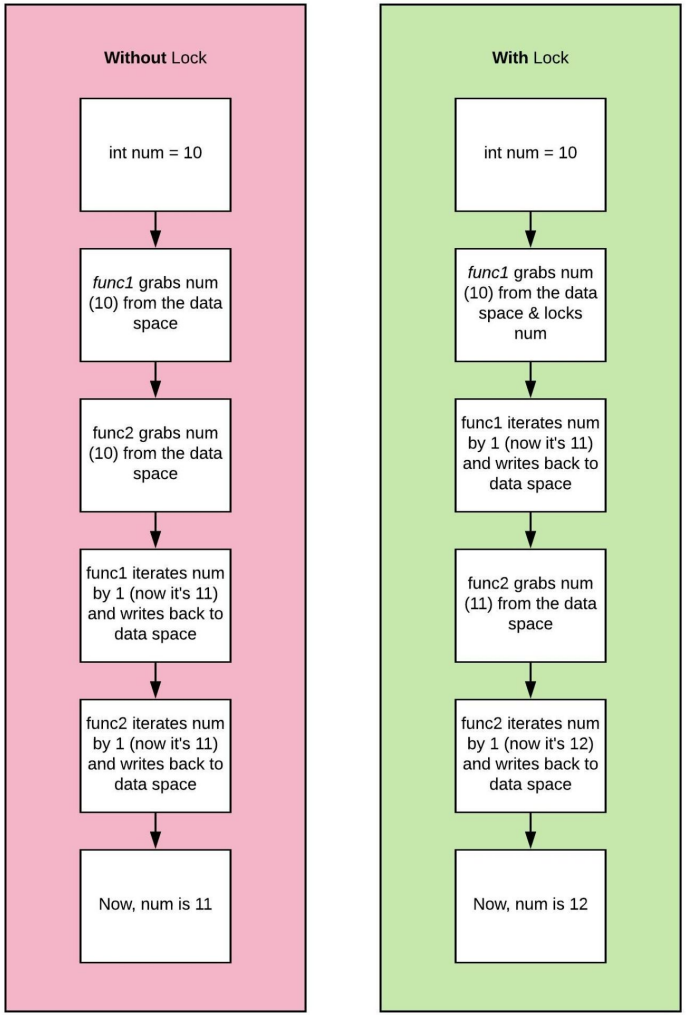
Every Python program has at least 1 thread



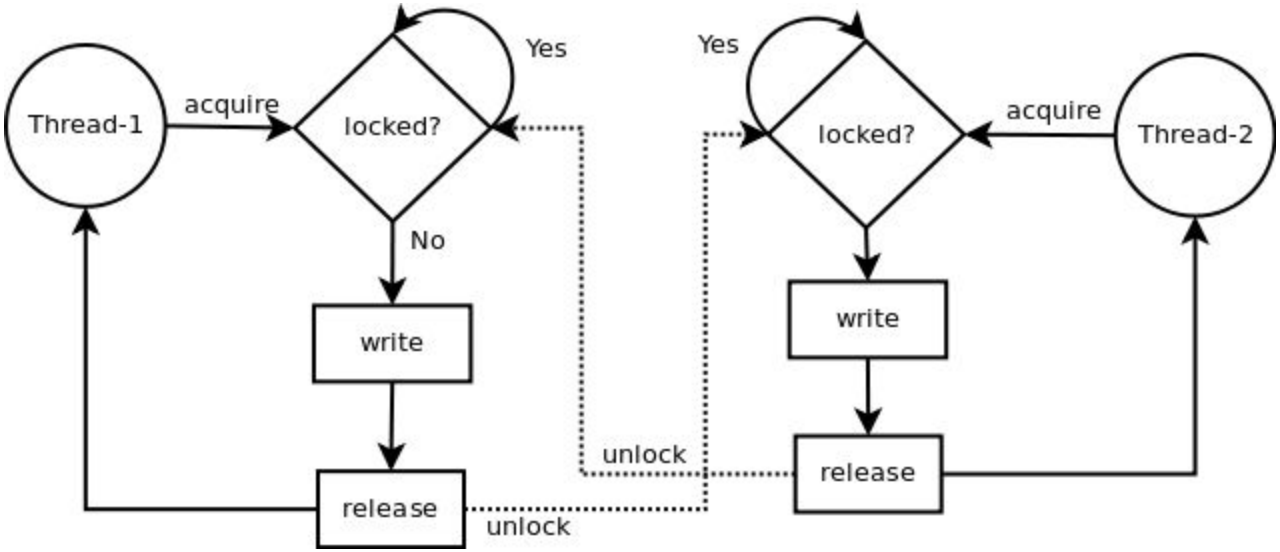
Memory access



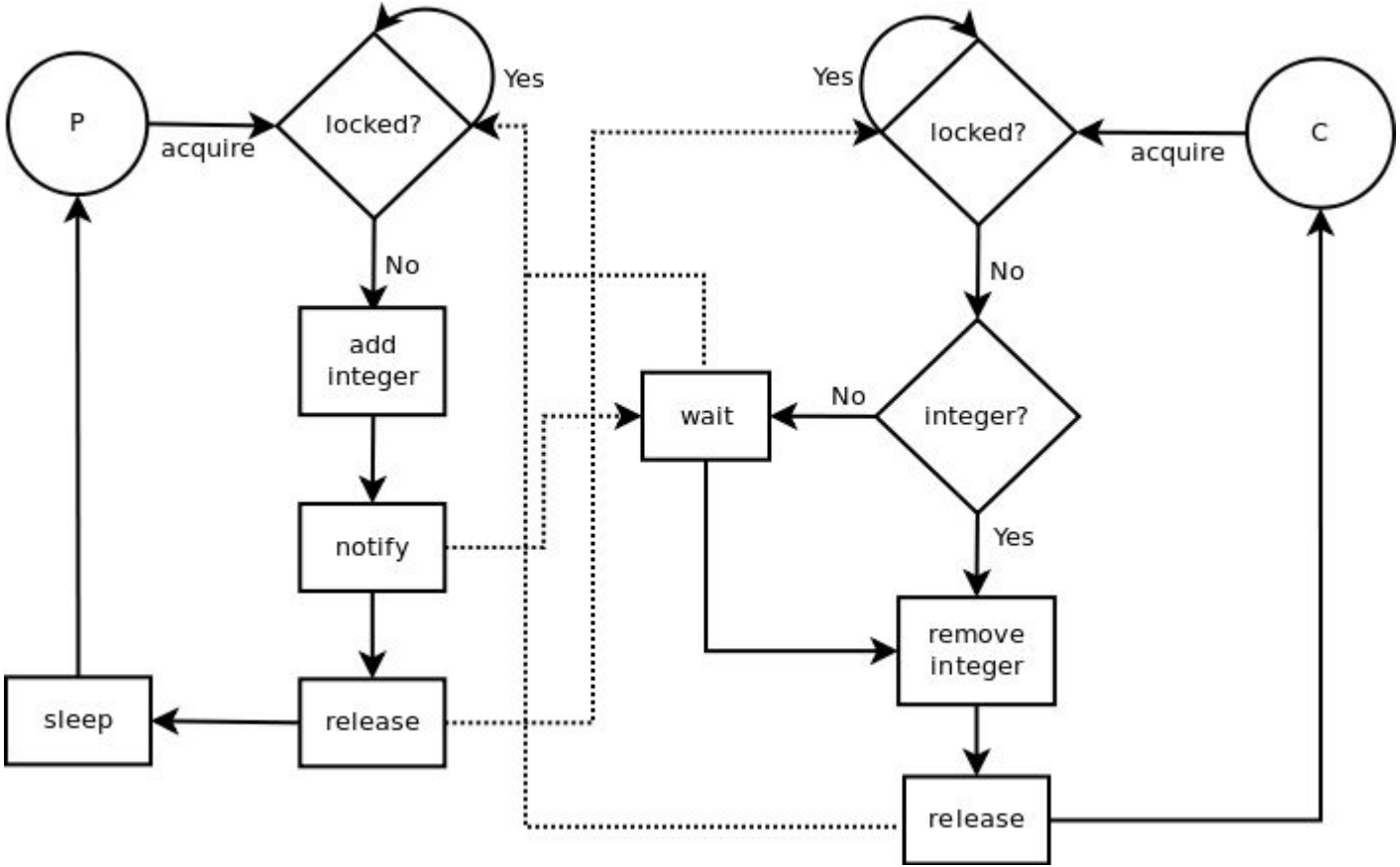
Race conditions



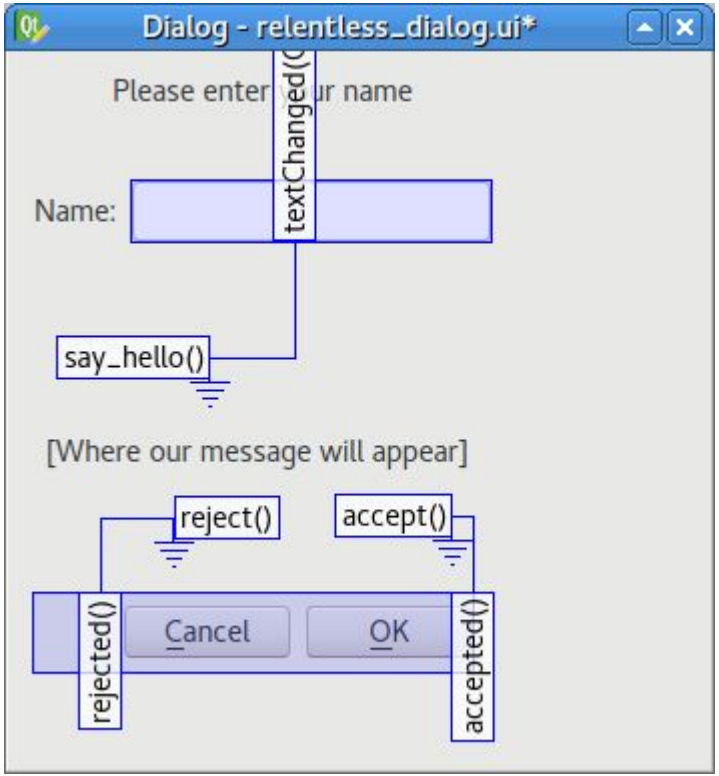
Synchronizing threads

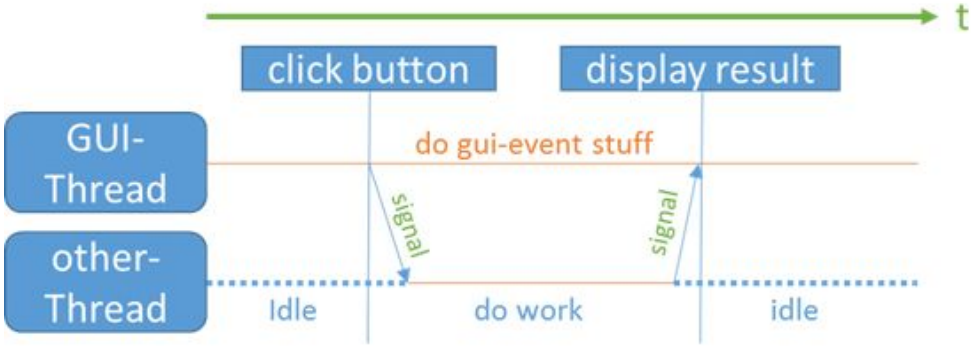


Talking to threads: conditioning



PyQt Signal/Slots



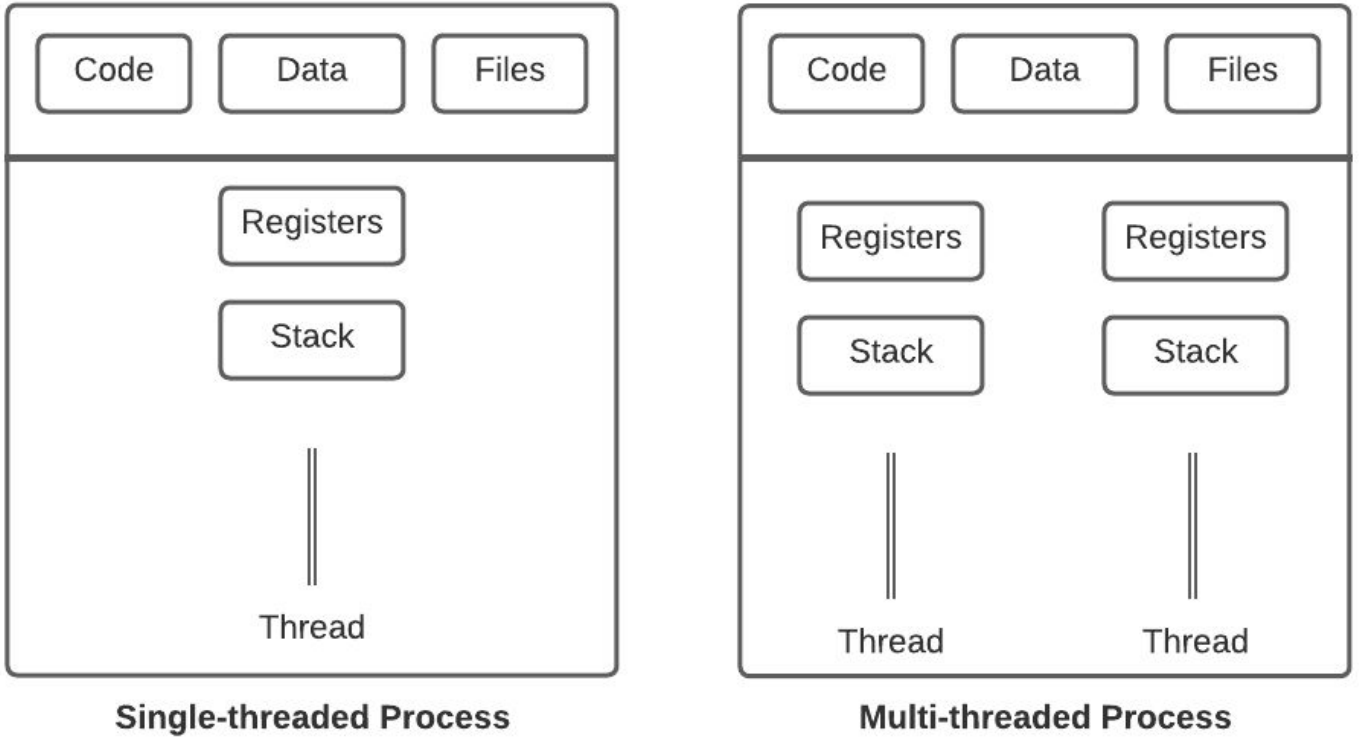


Why would we use threads?

- GUIs (not to hang main GUI thread when sth. Is being computed)
- I/O bound problems
- Parallelize data processing on shared data and race conditions can be (easily?) avoided

What is a process then?

https://www.baeldung.com/cs/process-vs-thread



Processes have their own memory space, but are more expensive to spawn than a thread.

Process/thread side-by-side comparison

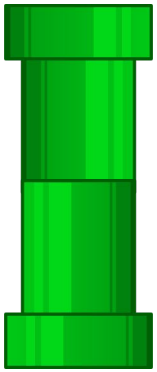
3.3. Key Differences Between a Process and a Thread

	Process	Thread
Definition	A process is the execution of a program.	A thread is a semi-process.
Creation	We need to use more than one system call to create more than one process.	We can create more than one thread with one system call.
Termination	Termination of the process take more compared to thread.	Termination of thread takes less compared to thread.
Communication	It requires extra mechanisms such as IPC.	It does not require any extra mechanism.
Context switching	Context switching of processes slower than threads.	Context switching between threads is much faster than processes.
Resource	Processes may consume more resources since they have separate memory spaces.	Threads may consume fewer resources.
Memory	Processes are mostly isolated.	Threads share memory.
Sharing	It requires extra mechanisms such as IPC to share data.	Threads share data with each other.

IPC = Inter-process communication

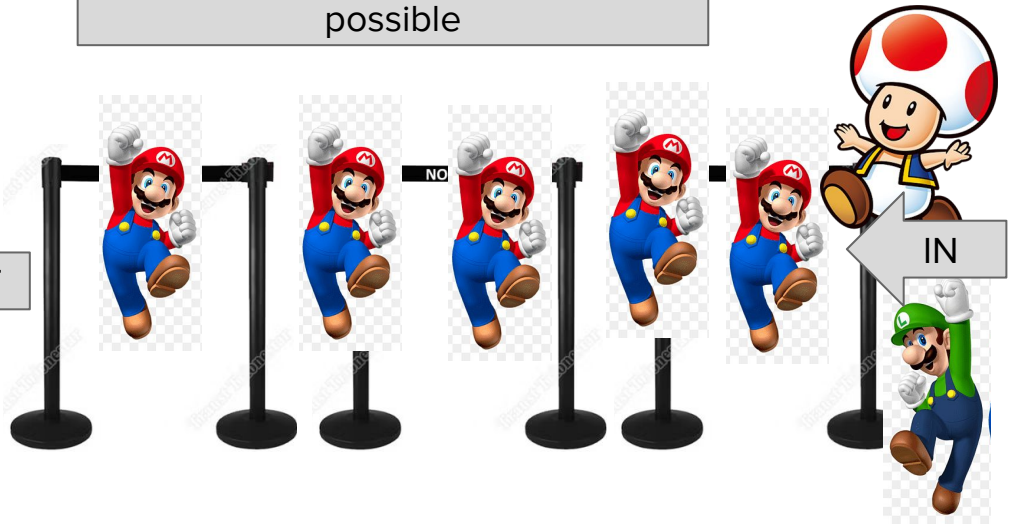
https://www.baeldung.com/cs/process-vs-thread

Inter-process communication



Pipe
(both can send and receive = duplex)

Queue: FIFO-principle; multiple consumer and producer easily possible

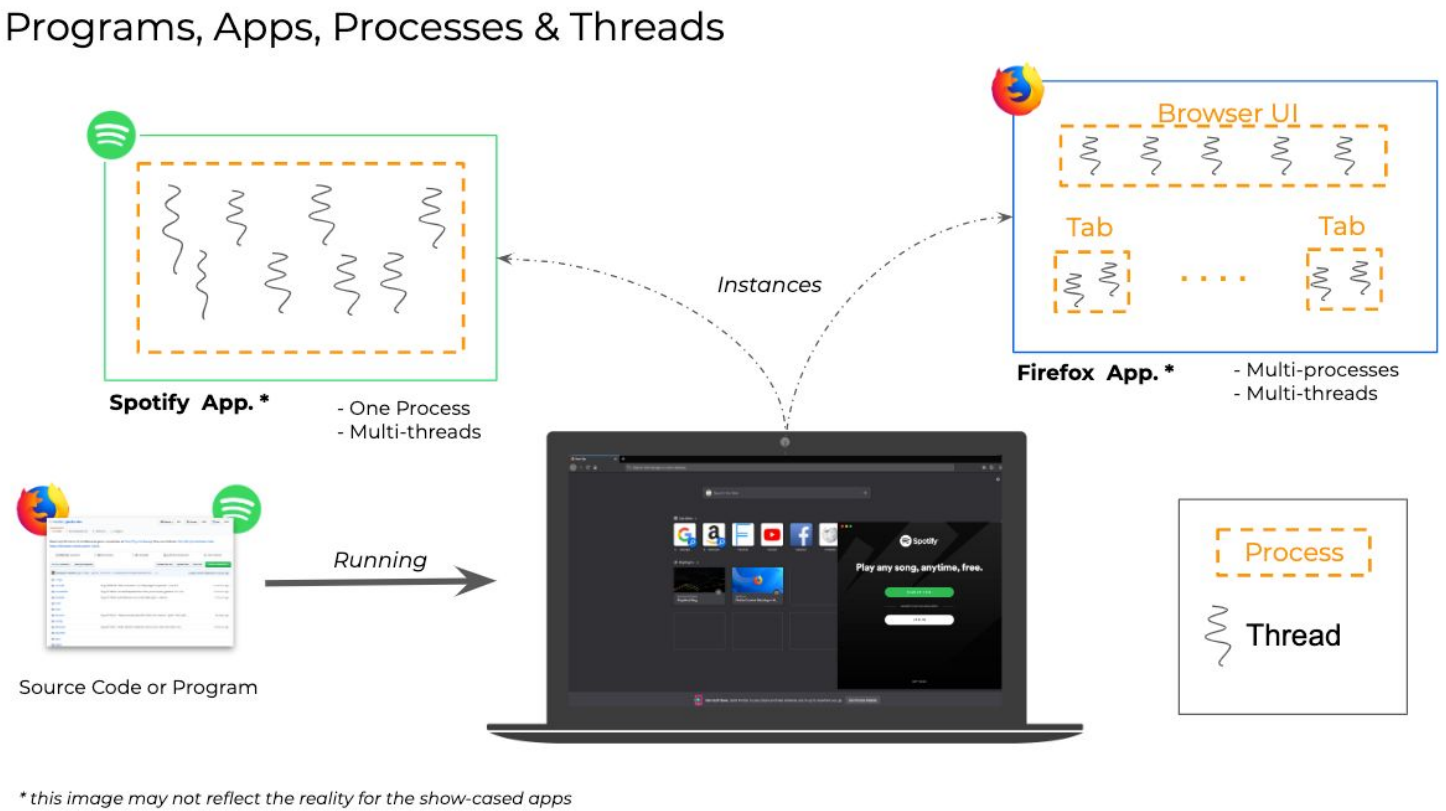


https://slideplayer.com/slide/6250227/

Overview pipes vs. queues

Aspect	multiprocessing.Pipe	multiprocessing.Queue
Basic Concept	A low-level, two-ended connection (by default, full-duplex) for sending and receiving data directly between two processes.	A higher-level FIFO (first-in-first-out) queue abstraction built on top of pipes and locks.
Data Flow Direction	By default, creates a pair of connection objects. Each end can both send and receive unless duplex=False is specified.	Typically used in a unidirectional manner: data is put in by producers and retrieved by consumers. Supports multiple producers and consumers.
Multiple Producers/Consumers	Not straightforward for multiple producers and consumers. Each Pipe endpoint is typically managed by one process on each side.	Designed for easy multiple-producer, multiple-consumer usage. Each process can safely call put() and get() with built-in synchronization.
Buffering and Storage	Does not inherently buffer multiple messages like a queue; once a message is sent, the other side should read it in a timely way.	Internally buffers items. It can store multiple items, so producers can continue putting objects without waiting for consumers to read them.
Synchronization & Complexity	Lower-level approach where you manually handle send/recv semantics. Less overhead but more do-it-yourself synchronization.	Higher-level abstraction that handles synchronization internally. Simpler for producer-consumer patterns, but has slightly more overhead.
Typical Use Cases	Direct, point-to-point communication between exactly two processes. Suited for two-way (full-duplex) communication.	Producer-consumer scenarios. Multiple processes feeding data to one aggregator process (or vice versa) without extra plumbing.
Performance & Overhead	Lower overhead for a direct, single-producer/single-consumer setup because there is minimal internal machinery.	Slightly higher overhead due to locks and a shared buffer. Scales well in multi-producer, multi-consumer scenarios.
Ease of Use	Requires careful handling of endpoints and explicit send/receive calls.	Very straightforward for enqueuing (put()) and dequeuing (get()) data with built-in concurrency safeguards.

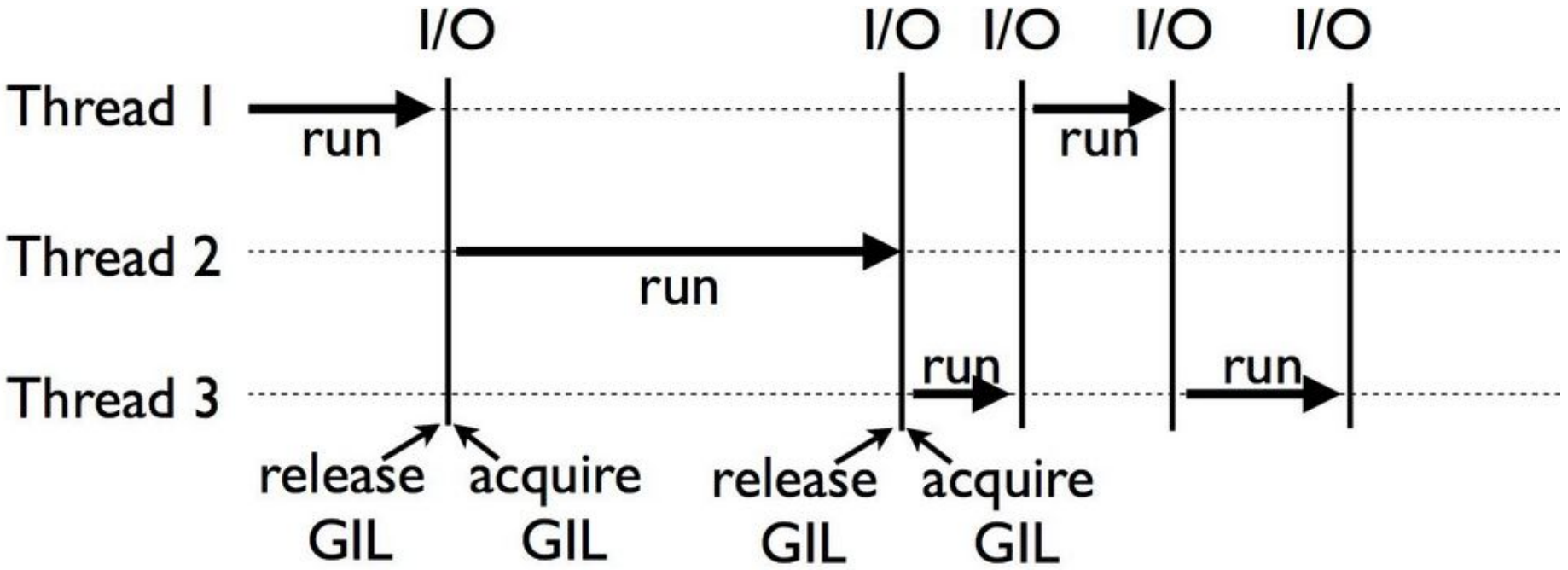
Overview of processes and threads



** this image may not reflect the reality for the show-cased apps*

The Global Interpreter Lock (GIL)

This is especially specific to Python!!



GIL

From the Python wiki:

> *In CPython, the **global interpreter lock**, or **GIL**, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe.*

Multiprocessing vs. multithreading with GIL

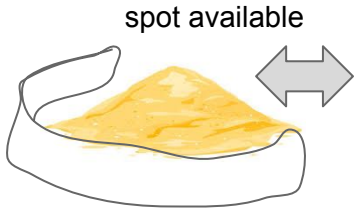
Multiprocessing

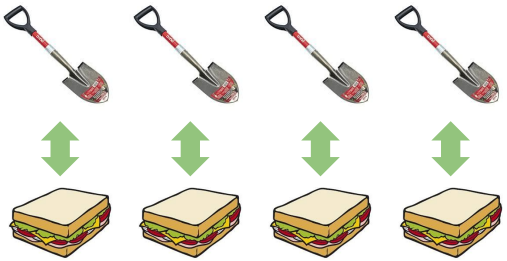
Job

Workers

Multithreading

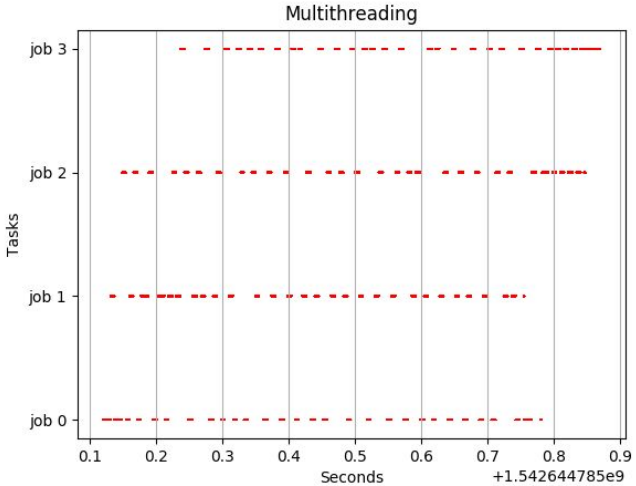
only **1** worker
spot available



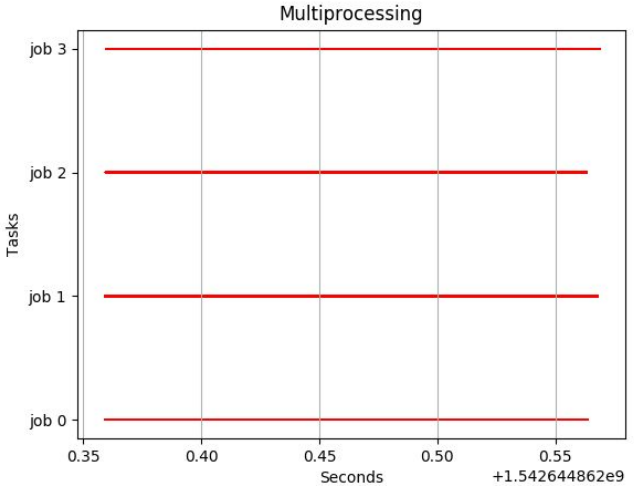


Multiprocessing vs. multithreading

Threads run “concurrently”!



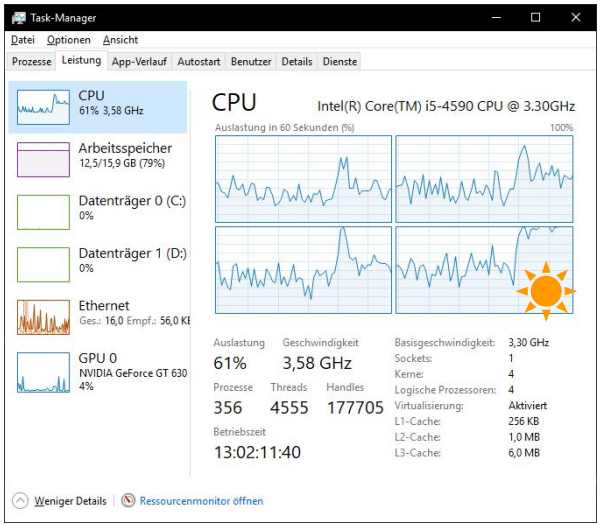
Processes run parallel!



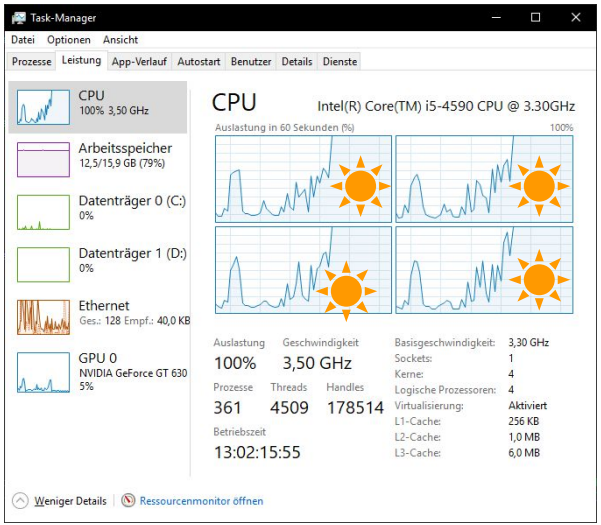
https://medium.com/contentsquare-engineering-blog/multithreading-vs-multiprocessing-in-python-ece023ad55a

CPU Heavy Task (GIL active)

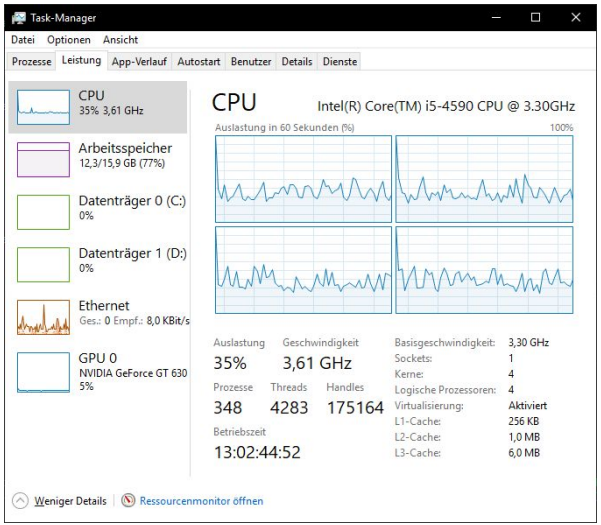
1 core (Python default)



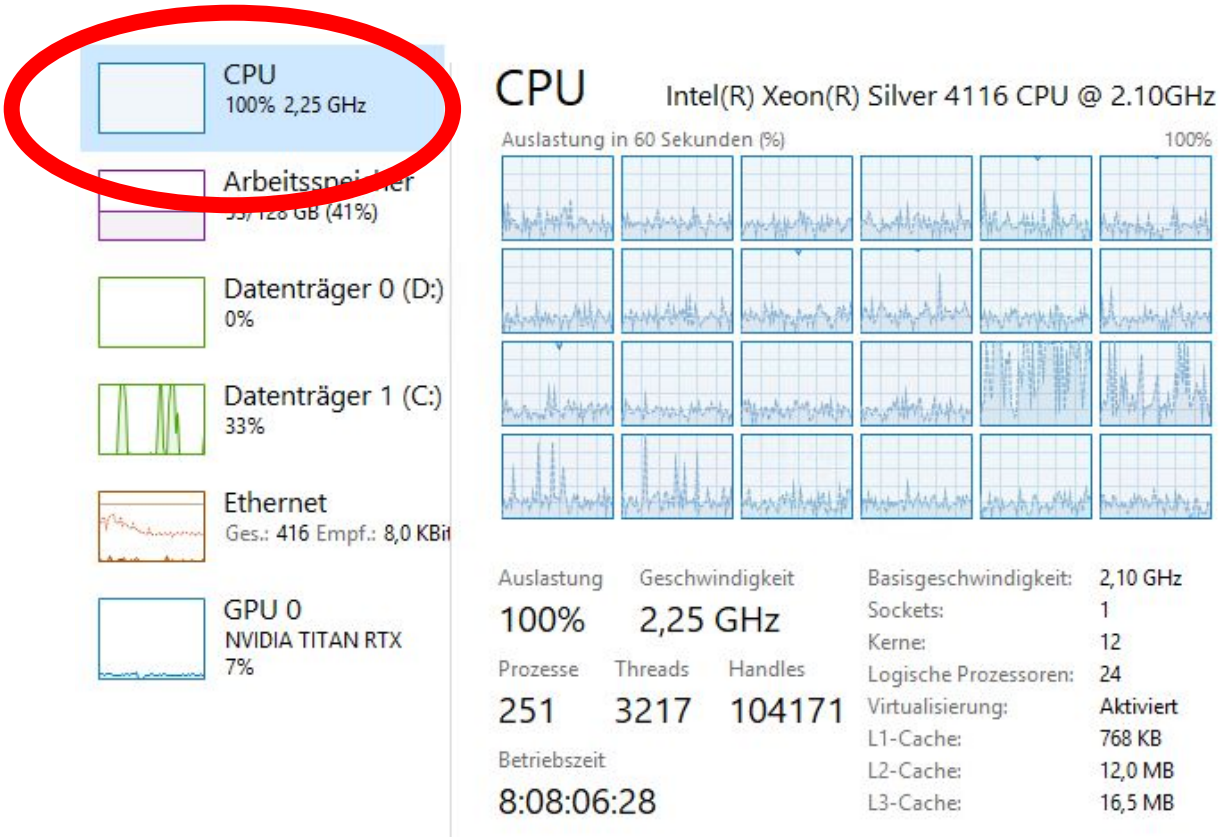
4 cores (multiprocessing)



8 threads (multithreading)



CPU Heavy Task - on a decent workstation



Killing a thread

- Not (really) possible
- Killing a process: possible.

There are workarounds to kill a thread, but these come with slowness (up to 10x lower execution speed), are not memory safe, etc...

➜ Processes killing is easy

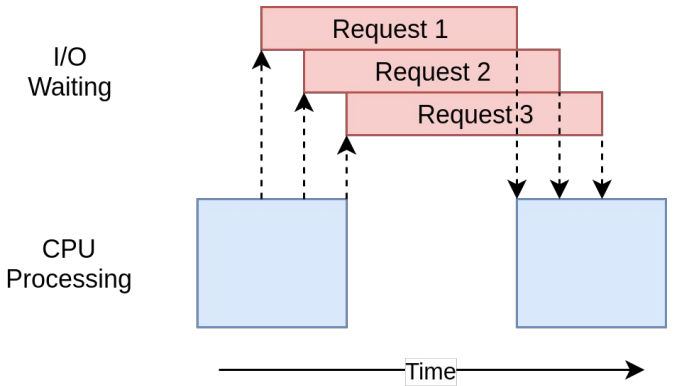
Check before what you need

asyncio

Python

```python
import asyncio
import time
import aiohttp


async def download_site(session, url):
    async with session.get(url) as response:
        print("Read {0} from {1}".format(response.content_length, url))


async def download_all_sites(sites):
    async with aiohttp.ClientSession() as session:
        tasks = []
        for url in sites:
            task = asyncio.ensure_future(download_site(session, url))
            tasks.append(task)
        await asyncio.gather(*tasks, return_exceptions=True)


if __name__ == "__main__":
    sites = [
        "https://www.jython.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
    asyncio.get_event_loop().run_until_complete(download_all_sites(sites))
    duration = time.time() - start_time
    print(f"Downloaded {len(sites)} sites in {duration} seconds")
```

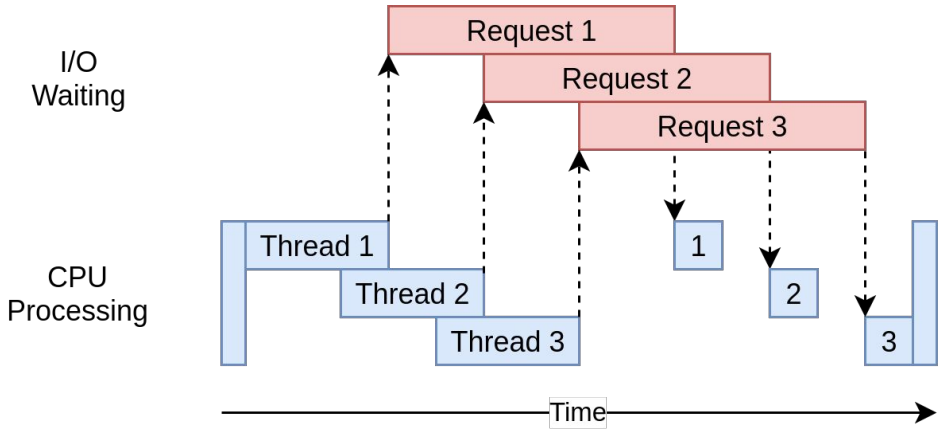
- Runs in 1 Thread
- Unclear what the optimal number of threads is? Asyncio scales better than threads
- Libraries need to support asyncio to leverage full potential
- Python 3.7+ have better asyncio refactoring



Code threads

Python

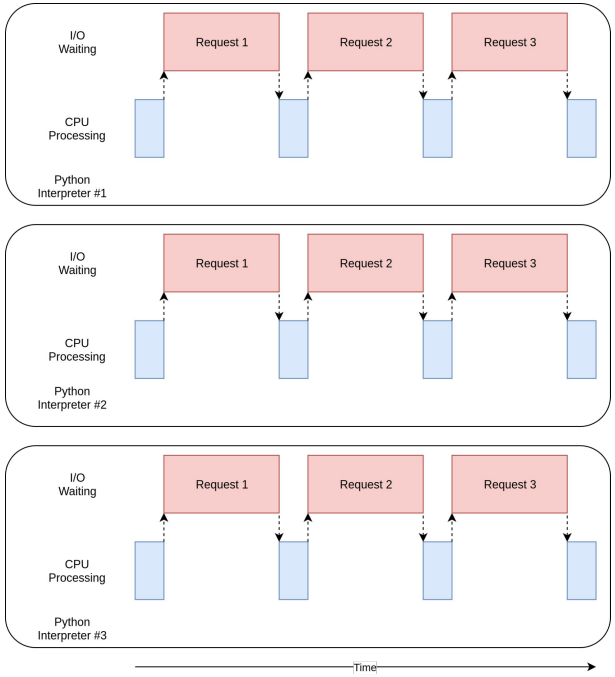
```python
import concurrent.futures
import requests
import threading
import time


thread_local = threading.local()


def get_session():
    if not hasattr(thread_local, "session"):
        thread_local.session = requests.Session()
    return thread_local.session


def download_site(url):
    session = get_session()
    with session.get(url) as response:
        print(f"Read {len(response.content)} from {url}")


def download_all_sites(sites):
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        executor.map(download_site, sites)


if __name__ == "__main__":
    sites = [
        "https://www.jython.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
```



Code processes (using multiprocessing)

Python

```python
import requests
import multiprocessing
import time

session = None


def set_global_session():
    global session
    if not session:
        session = requests.Session()


def download_site(url):
    with session.get(url) as response:
        name = multiprocessing.current_process().name
        print(f"{name}:Read {len(response.content)} from {url}")


def download_all_sites(sites):
    with multiprocessing.Pool(initializer=set_global_session) as pool:
        pool.map(download_site, sites)


if __name__ == "__main__":
    sites = [
        "https://www.jython.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
    download_all_sites(sites)
    duration = time.time() - start_time
    print(f"Downloaded {len(sites)} in {duration} seconds")
```



When to apply multiprocessing...

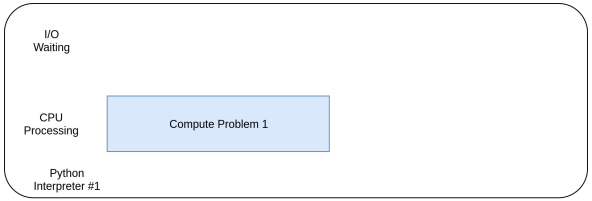
I/O
Waiting

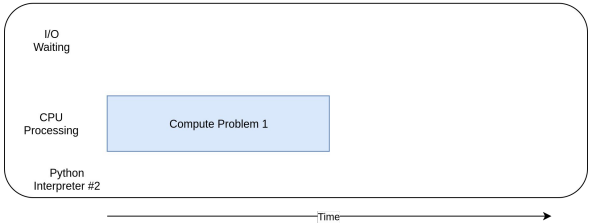
CPU
Processing

Compute Problem 1

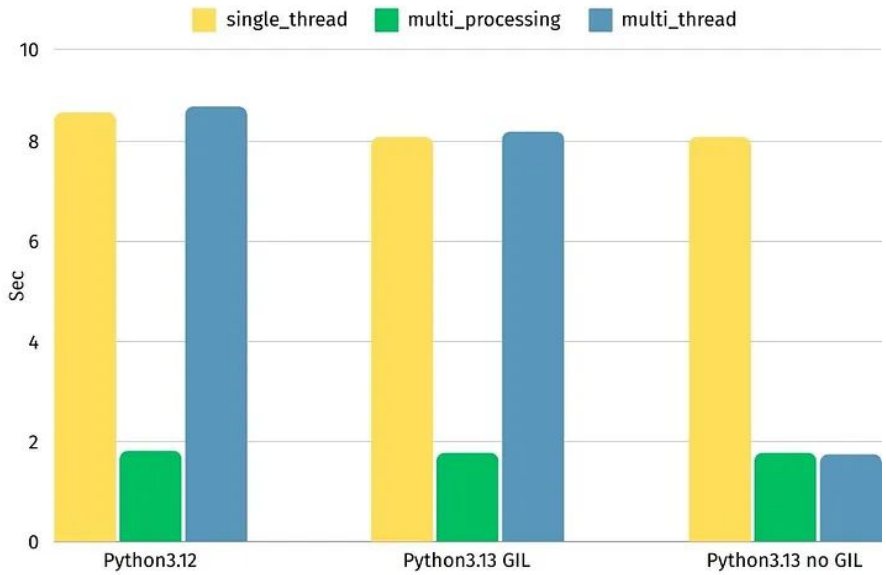
Compute Problem 2

Time





Python 3.13 - no GIL anymore?



But others use threading no? Yes, in C internally (e.g. numpy, torch, TF)

Why not just using processes? Starting process: 50 ms vs. 100 µs (thread)

PEP 684 – A Per-Interpreter GIL

Author: Eric Snow <ericsnowcurrently at gmail.com>
Discussions-To: Discourse thread
Status: Final
Type: Standards Track
Requires: 683
Created: 08-Mar-2022
Python-Version: 3.12
Post-History: 08-Mar-2022, 29-Sep-2022, 28-Oct-2022
Resolution: Discourse message

▸ Table of Contents

PEP 703 – Making the Global Interpreter Lock Optional in CPython

Author: Sam Gross <colesbury at gmail.com>
Sponsor: Łukasz Langa <lukasz at python.org>
Discussions-To: Discourse thread
Status: Accepted
Type: Standards Track
Created: 09-Jan-2023
Python-Version: 3.13
Post-History: 09-Jan-2023, 04-May-2023
Resolution: 24-Oct-2023

Homework

In this week's homework, we will be looking at how to make your code faster applying the concepts of multiprocessing and multithreading.