Overview of parallel and asynchronous computing in Python

Jakub Klinkovský

Czech Technical University in Prague

Faculty of Nuclear Sciences and Physical Engineering

Department of Software Engineering + Department of Mathematics

Workshop on Scientific Computing 2023

May 26 2023

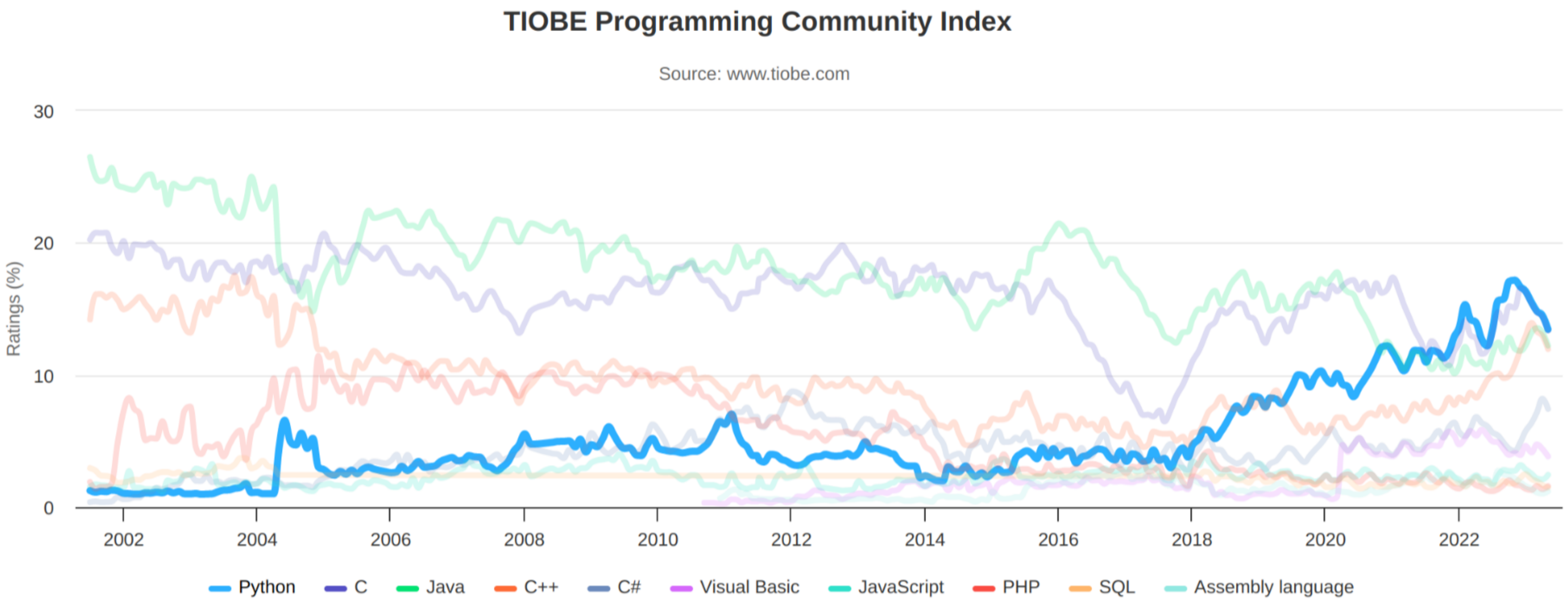
Introduction to Python

Python is general, high-level, dynamic, interpreted programming language.

There are several implementations (interpreters), e.g. CPython, PyPy, IronPython, Jython, etc.

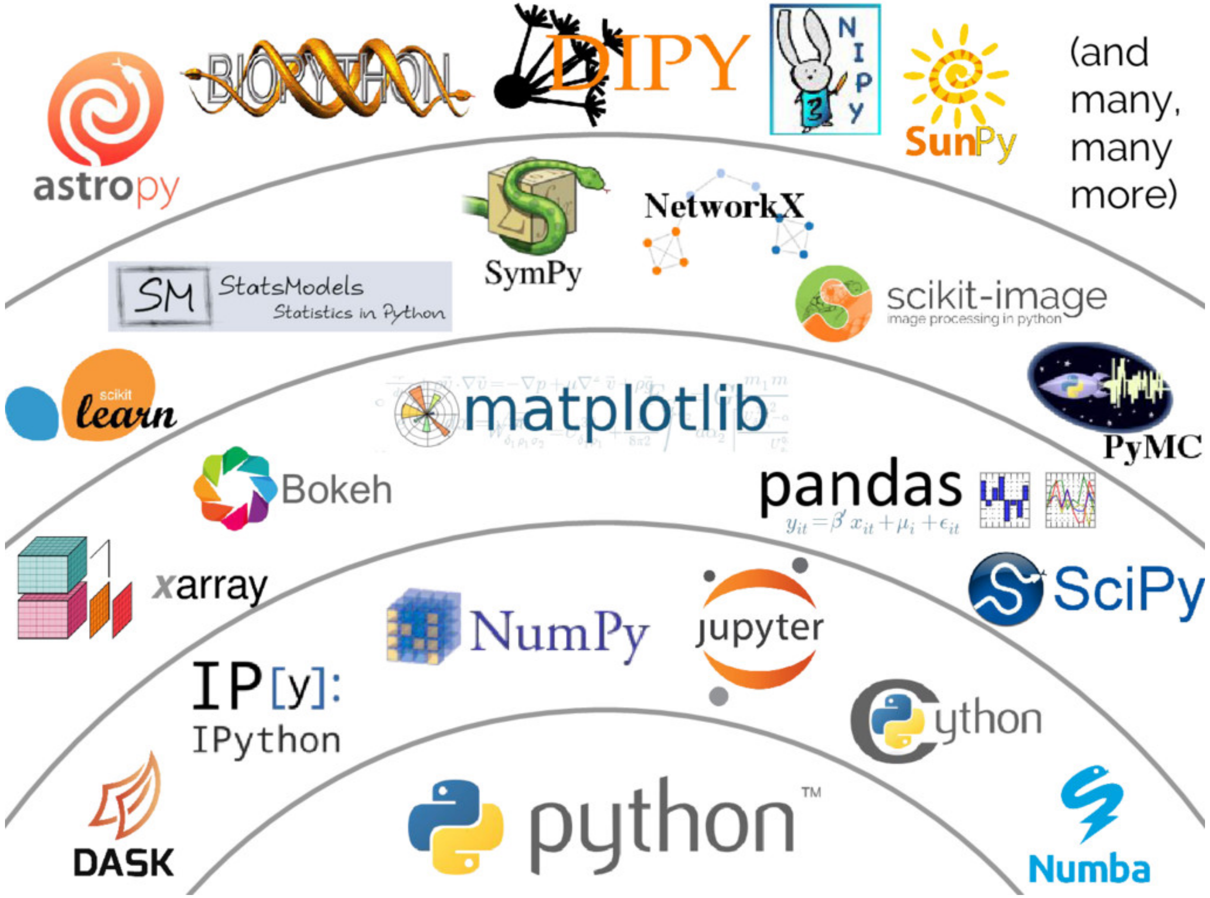
The reference implementation, *CPython*, is considered for this talk.

Most popular programming languages



Python modules for scientific computing

- **Data processing:**
  NumPy, SciPy, Pandas
- **Visualization:**
  Matplotlib, Seaborn, Bokeh, Plotly, Mayavi
- **Machine learning and neural networks:** Keras, SciKit-Learn, PyTorch, TensorFlow
- and many more...



Python modules for improving performance

As Python is an interpreted language, its performance is not great. Typical code optimization approaches are:

- static and just-in-time compilers: NumExpr, Numba, Cython
- reimplementing performance-critical parts in C/C++ and binding to Python (pybind11, Boost.Python)

Even with optimized code, some work loads may benefit from parallelism:

- via processes or via threads
- in high-level Python code or in low-level code (modules used in Python)

Multi-processing and multi-threading modules

1. Multi-processing:
    - separate instances of the Python interpreter that communicate with each other
    - modules such as `multiprocessing`, `mpi4py`, `loky`
2. Multi-threading:
    - **common pitfall:** the CPython interpreter has a Global Interpreter Lock (GIL)
    - low-level code can explicitly release the GIL when the code execution does not involve the Python interpreter
    - even using the `threading` module in Python may be beneficial (e.g. waiting for I/O can be hidden, but **Python code execution is serialized**)

Asynchronous computing in Python

High-level API for asynchronous IO has been developed between Python 3.4 and 3.7:

- two new Python keywords: `async` and `await` (definition of coroutines)
- standard module `asyncio` (API for running and managing coroutines)

Hello World example:

```python
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

Running coroutines

1. `asyncio.run()` – top-level entry point for running an `async` function from the *synchronous context*
2. awaiting on coroutines – using the keyword `await` in an `async` function
   - in general, there are 3 types of *awaitable* objects:
     - **coroutines** – Python function marked with `async`
     - **Tasks** – wrapper object used to schedule coroutines to run concurrently
     - **Futures** – special low-level objects that represent an eventual result that will arrive in the future
   - awaiting allows to express concurrency in the high-level language

Example

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")
    await say_after(1, 'hello')
    await say_after(2, 'world')
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Expected output:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

Running coroutines

3. `asyncio.create_task()` – run coroutines concurrently as *tasks*

Example

```
async def main():
    task1 = asyncio.create_task( say_after(1, 'hello') )
    task2 = asyncio.create_task( say_after(2, 'world') )

    print(f"started at {time.strftime('%X')}")
    await task1
    await task2
    print(f"finished at {time.strftime('%X')}")
```

Now the code runs 1 second faster (tasks are overlapped at `asyncio.sleep(delay)` in the `say_after()` function).

Running coroutines

4. `asyncio.TaskGroup` class – a modern alternative to `asyncio.create_task()` (since Python 3.11)

Example

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task( say_after(1, 'hello') )
        task2 = tg.create_task( say_after(2, 'world') )

        print(f"started at {time.strftime('%X')}")

    # The await is implicit when the context manager exits.

    print(f"finished at {time.strftime('%X')}")
```

Synchronization between tasks

The `asyncio` API provides synchronization primitives similar to `threading` module:

- `asyncio.Lock`
- `asyncio.Event`
- `asyncio.Condition`
- `asyncio.Semaphore`
- `asyncio.BoundedSemaphore`
- `asyncio.Barrier`

Event loop

- `asyncio.run()` runs a low-level *event loop* where all async tasks are scheduled
- coroutines themselves are useless until they are bound to an event loop
- the default event loop in CPython uses **a single thread**
- event loops are *pluggable* – the built-in event loop can be substituted with another
  - e.g. uvloop is a fast implementation in Cython (its performance is comparable to Go and other statically compiled languages)
  - theoretically, a multi-threaded event loop could be developed (but there is still a problem with the Global Interpreter Lock)

Applications

The asynchronous API is useful mainly for hiding latency when waiting for data (IO).

- HTTP requests – client and server packages (`httpx`, `starlette`)
- databases – `sqlalchemy`, ...
- web frameworks – Django, FastAPI, ...

My application demo

- small script to help fighting **linkrot** on the web
- HTTP client using the `httpx` package
- the goal is to check if given URLs are working (HTTP status 200) or not (HTTP status ≥ 400, DNS error, SSL error, connection timeout, ...)

Results

Dataset:

- 46946 URLs for 9084 domains
- links extracted from the Arch Linux wiki

HTTP client parameters:

- connection timeout 60s
- failed connections are retried 3 times

Synchronous vs asynchronous comparison:

- synchronous version: total time **≥ 751m** (the script stopped due to an error...)
- asynchronous version (using locks per every domain): total time **86m 18.613s**

Thank you for your attention!