Coding in Python

...or how to become an expert Python developer in half an hour. 😀

VarArgs, signature and overloads

```
def calculate_distance(x: float, y: float, z: float = 0.0, *args, **kwargs) -> float:
```

Above function can be called in various ways

1. `dist = calculate_distance(3.0, 4.0)` is using positional parameters, the z value is not provided and has a default of zero.
2. `dist = calculate_distance(3.0, y=4.0)` is the same thing but naming the second parameter. Thus a `dist = calculate_distance(y=4.0, x=3.0)` would work also and give the same result.
3. `dist = calculate_distance(3.0, 4.0, 5.0)` is specifying a non-default for z, either by position or by name.
4. But as the function defines a variable number of arguments by position `*args` and by name `**kwargs` specifying more values is fine, like `dist = calculate_distance(3.0, 4.0, 5.0, 6.0, 7.0, space='Riemann')`

Note that positional arguments always have to come first, because otherwise would would not know their, well, position. For example a `calculate_distance(x=3.0, 4.0)` is not possible, because it is not well defined what parameter the 4.0 is meant for. That is also the reason why the `*args` must come before the `kwargs`.

see this for more details

In Python it is common to use named parameters and default values instead of overloads. What would be in Java two methods with different signatures...

```
float calculate_distance(float x, float y) { return calculate_distance(x, y, 0.0) }
float calculate_distance(float x, float y, float z) { return ... }
```

.. can by written in Python as one method using a default value for z.

Having said that, overloads exist, see here.

Python context manager, the with ... statement

A relatively new construct is the `with` statement.

```
with open("hello.txt", "w",) as file:
    file.write("Hello, World!")
```

It is meant for resources that require a close. This is equivalent to...

```
file = open("hello.txt", "w")
try:
    file.write("Hello, World!")
finally:
    file.close()
```

In case a class is written that should support that, two methods must be implemented:

```
    def __enter__(self):
        print('enter method called')
        return self

    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('exit method called')
```

see this for more details

if for variable assignments

Another nice feature of Python is to use `if` with assignments.

```
def inverse(x: float) -> Optional[float]:
  return 1/x if x != 0.0 else None
```

Much shorter than

```
def inverse(x: float) -> Optional[float]:
  if x != 0.0:
    return 1/x
  else:
    return None
```

List and dict comprehension

A quite unusual syntax in Python is to build lists directly via iterators. Take this example

```
number_range: list[int] = [i for i in range(0,10)]
```

`range(0,10)` is a Python provided iterator, which provides the number 0..9 with each next() call. But wer want an array with these numbers. The classic approach would be a...

```
number_range: list[int] = list()
for i in range(0,10):
  number_range.append(i)
```

The list comprehension is just more concise and a bit faster.

List comprehensions also support `if` clauses.

```
even_number_range: list[int] = [i for i in range(0,10) if i % 2 == 0]
```

f-Strings

There are multiple methods for string formatting in Python, we use f-strings.

```
text = f"Current time is {datetime.now(timezone.utc)}"
```

Docs are here.

Parallel processing

As said in the video, Python is an interpreter language. It reads a line, executes it and then reads the next line. One by one. Well, almost. To avoid the constant text parsing, what it actually does is reading the Python file first, parse it and convert that into a binary tree of function calls.

But it does not create Byte code in the sense of CPU specific code, not even code optimization is performed. A Python interpreter is a single thread that executes one command after the other.

See here for more details.

This has obvious implications on parallel execution - it is simply not possible within a single Python interpreter. Concurrent programming yes, parallel processing no. The only ability Python has it to understand when there is an I/O call being made and execute something else during the wait. Unlike with Java and all other languages, a TreadPoolExecutor or a parallel Thread does never utilize an entire second CPU core.

This experiment proves the point

Test #	Test description	Avg. processing time [s]	Perc. time of baseline [%]
1	Execute the function eight times serially without any threads.	10.4	N/A
2	Execute the function eight times concurrently using Thread.	13.3	128%
3	Execute the function eight times using a ThreadPoolExecutor.	13.6	131%
4	Execute the function eight times using a ThreadPoolExecutor inside an asyncio loop.	13.8	133%
5	Execute the function eight times using a ProcessPoolExecutor.	2.2	21%
6	Execute the function eight times using a ProcessPoolExecutor inside an asyncio loop.	2.1	20%

- Test 1 is when the same function is executed one after the other and that shall be the base line of 100% execution time.
- Tests 2, 3, 4 are all using different variants of multi threading, which contrary to any expectations increase the run time by 30% due to overhead.
- Tests 5 and 6 are using multiple Python interpreters, multiple processes, the ProcessPoolExecutor instead of the ThreadPoolExecutor. With all the downsides of multiple independent processes.

asyncio, async, await or Python CoRoutines

For some short lived tasks even multi-threading is too expensive, thus Node.js added that from the start and Java recently as well. The main use case is for web servers. Each request for a file is a tiny task, yet it does a lot of I/O and it can pause its execution for a bit.

Example:

```
def count_file_lines(filename: str):
  count = 0
  with open(filename, "r",) as file:
    count += 1
  return count


if __name__ == "__main__":
  line_count = count_file_lines("file1.txt")
  print(line_count)
```

The code above is quite obvious: The count_file_lines() is called, does something, returns a value and the returned value is used in the print statement.

By adding the keyword `async` to the function, things change slightly.

```
async def count_file_lines(filename: str):
  count = 0
  with open(filename, "r",) as file:
    count += 1
  return count

async main():
  line_count = await count_file_lines("file1.txt")
  print(line_count)

asyncio.run(main())
```

Now the function does not return the value, but because its `async def ...` signature it returns a task, or in Python speak, a Coroutine. This tells Python that this task can be executed in parallel.

To get the actual value of the function, the `await` key word is used. With the obvious consequence that we must wait for the task to be completed. And the entire process must be started with `asyncio.run()` .

But this does not change anything in regards to the execution. Both variants are essentially doing the same thing. The limitation of the Python interpreter to execute only one statement at a time, still applies. That is also the reason why in above tests, Test 4 & 6 have the same execution time as their none-asyncio variants.

So what it is for then? For concurrent tasks that include I/O adn are executed in the same(!) thread.

Take this example: We want to make 1000 Restful API calls to a server.

Option 1: Call the rest endpoint, wait for the result, next call

```
for i in range(0, 1000):
  r = requests.get('https://time.com/current_time')
  print(r.text)
```

Option 2: ThreadPoolExecutor means that each http call is done in a separate thread. Given that a http call is I/O this will have some positive effect but less than expected.

```
def get_time() -> str:
  r = requests.get('https://time.com/current_time')
  return r

with ThreadPoolExecutor(max_workers=10) as executor:
  tasks: list = list()
  for i in range(0, 1000):
    future = executor.submit(get_time)
    list.append(future)

  for future in concurrent.futures.as_completed(tasks):
    try:
      r = future.result()
      print(r)
    except Exception as e:
      print(f'exception {e}')
```

Option 3: asyncio

```
async def get_time(session) -> str:
  async with session.get('https://time.com/current_time') as r:
    return await r


async def get_all():
  async with aiohttp.ClientSession() as session:
    tasks: list = list()
    for i in range(0, 1000):
      task = syncio.ensure_future(get_time(session))
      tasks.append(task)
    all = await asyncio.gather(*tasks)
    for r in all:
      print(r)

asyncio.run(get_all())
```

The important part is to make sure the awaits are properly used. Calling `get_time()` and waiting for it would forfeit the purpose of coroutines. First submit all work and then wait via `gather()` is the desired plan.

Python outlook

There are two features on the horizon in regards to Python, remove the GIL and JIT.

The lack in parallel processing in Python is due to the Global-Interpreter-Lock, the GIL. It is the thing that controls that only one line of code is executed by the Python interpreter at any time. This is going to be removed soon and thus it will enable use to use ThreadPoolExecutors efficiently. For most of our code that will not be important as we use Python more as an orchestration tool and the heavy lifting is done in the libraries but the Lakehouse Writer Container can greatly benefit from it.

Azure Function Apps in general are single CPU only (with our current pay-per-use model), hence asyncio is good enough.

Java has a Just-In-Time compiler and Python will get one as well. The idea is, once a code sequence has been executed a couple of times, e.g. a function call being executed 1000s of times in the current execution, it might make sense to rather turn that part of the code in the CPU native assembler code. Should speed up Python, because we are all about row processing. Millions of rows processed by the same code.

So as you can see, all the shortcomings of Python that are important for us are bound to be removed in the coming years.