PCP-DeepDive-Python

Within the module Programming Concepts and Paradigms (PCP) at HSLU Thomas and Maurizio decided to have an extended look at Python. The programming language shall be analyzed and compared to other languages discussed in the module. Furthermore, some previously solved exercise shall also be completed in Python to allow for further comparison.

Basics

Python is an interpreted object-oriented/functional programming language with dynamic typing

Python Enhancement Proposals (PEPs)

In an analog manner to Java Enhancement Proposals (JEP), Python developers maintain a list of possible future enhancements to the Python Language named PEP. Some features which we are going to discuss were such a proposal once. They are going to be marked accordingly.

A Special guest here is PEP 8 – Style Guide for Python Code which deserves some special attention. PEP8 is a styleguide on how to write Python code. This leads (if you read warnings and follow the recommendations) to a more uniform look of the written programs in general.

DeepDive

Within this Project we are going to shed some light especially on the following topics

- Concurrency / Parallelism
- Duck-Typing
- Indentation
- List Comprehension
- Yield
- Zen of Python

Setup

This project is built and tested with Python version 3.9. As for now only the library "requests" is needed in addition to the plain Python installation. Your IDE will probably notify you about this by pointing out this information based on the requirements.txt If this is not the case, please use pip or your favorite package manager to install this dependency.

Concurrency / Parallelism

Python and Ruby (and probably some otters?) use a Mechanism called **Global Interpreter Lock (GIL)**. This mechanism is used to synchronize the execution of threads so that there is only one thread (per process) active at a given time. Even if there are multiple CPU cores available only one thread will be active. **All threads share the same memory space!** Threads are still useful if a program has to deal with a lot of IO which is considered slow. While a thread is waiting for IO to complete, another thread can do it's work.

On the other hand there is true parallelism by spawning multiple processes (instead of threads). These are not affected by the GIL and will be running at the same time if the architecture allows it. The downside is that Python needs to copy the whole memory space for each process and a separate interpreter running the code. Depending on available resources (RAM) this could lead to issues.

```
Measuring performance of CPU Bound tasks - Counting down from 200000000 to Zero
single_threaded took 4.39 seconds
multi_threaded took 4.49 seconds
multi_process took 0.79 seconds

Measuring performance of IO Bound tasks - Get IP by calling https://httpbin.org/ip 8 times
single_threaded took 4.22 seconds
multi_threaded took 0.91 seconds
multi_process took 0.83 seconds

Process finished with exit code 0
```

Duck-Typing

The name "Duck Typing" comes from the phrase: "If it looks like a duck and quacks like a duck, it's a duck".

Duck Typing is a concept related to dynamic typing. The type of object is less important than the method and attributes that define it. Duck Typing only checks whether a particular method or attribute is present. If it is not present it throws a AttributeError.

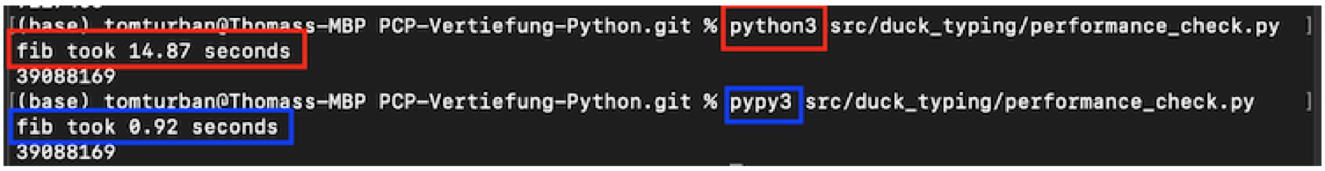
```python
class Casino:
    def lose_money(self):
        print("next time I win!")

class StockMarked:
    def lose_money(self):
        print("in ten years it will be up again!")

class TryBecomeRich:
    def __init__(self, methode):
        methode.lose_money()

TryBecomeRich(Casino())
# next time I win!

TryBecomeRich(StockMarked())
# in ten years it will be up again!
```

Dynamic Typing

Python is dynamically typed, so type checking is done only at runtime. This means that a type of variable can change. The type of variable is determined by the type of the value assigned to it. Dynamic typing has a big disadvantage. A program has a poor performance because of it.

There are several ways to improve the performance of a program. One of them is to use Pypy. Pypy is a just-in-time compiler written in Python. A just-in-time compiler translates programs into machine code at runtime. This can be used to increase performance.

In an example the performance of Pypy was compared with the Python interpreter. The task was to recursively calculate the 38 Fibonacci number.



```
[(base) tomturban@Thomass-MBP PCP-Vertiefung-Python.git % python3 src/duck_typing/performance_check.py   ]
fib took 14.87 seconds
39088169
[(base) tomturban@Thomass-MBP PCP-Vertiefung-Python.git % pypy3 src/duck_typing/performance_check.py     ]
fib took 0.92 seconds
39088169
```

You can see that execution with Pypy is about 14 times faster than with the Python interpreter.

Indentation

In Python indentation is used as a structuring element to tell the Python interpreter that this code belongs together. Many other languages use braces or keywords to mark blocks of code.

It is important that the indentation is the same throughout the code block and at least 1 space. A wrong indentation leads to an "IndentationError" and the code is not compiled.

```python
def speed_check(speed: int) -> str:
    # indention can be different in each code block
    if speed > 50:
            return "to fast"
    else:
        return "everything ok"
```

List Comprehension

PEP 202 – List Comprehensions was created on July, 13 2000 for Python 2.0. The idea was/is to allow conditional construction of list literals with if statements and loops.

```python
# Example before PEP202
my_list = []
for i in range(10):
    if i % 2 == 0:
        my_list.append(i)

# Example after PEP202
my_list = [i for i in range(10) if i % 2 == 0]
```

Yield

The yield statement is very similar to the return statement. Both return a value of the function. The difference is that when return is called, the function is terminated. The yield statement, on the other hand, only interrupts the function and stores the necessary data so that the function can continue later at the same point.

```
def yield_example():
        yield "a"
        print("hello")
        yield "b"

example = yield_example()
print(example.next())          # a
print(example.next())          # hello  # b
```

Yield also exists in other languages such as Kotlin. It behaves very similar to the yield in Python. In Kotlin the yield statement is a standard library function and not a key word as it is in Python.

Generator

A function is a generator function as soon as it contains a yield statement. It can also contain multiple yield statements as well as return statements.

A generator function returns a generator object. This can be used as an iterator.

```
def fib_generator(end):
    a = 0
    b = 1
    while a < end:
        yield a
        a, b = b, a + b
```

Yield in Coroutine

Coroutines are very similar to generators. However, they have additional methods and the yield statement is used differently. Coroutines can produce data like generators. In addition, they can also consume data. This is achieved with a different use of the yield statement.

```
def squarer(next_coroutine):
        while True:
            # receive value from other coroutine or function
            number = (yield)
            square_number = number ** 2
            # send value to other coroutine
            next_coroutine.send((number, square_number))
```

Data can be sent to the coroutine with send() method. Coroutines only run if the next() or send() method has been called. It can be stopped with the close() method, because otherwise the coroutine run indefinitely.

Zen of Python

PEP 20 – The Zen of Python was created in late August 2004. Tim Peters, a long time Pythoneer, wrote down 19 guiding principles on how to write Python programs. Those principles remind us

(Thomas/Maurizio) of Clean Code by Robert C. Martin.

```
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Team Conclusion

- Working in a Team of two is quite ok
- Fast results
- Small projects YES! Big projects... maybe less so :)
- Slow compared to other (compiled) languages -> use pypy or other speed-up possibilities
- Sleek and clean code due to less cluttering

Thomas Conclusion

- interesting language
- Dynamic typing takes some time to get used to
- Easy to start
- there are interesting ways to improve performance

Maurizio Conclusion

- Fast and fun to code
- easy to learn / hard to master
- good for prototyping and small programs
- advantages in Machine Learning due to the possibility to use Google Colab