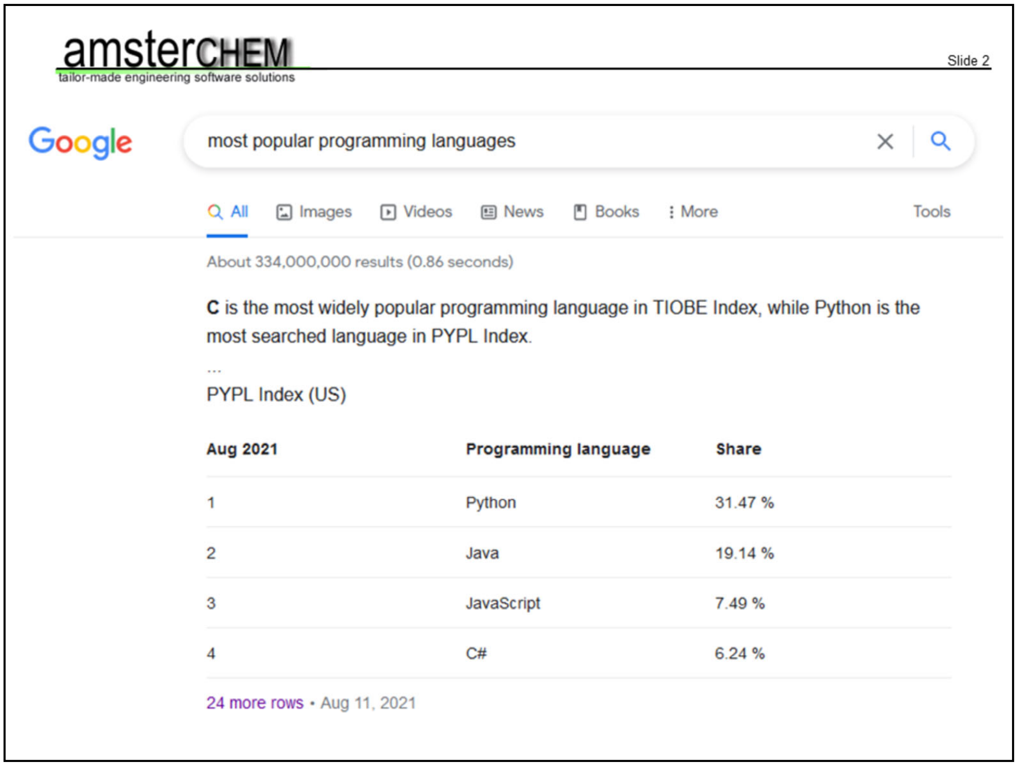
amsterCHEM
tailor-made engineering software solutions

Jasper van Baten – AmsterCHEM

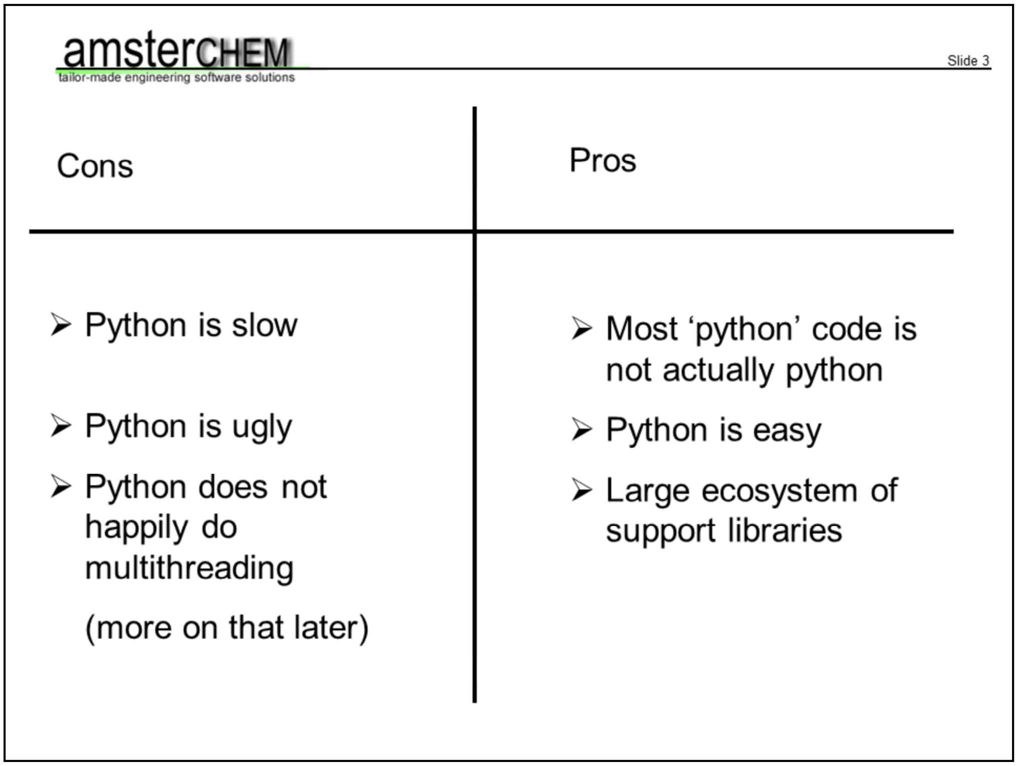
PYTHON UNIT OPERATION

CAPE-OPEN Annual Meeting 2021, October 27-28

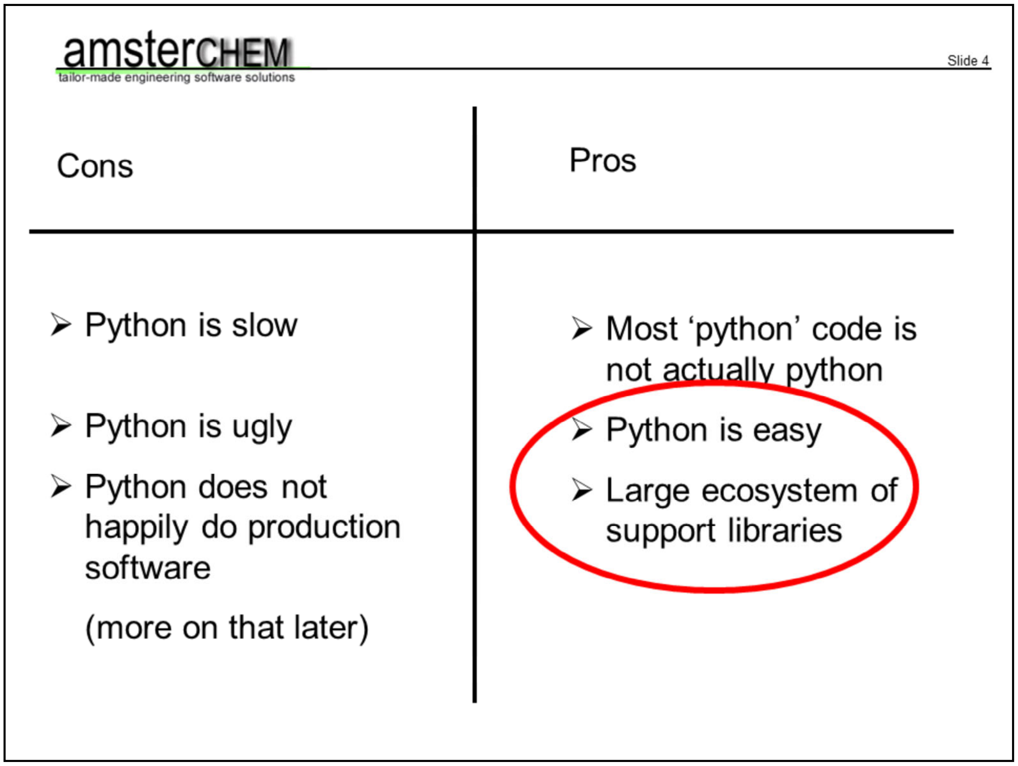
Good afternoon. My name is Jasper. This presentation will feature my new COBIA based CAPE-OPEN unit operation: the Python Unit Operation.



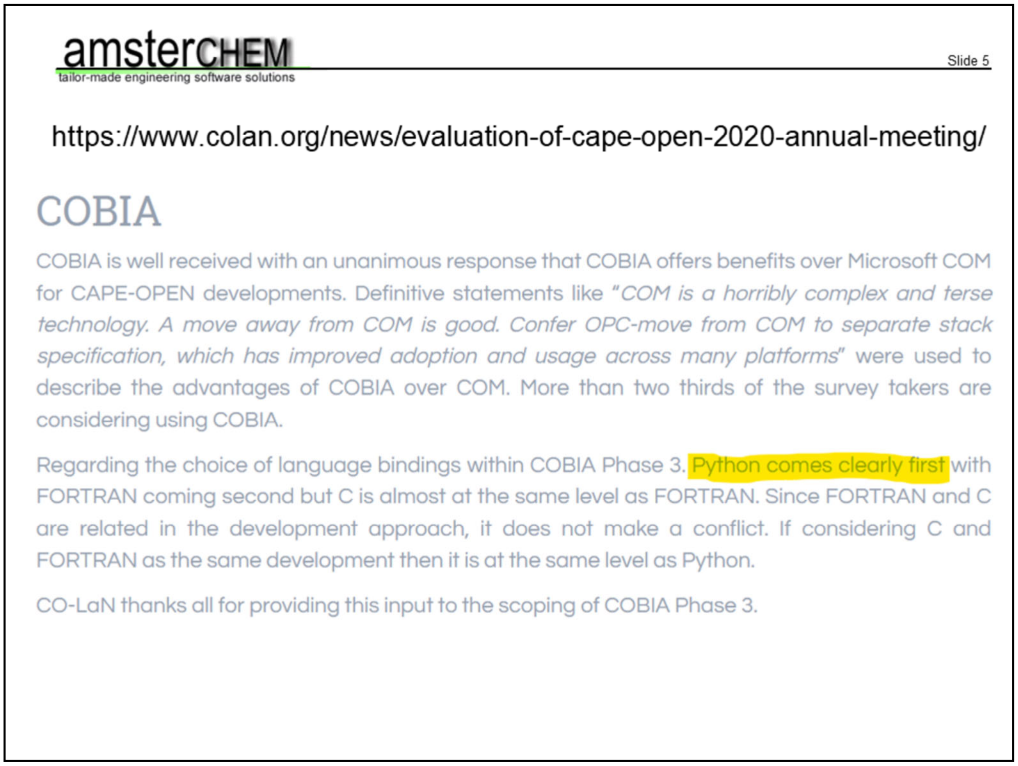
Let’s start with the motivation. Python has made a rapid climb in the last years becoming the most popular programming language. When googling “most popular programming languages” this is what comes up.



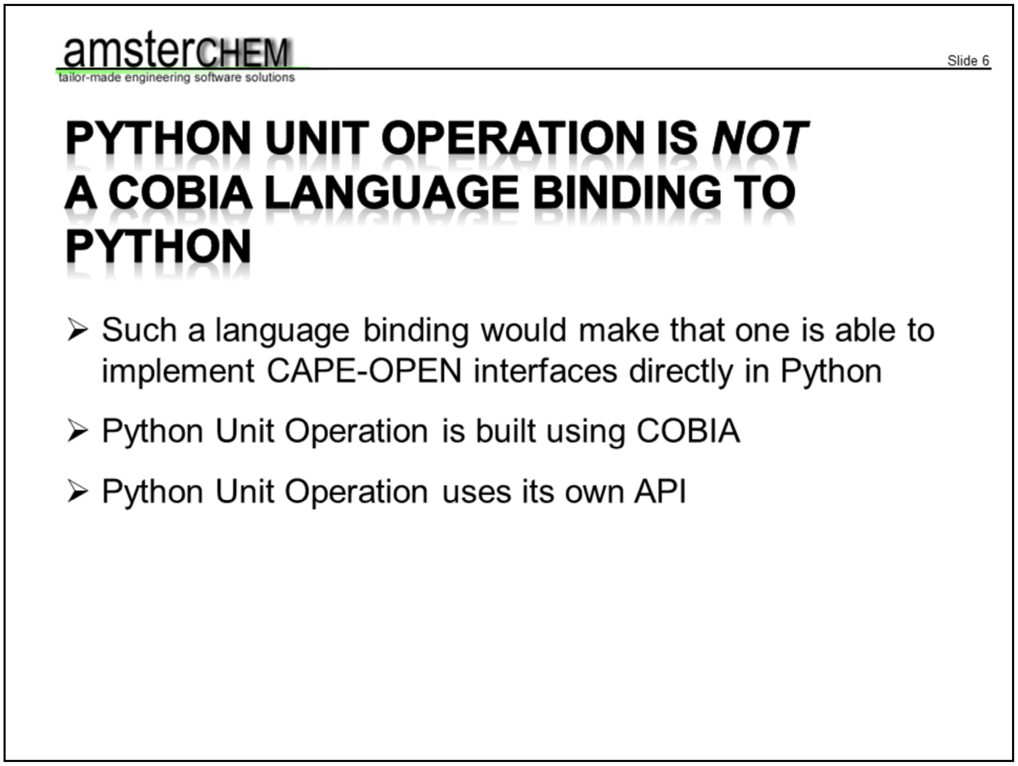
To understand why this is so I have made a list of pros and cons for Python. First con, Python is slow. This is of course a disadvantage, but on the pro side we see that most code executing in Python is not actually written in Python. As an example, if you use a numeric solver, it is likely from the scipy package, which uses highly optimized native routines. Next, Python is ugly. There was probably a good reason to call the language after Monty Python, it is so ugly that it is almost funny. With spacing and indenting being part of the syntax specification I get all nostalgic and I feel like I am back in 1977. Of course on the up side, many people find Python really easy to learn and to understand and to read. For me a rather big down side is that Python does not really lend itself well for multi-threaded production software, more on this later. But on the up side, there is a large ecosystem of support libraries ready for you to use.



I am going to put some emphasis on these two pros: the combination of ease of use and plenty of support libraries, including numerical solvers, makes Python an excellent platform for prototyping and quick development. This is I think why Python is so popular.



Another big motivator of this project is the feedback from our own community. After the CAPE-OPEN 2020 Annual Meeting, there was a questionnaire, and one of the results from this is that our community really wants to see Python as being accessible though CAPE-OPEN. The Python Unit Operation, and its sister product Python CAPE-OPEN Thermo import, may fill this void to some extent, but….



(and this is a really long title for a slide), Python Unit Operation is not a COBIA language binding for Python. Such a language binding would have you implement CAPE-OPEN interfaces directly in Python. This is not what the Python Unit Operation does. It is indeed built on COBIA. As it is recent software development, that decision makes sense.. But Python Unit Operation offers its own programming interface towards coding a unit operation, bypassing most of the CAPE-OPEN details, which are taken care of under the hood. So now that we know what the Python Unit Operation is not, let us look at what it is. And how we can learn from it for making a COBIA Python language binding.

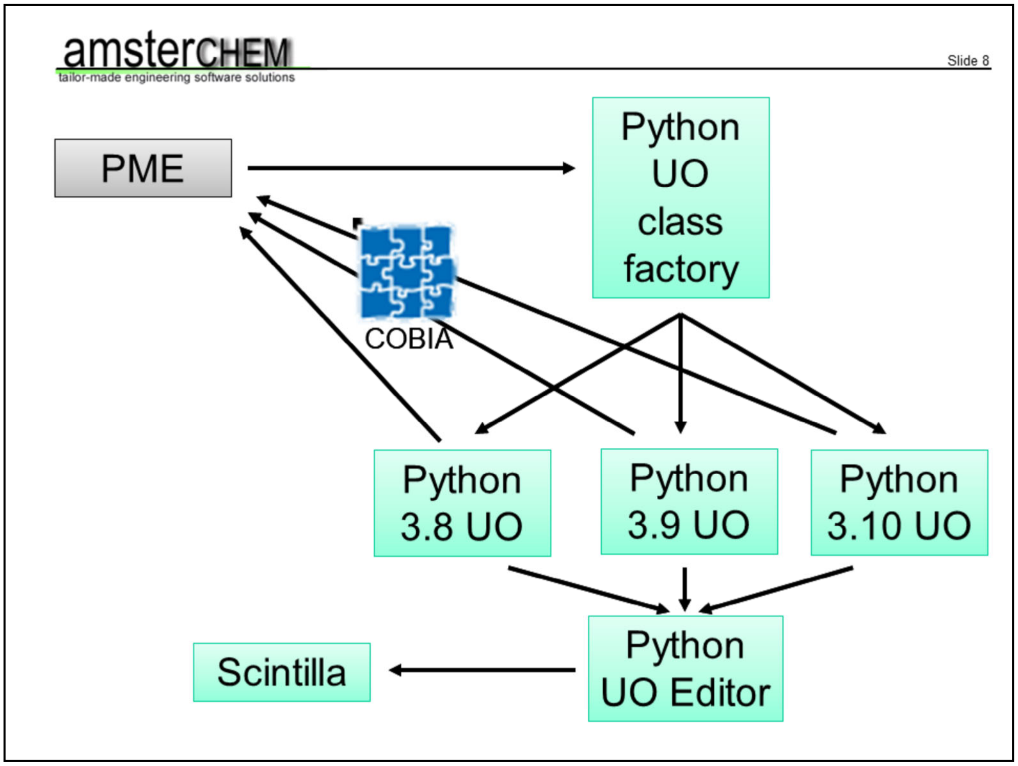
amsterCHEM
tailor-made engineering software solutions

Slide 7

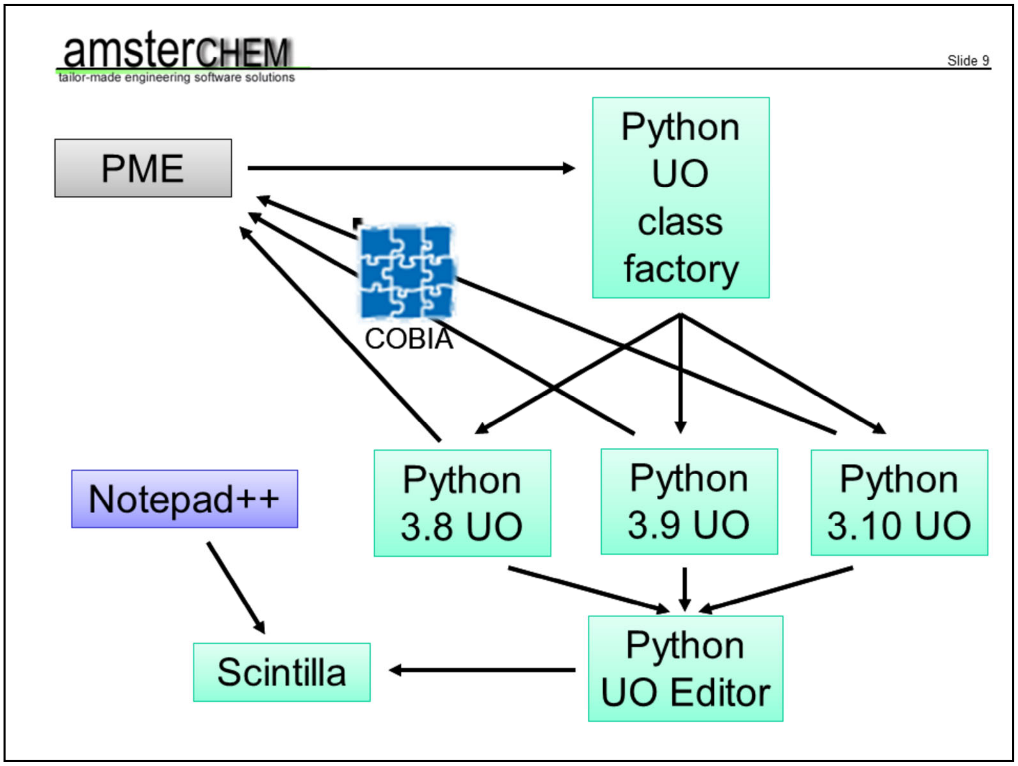
PRESENTATION OUTLINE

- Motivation
- What is the Python Unit Operation?
- Learnings for COBIA Python Language binding

So after all that, I finally get to the outline of this presentation. I want to introduce you to the Python Unit Operation. I also want to share what I have learned building the Python Unit Operation, and how it would reflect on a possible COBIA language binding to Python.

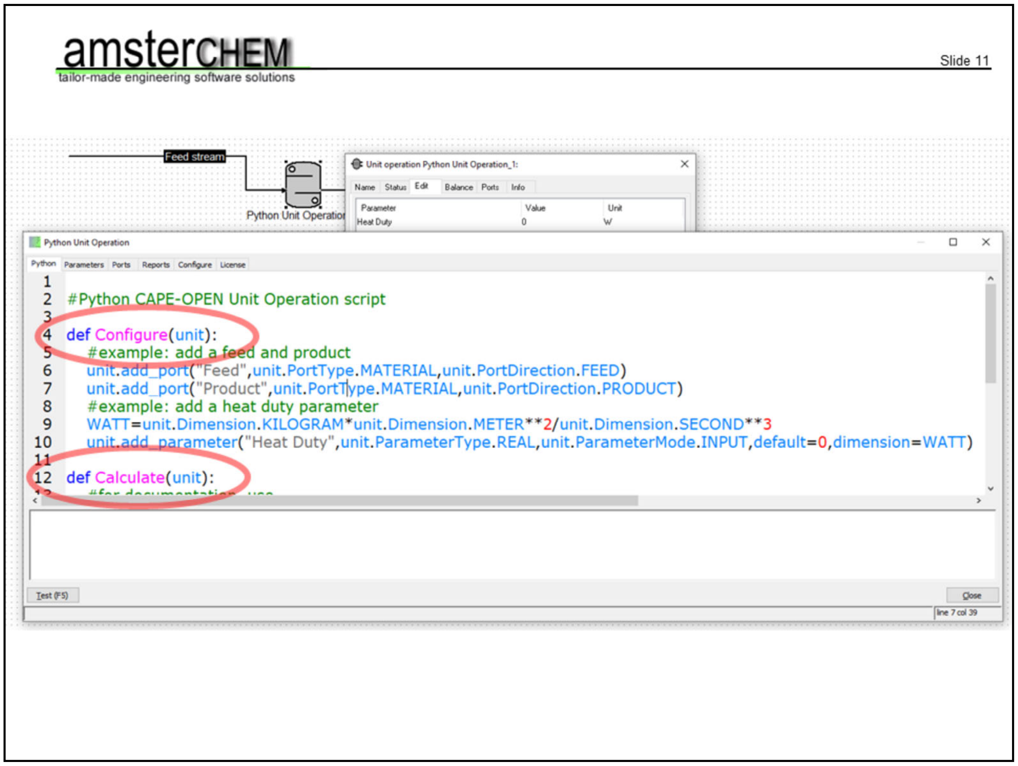


The Python Unit Operation is not actually one unit operation. If you instantiate the unit, the class factory will determine if it can find a supported version of Python on your system, for the current bitness. If you run a x64 PME, you will need an x64 Python installation. Then it will instantiate the proper Python Unit Operation using the appropriate module for the appropriate Python version. All of these make use of the same editor component, which in turn makes use of the Scintilla editing component, giving a syntax highlighting editor experience. All CAPE-OPEN interactions use COBIA as the middleware.

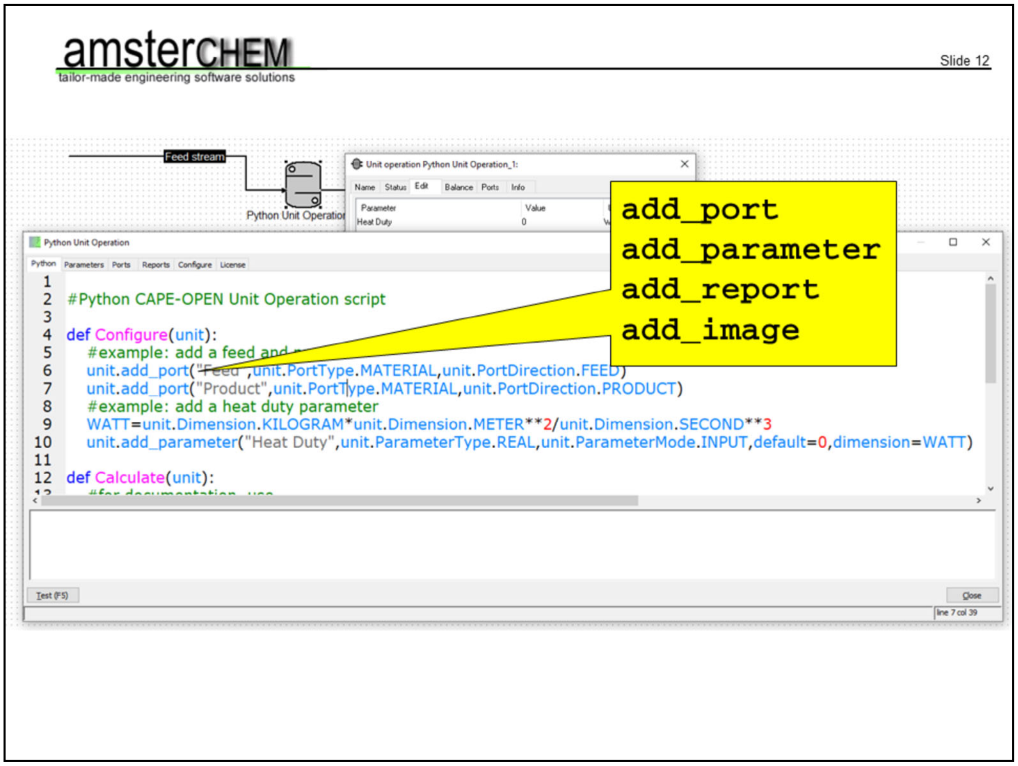


For those of you not familiar with Scintilla, it is the editor component underlying the popular Notepad++ editor.

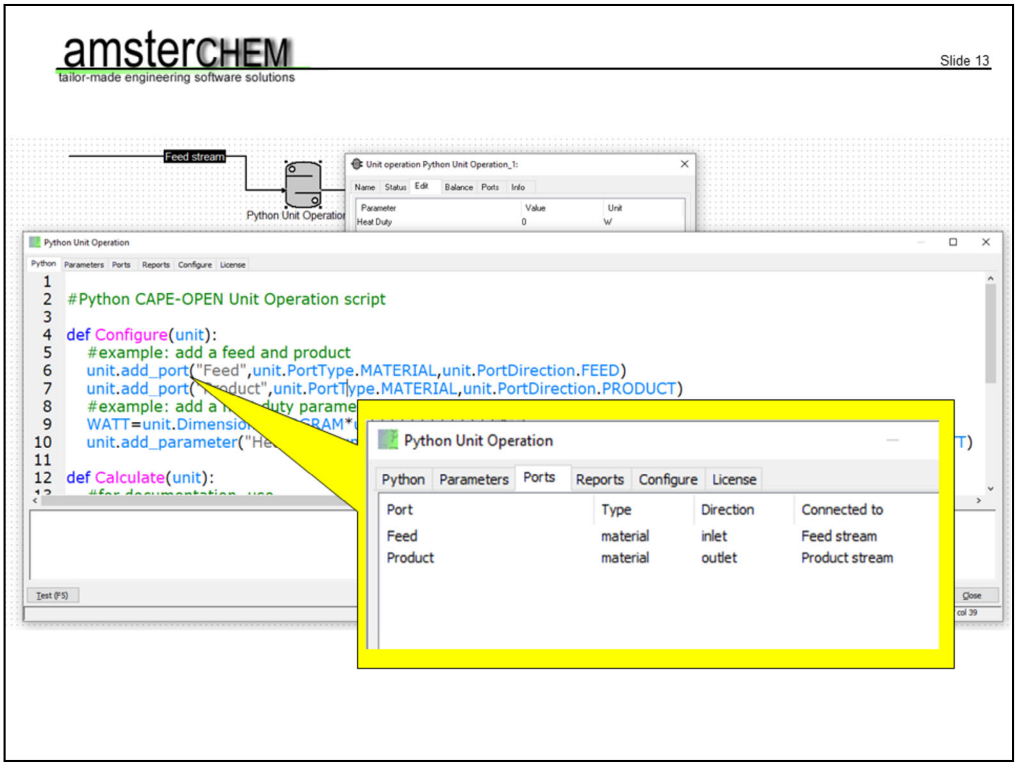
So I will give a short introduction in what the Python Unit Operation is and does. For this I will use the default Python script that will be there when you drop a Python Unit Operation in the flowsheet. A simple constant duty heater where heat duty is an input parameter. One feed, one product, no pressure drop. If you edit the unit operation, this editor window will show.



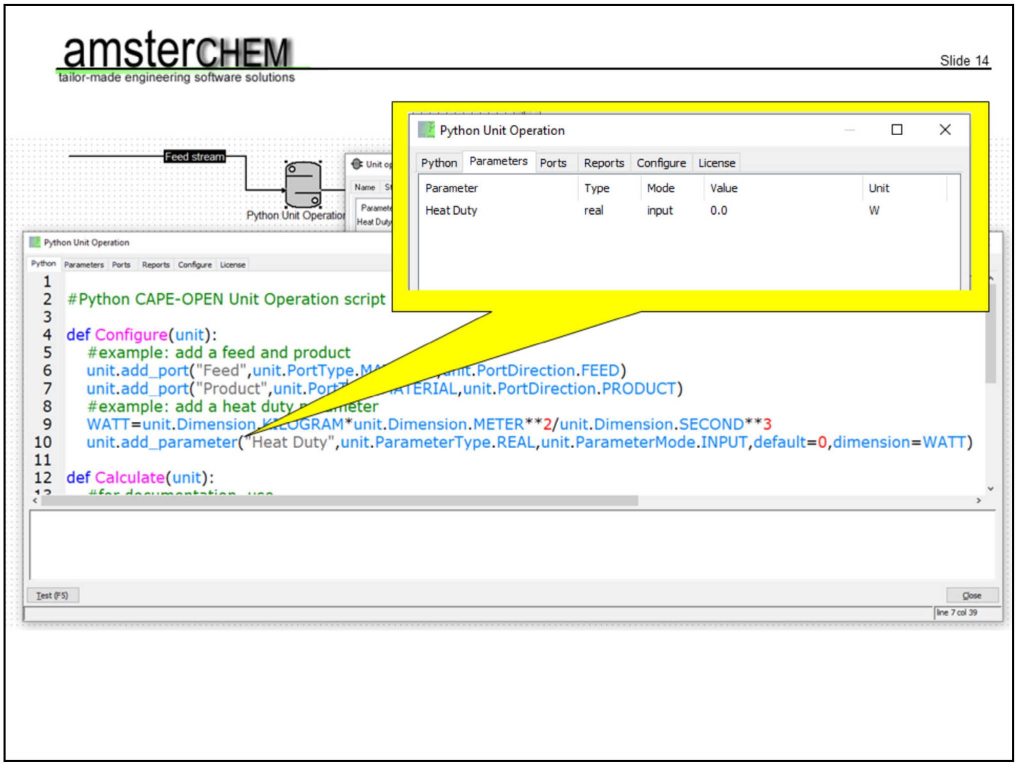
There are essentially two functions you need to provide. The Configure function allows you to …. well … set up the unit operation configuration, and the Calculate function, you guessed it, is called when the unit operation is calculated. There are other functions that you can provide, but these two are the only required ones. Let's start with the Configure function. The argument to the configure function is the object that provides the access to the CAPE-OPEN unit operation.



In the configure function, you can add ports, you can add parameters, and you can add two kinds of reports. The old fashioned textual reports that were introduced in CAPE-OPEN 1.0 Unit Operations, and image reports that are new to the reporting interface in CAPE-OPEN 1.2. Of course when used from a CAPE-OPEN 1.0 or 1.1 compliant PME, these image reports will be unavailable to the PME, but you can still inspect them from the Reports tab.



As said, you are not implementing a CAPE-OPEN unit operation yourself. You do not need to implement a port collection, this is all done for you. Adding a port will automatically add it to the port collection. You can inspect the results from the ports tab. The Python Unit Operation supports material-, energy- and information ports.



Similarly you do not need to implement parameter objects, parameter specification objects, a parameter collection; also this is all taken care of. Just add a parameter, with dimensionality, type and mode (input or output), and again the result can be immediately checked from the parameter tab. The Python Unit Operation supports real, integer, Boolean, string and real array parameters. Setting up text reports or image reports is equally simple, and not shown here. There is a default report, which will capture any output you print to the standard output.

```
def Calculate(unit):
    #for documentation, use
    #  help(unit)
    #example: copy Feed to Product, apply heat duty
    fd=unit.ports["Feed"]
    duty=unit.parameters["Heat Duty"].value
    if fd.flow_rate==0 and duty!=0:
        raise RuntimeError('Zero feed flow rate;
                 cannot apply duty')
    h=fd.get_property('enthalpy') #molar enthalpy
    if (duty!=0):
        h+=duty/fd.flow_rate
    unit.ports["Product"].set(x=fd.x,p=fd.p,
              flow_rate=fd.flow_rate,enthalpy=h)
```

So let's have a look at the calculation script. Again for the sake of simplicity I will not go past the default example script that is there when you drop the unit operation into the flowsheet.

amsterCHEM
tailor-made engineering software solutions

Slide 16

```
def Calculate(unit):
    #for documentation, use
    #  help(unit)
    #example: copy Feed to Product, apply heat duty
    fd=unit.ports["Feed"]
    duty=unit.parameters["Heat Duty"].value
    if fd.flow_rate==0 and duty!=0:
        raise RuntimeError('Zero feed flow rate;
                 cannot apply duty')
    h=fd.get_property('enthalpy') #molar enthalpy
    if (duty!=0):
        h+=duty/fd.flow_rate
    unit.ports["Product"].set(x=fd.x,p=fd.p,
              flow_rate=fd.flow_rate,enthalpy=h)
```

Any port added using add_port in configure, is available during Calculate from the unit.ports dictionary. Shown here are all references to the feed port. One can just get properties of the feed port. If this is not sufficient, the full thermodynamic API, compatible with Python CAPE-OPEN Thermo Import, is available via each object connected to a material port. The CAPE-OPEN Unit Operation standard says you cannot do anything that has side effects on material objects connected to feed ports. Hence, if you need to calculate a property, which would have the side effect of the property being available, you should do so on a duplicate of the material object. None of these CAPE-OPEN concerns matter much here, this duplication, when needed, is done automatically behind the scenes. In this case it is needed, as we calculate and obtain enthalpy.

amsterCHEM
tailor-made engineering software solutions

Slide 17

```python
def Calculate(unit):
    #for documentation, use
    #  help(unit)
    #example: copy Feed to Product, apply heat duty
    fd=unit.ports["Feed"]
    duty=unit.parameters["Heat Duty"].value
    if fd.flow_rate==0 and duty!=0:
        raise RuntimeError('Zero feed flow rate;
                     cannot apply duty')
    h=fd.get_property('enthalpy') #molar enthalpy
    if (duty!=0):
        h+=duty/fd.flow_rate
    unit.ports["Product"].set(x=fd.x,p=fd.p,
              flow_rate=fd.flow_rate,enthalpy=h)
```

Material streams expose thermodynamics consistent with:

Python CAPE-OPEN Thermo Import (AmsterCHEM)

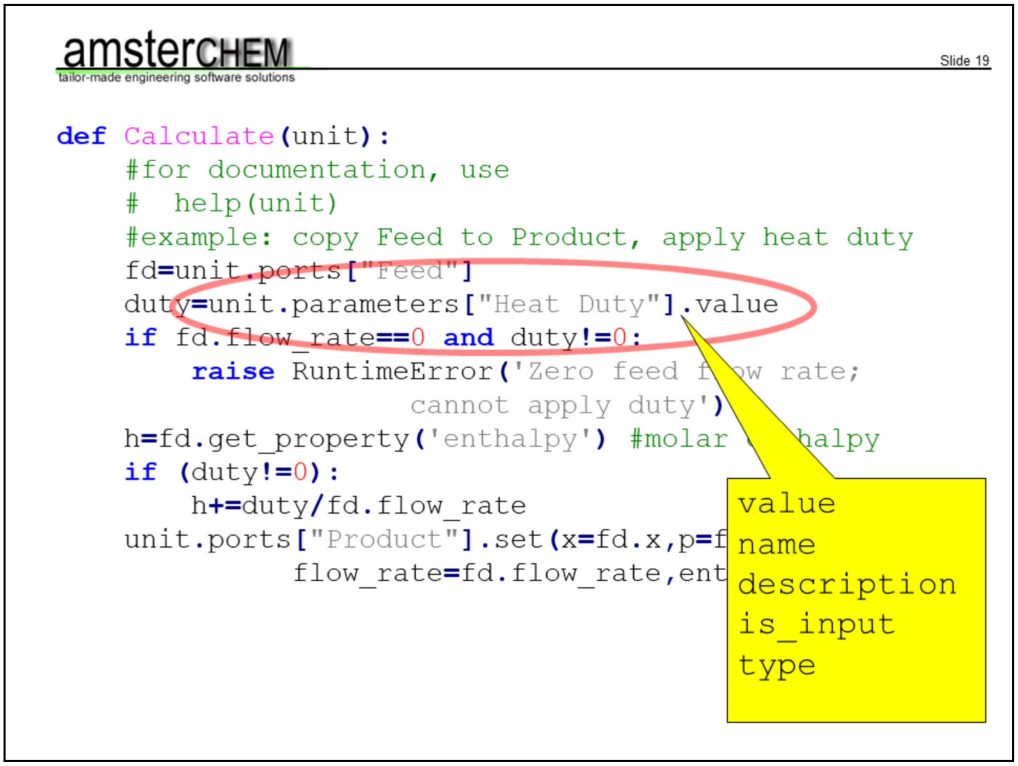
Finally the CAPE-OPEN Unit Operation standard says we must flash our product ports. This is taken care of by the set function, that is circled here for the product port. This unit does a PH flash.

amsterCHEM
tailor-made engineering software solutions

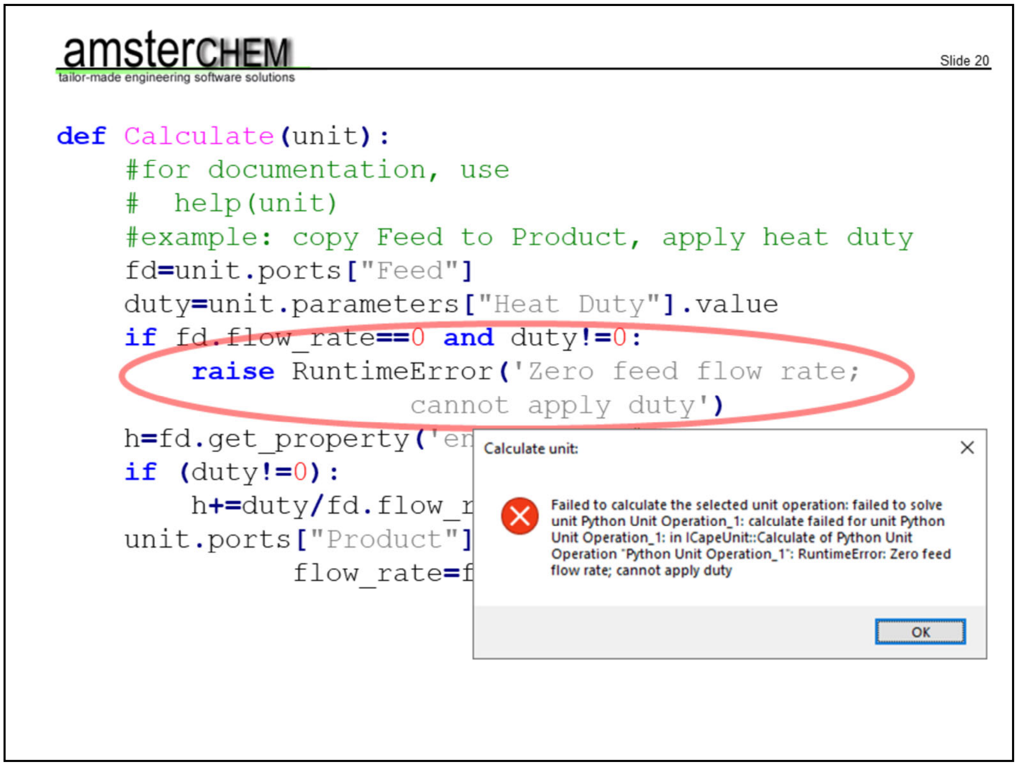
Slide 18

```
def Calculate(unit):
    #for documentation, use
    #  help(unit)
    #example: copy Feed to Product, apply heat duty
    fd=unit.ports["Feed"]
    duty=unit.parameters["Heat Duty"].value
    if fd.flow_rate==0 and duty!=0:
        raise RuntimeError('Zero feed flow rate;
                cannot apply duty')
    h=fd.get_property('enthalpy') #molar enthalpy
    if (duty!=0):
        h+=duty/fd.flow_rate
    unit.ports["Product"].set(x=fd.x,p=fd.p,
            flow_rate=fd.flow_rate,enthalpy=h)
```

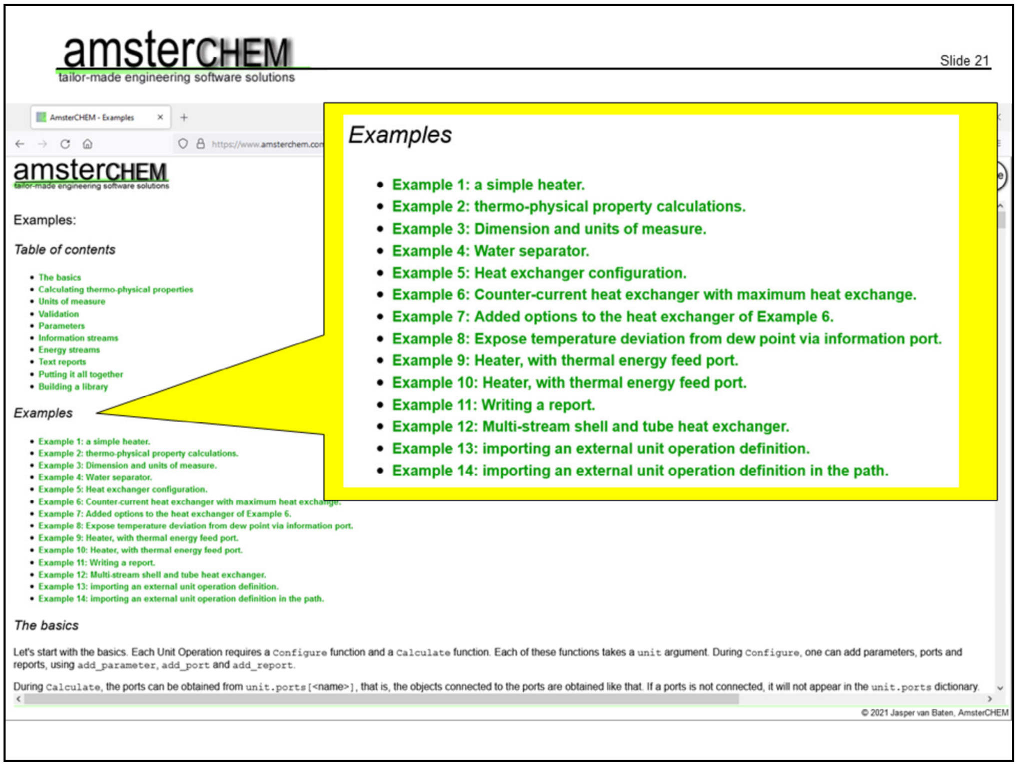
Parameters are available from the unit.parameters dictionary, and the value can be set and obtained.



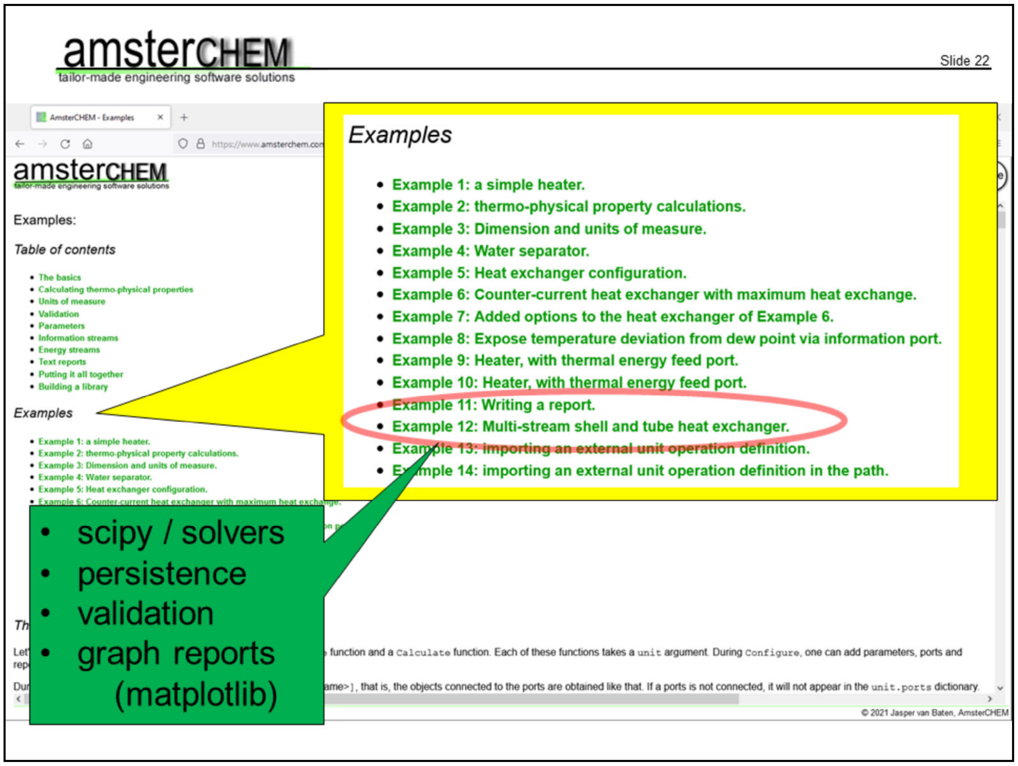
Other members include name, description, is_input, and type, all of which are read only during Calculate.



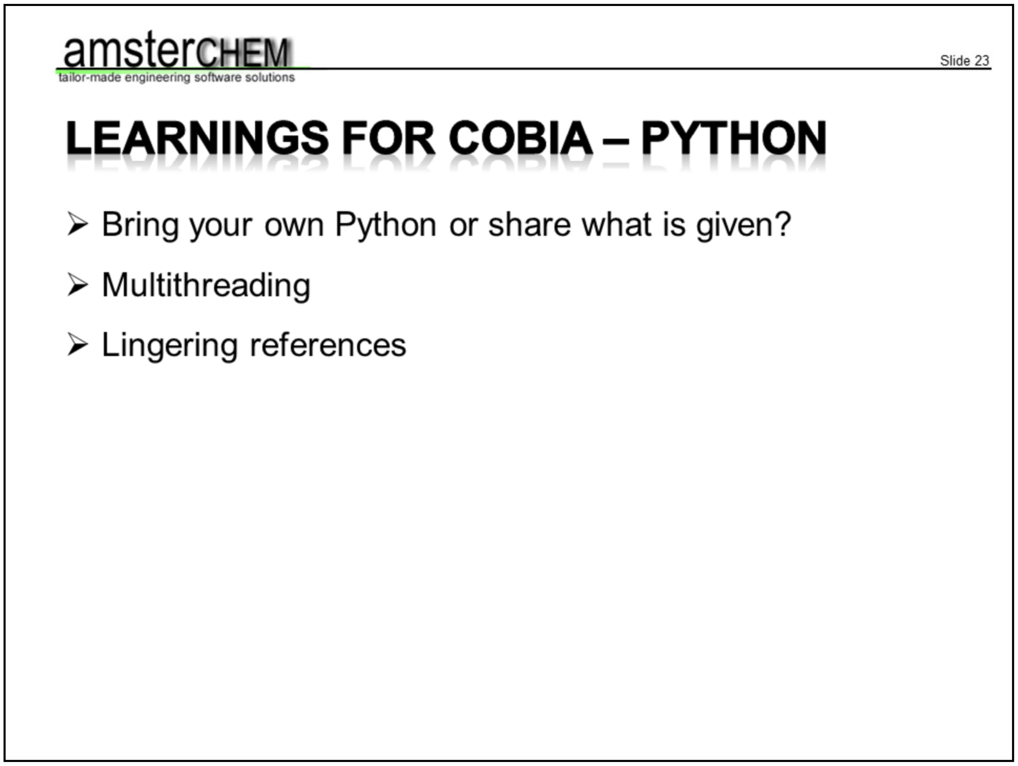
Error handling is as simple as raising a Python exception of your choice. This is automatically converted to a CAPE-OPEN error, and handled by the PME as it sees fit. Shown here is what COFE does if you select Calculate for a single unit operation.



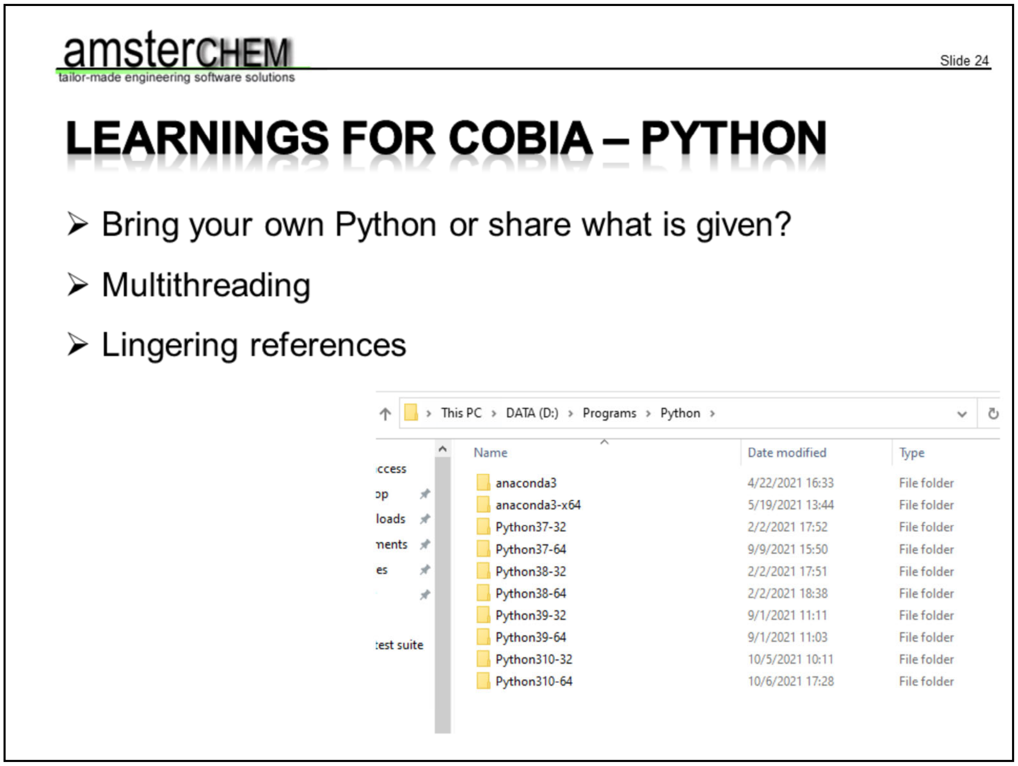
OK – that may have been an extremely simple example. But fear not, much more illustrative examples are available from the AmsterCHEM web site.



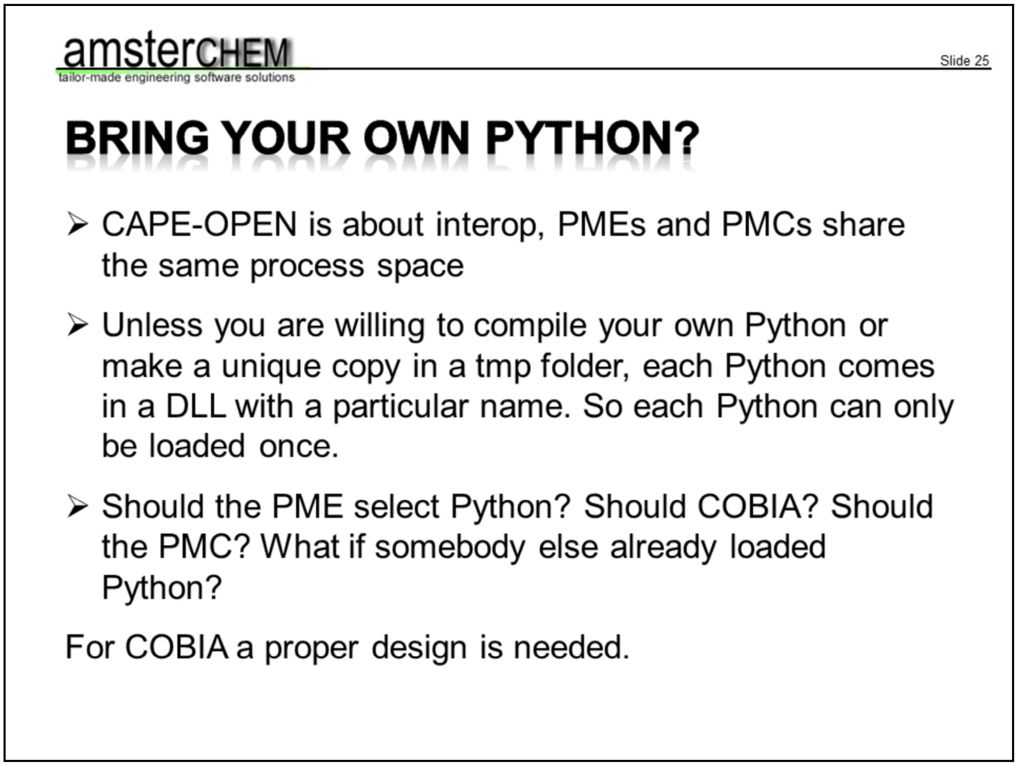
If you immediately want to dive into the more complex functionality, the multi-stream heat exchanger uses solvers from scipy, it uses persistence, it uses validation and it creates plots temperature profiles along the device. Our friends from ChemSep immediately applied this multi stream heat exchanger example in several of their example flowsheets, available from chemsep.org or cocosimulator.com.



So much for showing what the Python Unit Operation is. I promised I would share my findings on using Python in a COBIA language binding, and I said I would elaborate on why Python does not happily do production software. There are three points in particular I would like to discuss. The first is the choice of which Python to use.



My computer contains a whole nest of Pythons and they seem to multiply. Secondly I will gloss over Python’s multithreading provisions, or rather the lack thereof, particularly in the context of sharing Python with your PME or other PMCs, and finally I will touch on an issue that is not unique to Python, the issue of lingering references. From here on out I am afraid the presentation will get a bit more technical. So hold on to your hats.



CAPE-OPEN is about interop, PMEs and PMCs share the same process space. Who is responsible for picking which Python is to be used in such a context, and who loads Python, and if Python is already loaded, how do you with it not being the version of Python you were hoping for. These are far from a trivial problems. The Python Unit Operation has several steps in place to take care of this, including a method in which you identify yourself which Python is to be used. It skips loading python3.dll, as this DLL name is shared between all Python 3 versions, and each Python Unit Operation implementation is specific to a particular Python version, and statically binds to a particular DLL name, e.g. python39.dll for Python 3.9. It could of course be that this Python is already loaded by somebody else, in which case we are sharing Python with others. So best not to make use of any global variables, short of the list of loaded modules (you can of course immediately see there are potential problems there too). On top of this, how we deal with threading also is affected by who we are sharing Python with and how they are dealing with threading.

Note that all Python Unit Operations share the same underlying Python. So they should probably stay away from the use of global variables. The Python Unit Operation resolves this by loading the user script under the hood inside a private, unique module. That all Python Unit Operations share the same instance of Python also has consequences for multi-threading.

amsterCHEM
tailor-made engineering software solutions

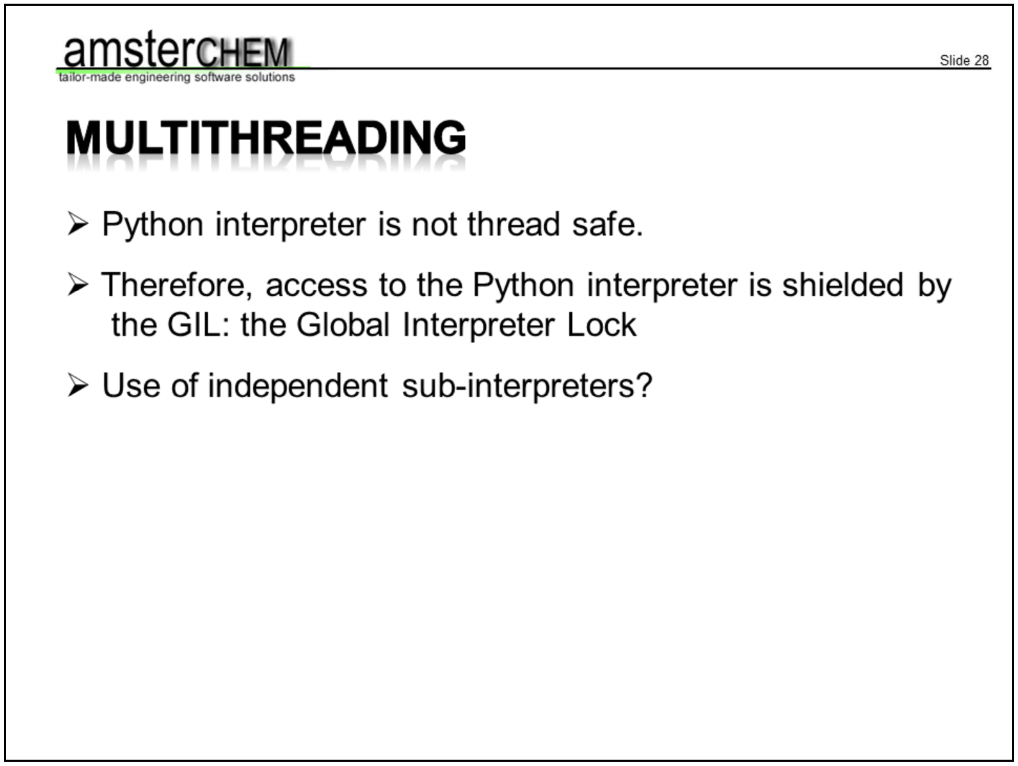
Slide 26

MULTITHREADING

- Python interpreter is not thread safe.
- Therefore, access to the Python interpreter is shielded by the GIL: the Global Interpreter Lock

The Python interpreter is, alas, not thread safe. Access to the Python Interpreter is shielded by the Global Interpreter Lock, also infamously known as the GIL. You need to acquire the GIL any time you want to execute something in the interpreter. During any interpreted code, Python itself may temporary unacquire the GIL, during for example lengthy file operations or other external actions, and then the GIL is reacquired by Python itself. Then you must release the GIL when you are done with the interpreter. Consequently of course, multithreading Python is rather inefficient in this manner. Surprisingly, this is how Python's built-in module python.threading is working.

Here's the excerpt from the documentation in Python 3.10. Multiprocessing would be an option, if you don't mind intra-process marshaling, which is surely a performance killer altogether.



The next route down to investigate would be the use of sub-interpreters, which are independent of each other and can run concurrently. There is an API on this, but there are particular items in the documentation and corresponding PEP (Python Enhancement Proposal) that are rather frightening.

amsterCHEM
tailor-made engineering software solutions

Slide 29

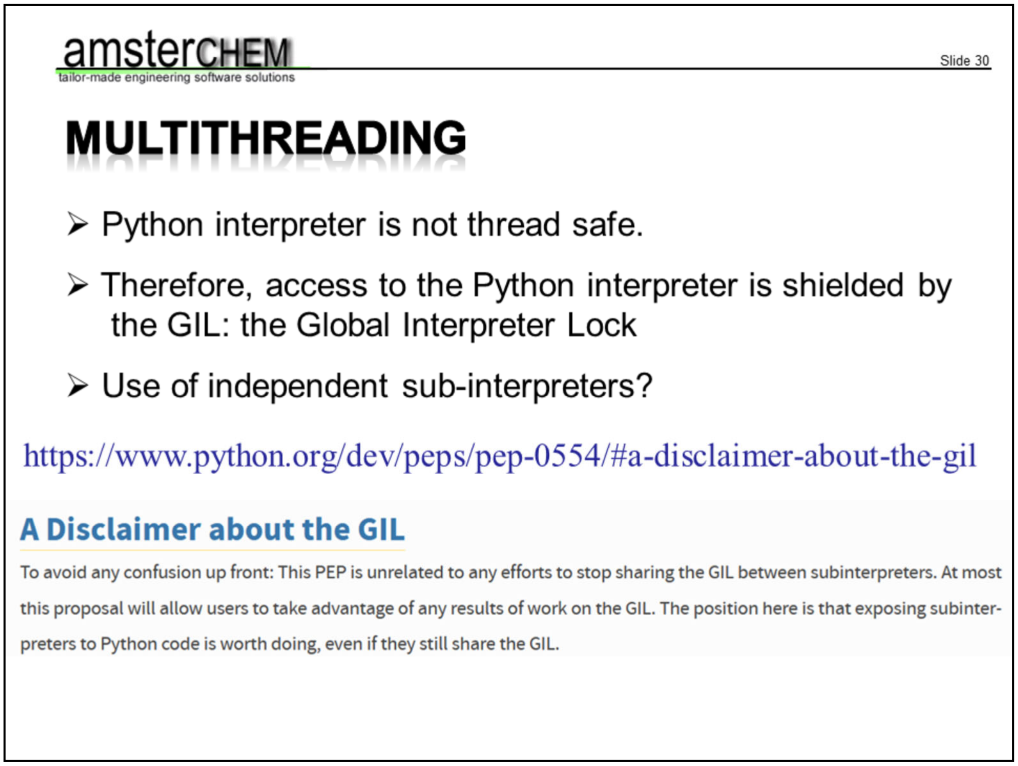
MULTITHREADING

- Python interpreter is not thread safe.
- Therefore, access to the Python interpreter is shielded by the GIL: the Global Interpreter Lock
- Use of independent sub-interpreters?

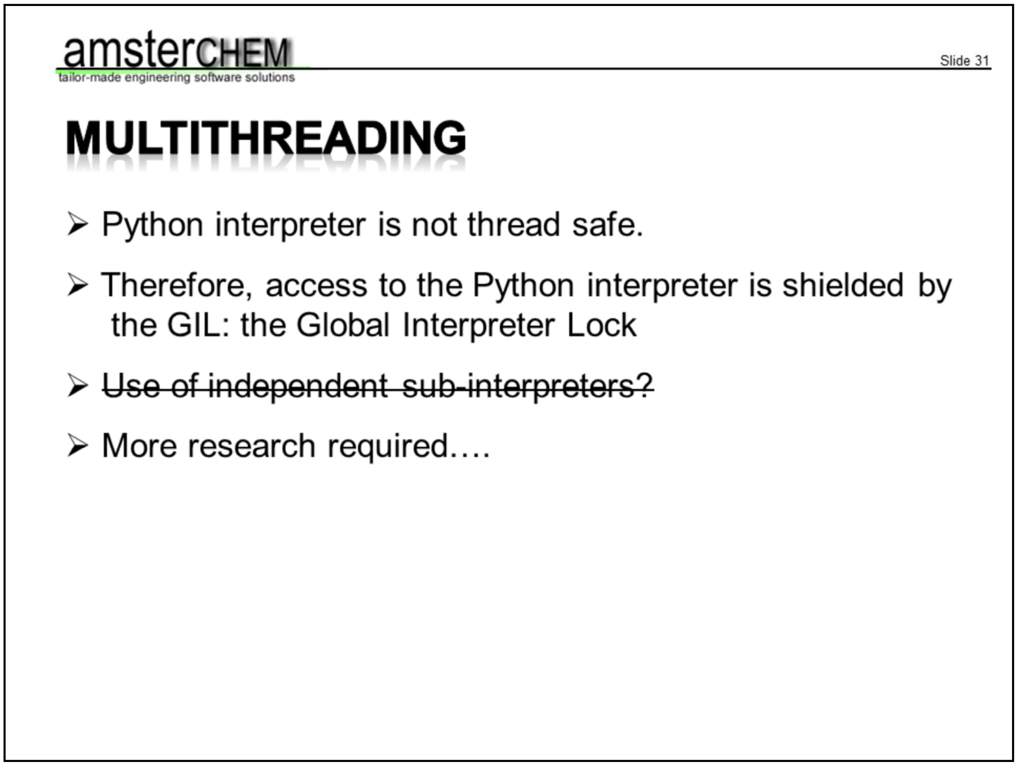
https://docs.python.org/3/c-api/init.html#sub-interpreter-support

Also note that combining this functionality with PyGILState_* APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching `PyGILState_Ensure()` and `PyGILState_Release()` calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

We find this excerpt in the API documentation under Bugs and Caveats. I have highlighted the bit of interest that says that anybody that does not have the same plan may face broken essential functionality, including ctypes, which we we surely use for native interop. Keep in mind that this is CAPE-OPEN so there are bound to be 3rd party software products inside the process, that may not appreciate what you are doing if you are accessing this API.



The PEP that suggested the sub interpreters API says this. So whether this is a viable route to avoid the GIL is, well, questionable.



I would carefully say that before jumping on a COBIA Python language binding, more research into the whole threading issue is required.

As said, all Python Unit Operations share the same Python. Therefore, each Unit Operation shields use of the Python interpreter by using the GIL.

Work is being done on making a Python interpreter that is thread safe, but as far as I know this is not publicly available yet. At least not main stream. Perhaps in a few years this problem will resolve itself.

amsterCHEM

tailor-made engineering software solutions

Slide 32

LINGERING REFERENCES

- CAPE-OPEN defines finite life span of some objects
- Problem is not unique to Python: Garbage Collection (.NET, java)
- Python keeps references of a stack frame
  - `last_type, last_value, last_traceback`
- Note these variables are global, and we may be sharing Python

Finally the CAPE-OPEN standards says that when a PMC is terminated, all external references must be dropped. There are other contexts in which you are not supposed to keep references to objects, for example, you should not cache a duplicate material object past the Calculate call of a unit operation. That this leads to problems is not unique to Python. In .NET and java for example, objects are not reference counted as they are in COBIA, COM and Python, but memory management is obtained though the process of garbage collection, which means that objects may not actually be destroyed until well after they are no longer referenced. In various .NET implementations this leads to having to carefully figure out which objects are related to each other and which not, and explicitly telling the .NET marshaler to drop the COM binding of objects when needed, well before garbage collection.

The lingering references in Python come from a corner that surprised me somewhat. When an error is raised, the Python interpreter funnily saves some global debugging values: last_type, last_value, last_traceback. The latter contains the entire stack trace, including all variables on the stack on each frame. Clearly if there are some external CAPE-OPEN variables in there, we did not actually release them in time.

As a cherry on the cake, these are global variables. Again something we would like to avoid manipulating in case we are sharing the Python instance with others.

Ok – that completes my summary of potential Python issues in an interop framework. Some final notes.

amsterCHEM
tailor-made engineering software solutions

CONCLUDING REMARKS

- Python Unit Operation available from AmsterCHEM:

  https://www.amsterchem.com/pythonunitoperation.html

- A similar module to use CAPE-OPEN thermo in Python:

  https://www.amsterchem.com/pythonthermo.html

- Free for academic use
- Give it a test spin: 1 month trial for non-academic use.

LEARNINGS FOR COBIA

- Several issues to be sorted for COBIA-Python binding

Thank you for your attention.