Public

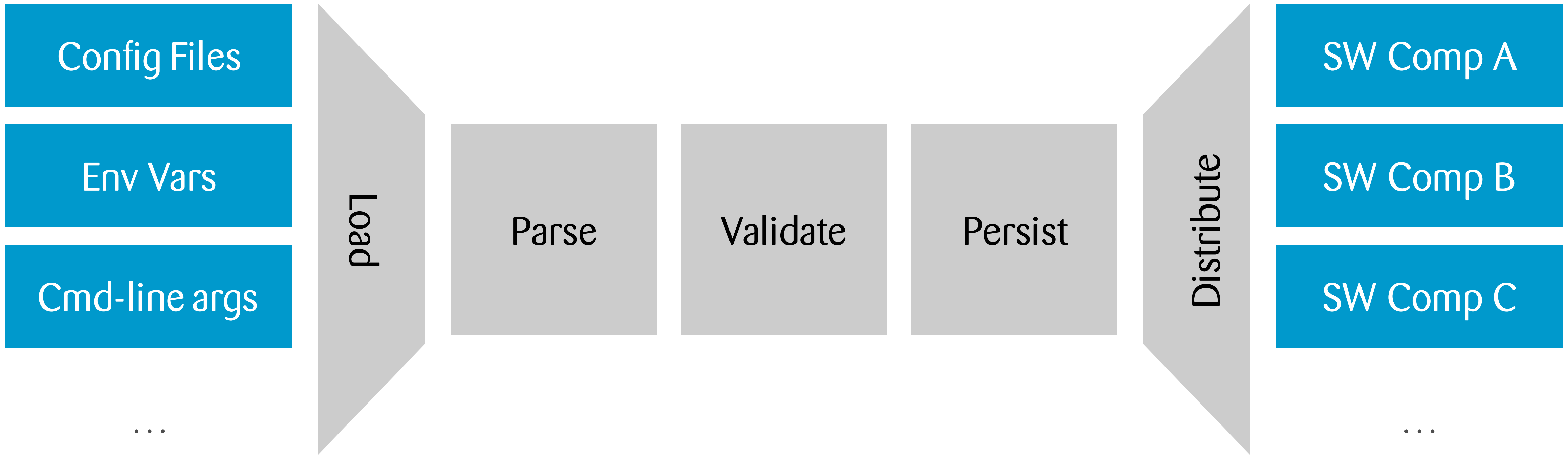
zühlke
empowering ideas

ConfZ

Pydantic Configuration Management

Silvan Melchior | 22.09.2022 | Swiss Python Summit

Config Management



Existing Solutions

Code these steps in almost all programs ➔ Use a library

Many libraries, but all miss something

- Diverse sources
- Validation
- Multiple environments
- Unit test support
- IDE support
- …

ConfZ

Config management library developed & maintained by Zühlke

open source (MIT License)

~1 year old

https://github.com/Zuehlke/ConfZ

~150 ★

https://confz.readthedocs.io/

```
pip install confz
```

Basic Principles

Thin layer around Pydantic

- Pydantic: Validation, IDE support (great dev-experience)
- Load data from sources
- Pass to Pydantic
- Cache output (global singleton with lazy loading or local instance with instant loading)

Pydantic:
"dataclasses with validation and rich types"

Allow a heterogenous set of sources (so far: files, env-vars, cmd-line args, constants)

- Easily extendable

Special support for

- Multiple environments
- Testing

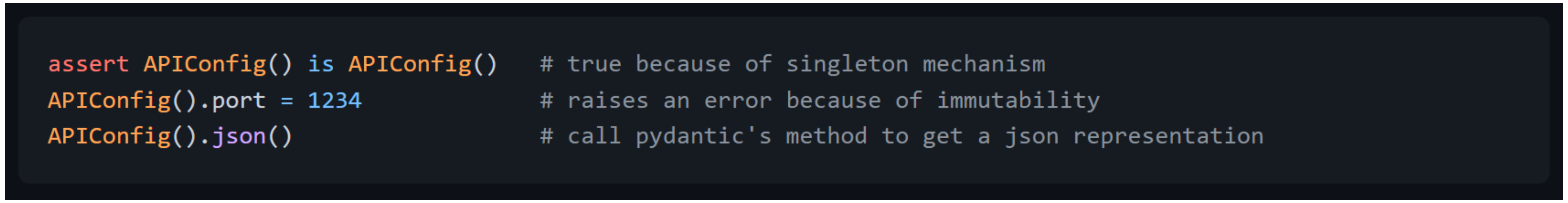
Usage by Example

Basic Usage

```python
from pathlib import Path

from confz import ConfZ, ConfZFileSource
from pydantic import SecretStr, AnyUrl

class DBConfig(ConfZ):
    user: str
    password: SecretStr

class APIConfig(ConfZ):
    host: AnyUrl
    port: int
    db: DBConfig

    CONFIG_SOURCES = ConfZFileSource(file=Path('/path/to/config.yml'))
```

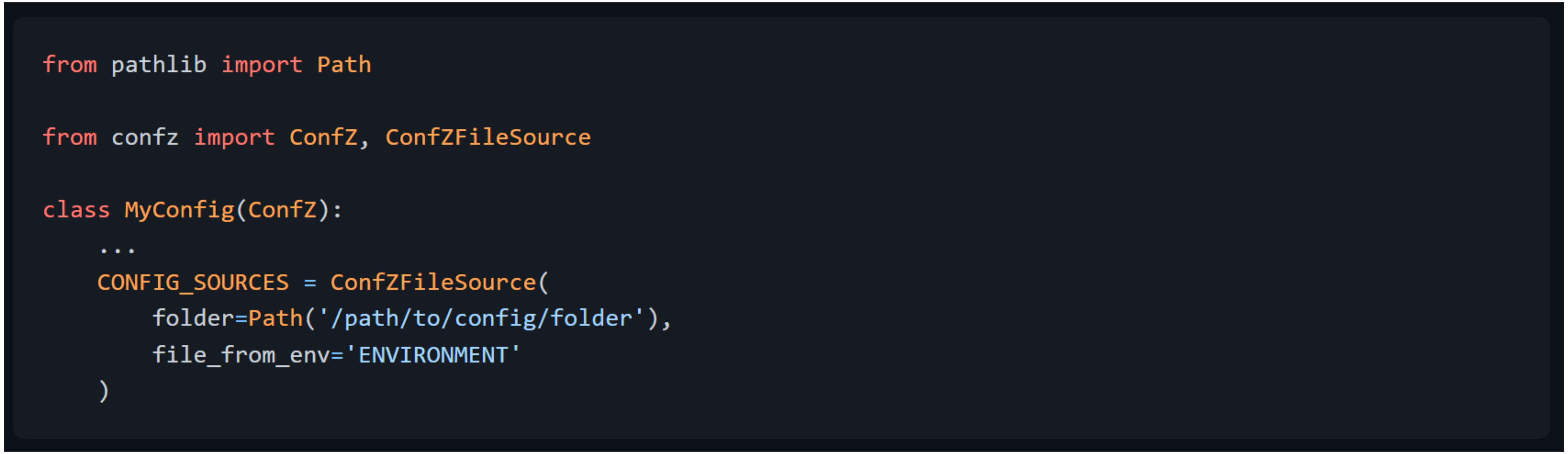
```python
from config import APIConfig

print(f"Serving API at {APIConfig().host}, port {APIConfig().port}.")
```

Caching, Lazy Loading, Immutability



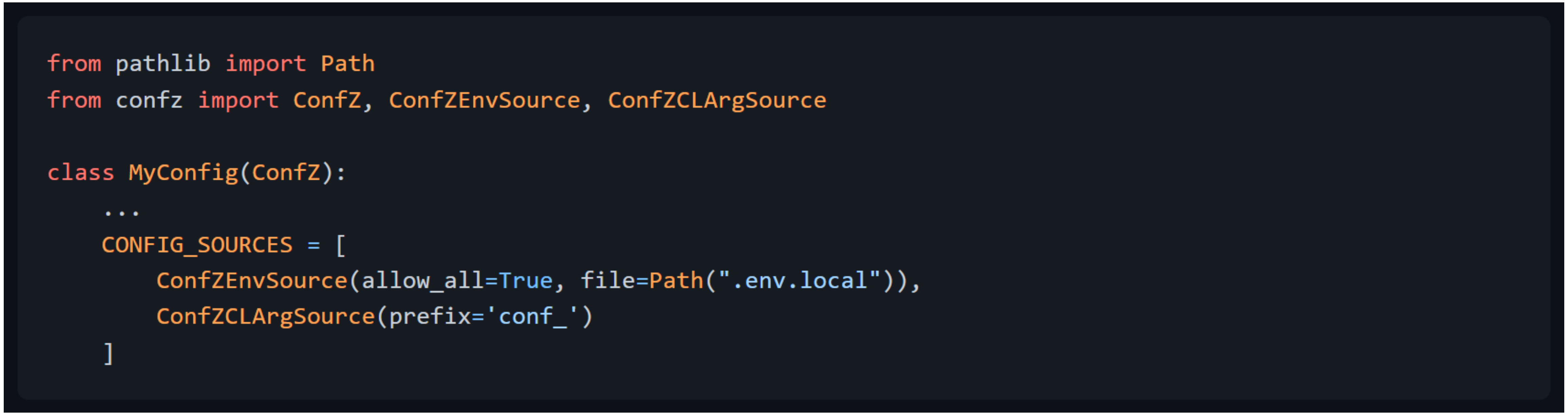
```python
assert APIConfig() is APIConfig()   # true because of singleton mechanism
APIConfig().port = 1234             # raises an error because of immutability
APIConfig().json()                  # call pydantic's method to get a json representation
```

Multiple Environments



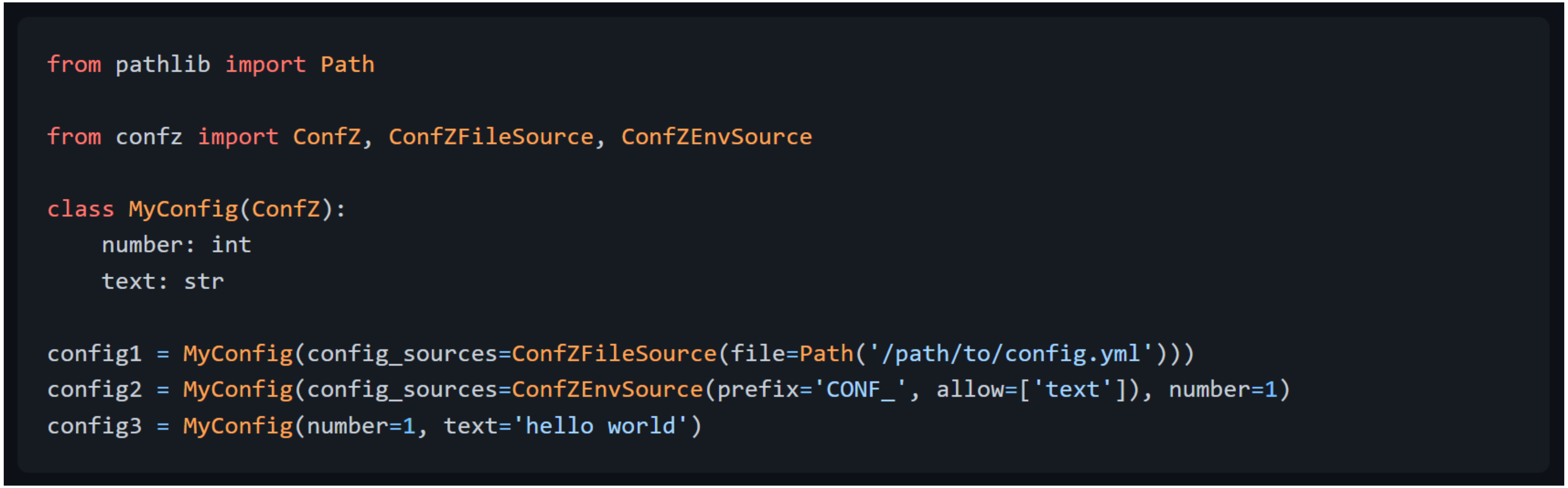
```python
from pathlib import Path

from confz import ConfZ, ConfZFileSource

class MyConfig(ConfZ):
    ...
    CONFIG_SOURCES = ConfZFileSource(
        folder=Path('/path/to/config/folder'),
        file_from_env='ENVIRONMENT'
    )
```

More Config Sources



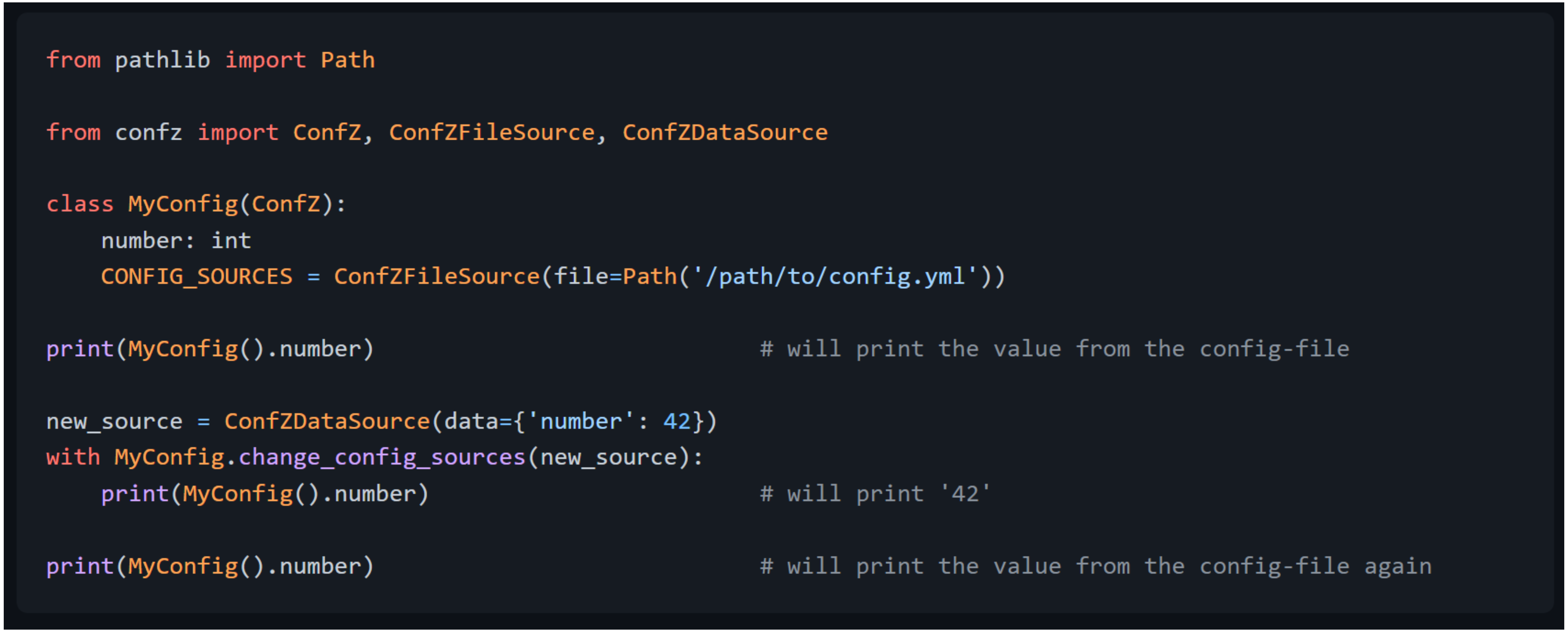
```python
from pathlib import Path
from confz import ConfZ, ConfZEnvSource, ConfZCLArgSource

class MyConfig(ConfZ):
    ...
    CONFIG_SOURCES = [
        ConfZEnvSource(allow_all=True, file=Path(".env.local")),
        ConfZCLArgSource(prefix='conf_')
    ]
```

Local Configs



```python
from pathlib import Path

from confz import ConfZ, ConfZFileSource, ConfZEnvSource

class MyConfig(ConfZ):
    number: int
    text: str

config1 = MyConfig(config_sources=ConfZFileSource(file=Path('/path/to/config.yml')))
config2 = MyConfig(config_sources=ConfZEnvSource(prefix='CONF_', allow=['text']), number=1)
config3 = MyConfig(number=1, text='hello world')
```

Overwrite Config for Testing



```python
from pathlib import Path

from confz import ConfZ, ConfZFileSource, ConfZDataSource

class MyConfig(ConfZ):
    number: int
    CONFIG_SOURCES = ConfZFileSource(file=Path('/path/to/config.yml'))

print(MyConfig().number)                            # will print the value from the config-file

new_source = ConfZDataSource(data={'number': 42})
with MyConfig.change_config_sources(new_source):
    print(MyConfig().number)                        # will print '42'

print(MyConfig().number)                            # will print the value from the config-file again
```

Under the Hood: Metaclasses in Python

Python Metaclasses

Metaprogramming:

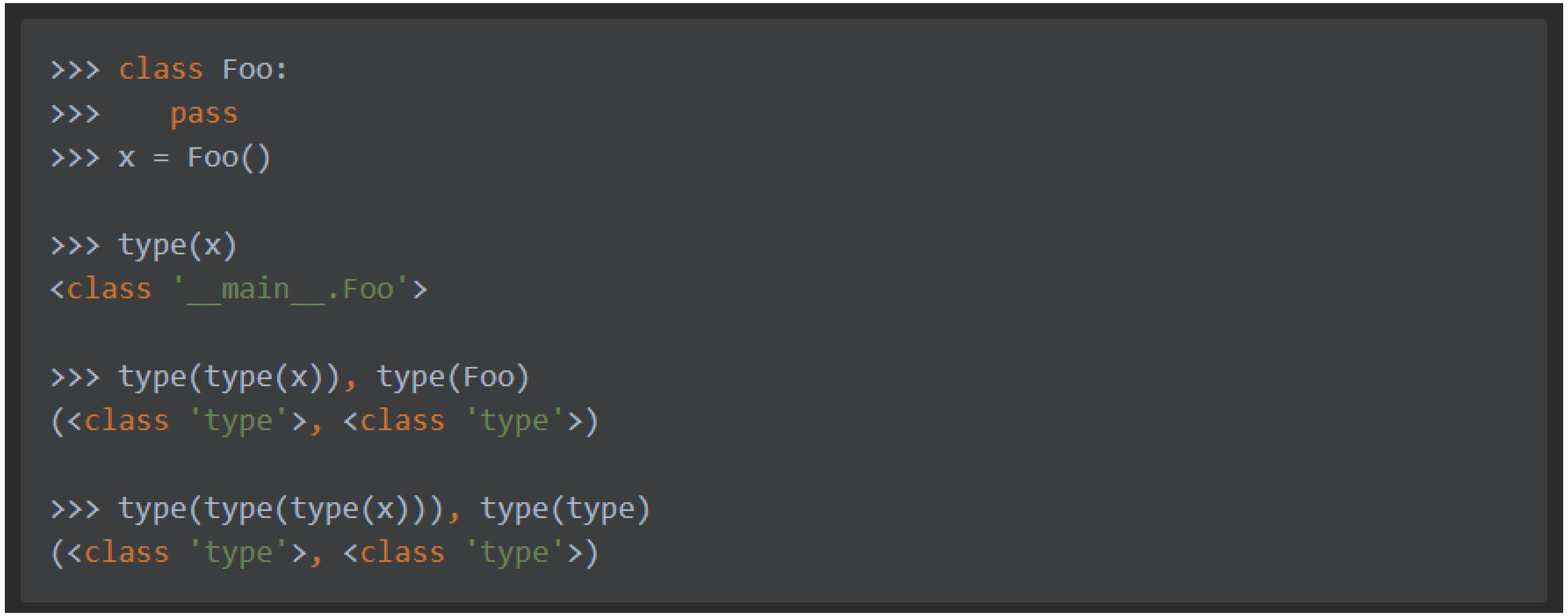
- Potential for a program to have knowledge of or manipulate itself
- Metaclasses are everywhere in Python, but you normally do not see them

Tim Peters: *“Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don’t. The people who actually need them know with certainty that they need them, and don’t need an explanation about why.”*

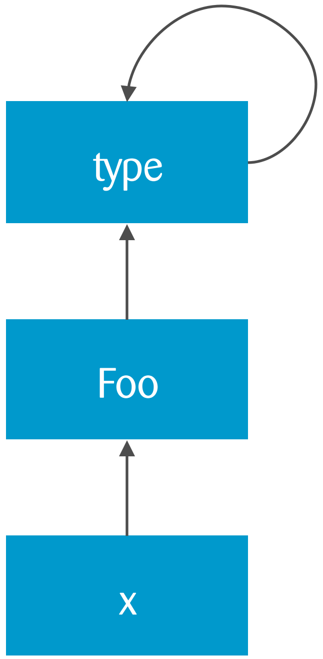
Still: Understanding them helps to understand the internals of Python

Intuitive description: A metaclass is to a class the same as a class is to an instance

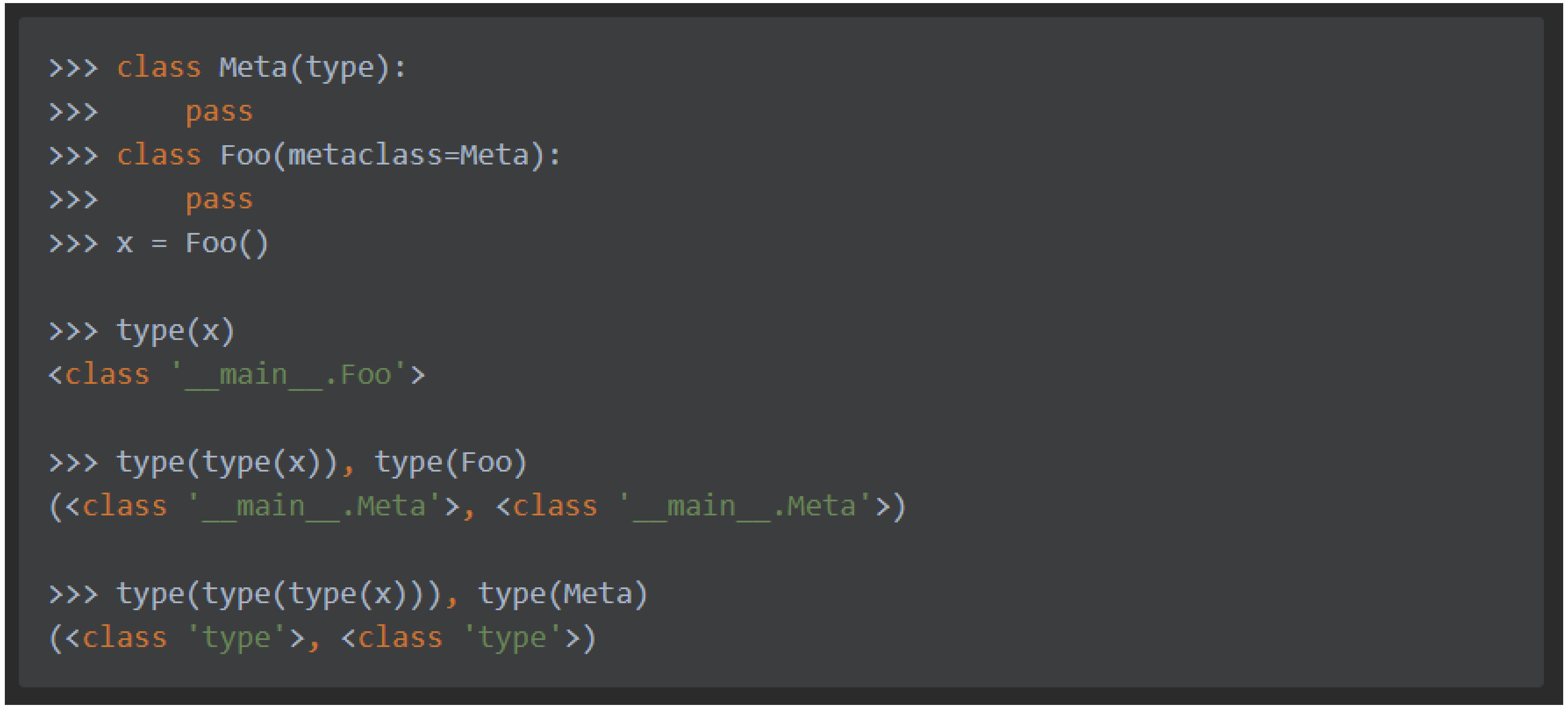
In Python, everything is an object



```
>>> class Foo:
>>>     pass
>>> x = Foo()

>>> type(x)
<class '__main__.Foo'>

>>> type(type(x)), type(Foo)
(<class 'type'>, <class 'type'>)

>>> type(type(type(x))), type(type)
(<class 'type'>, <class 'type'>)
```



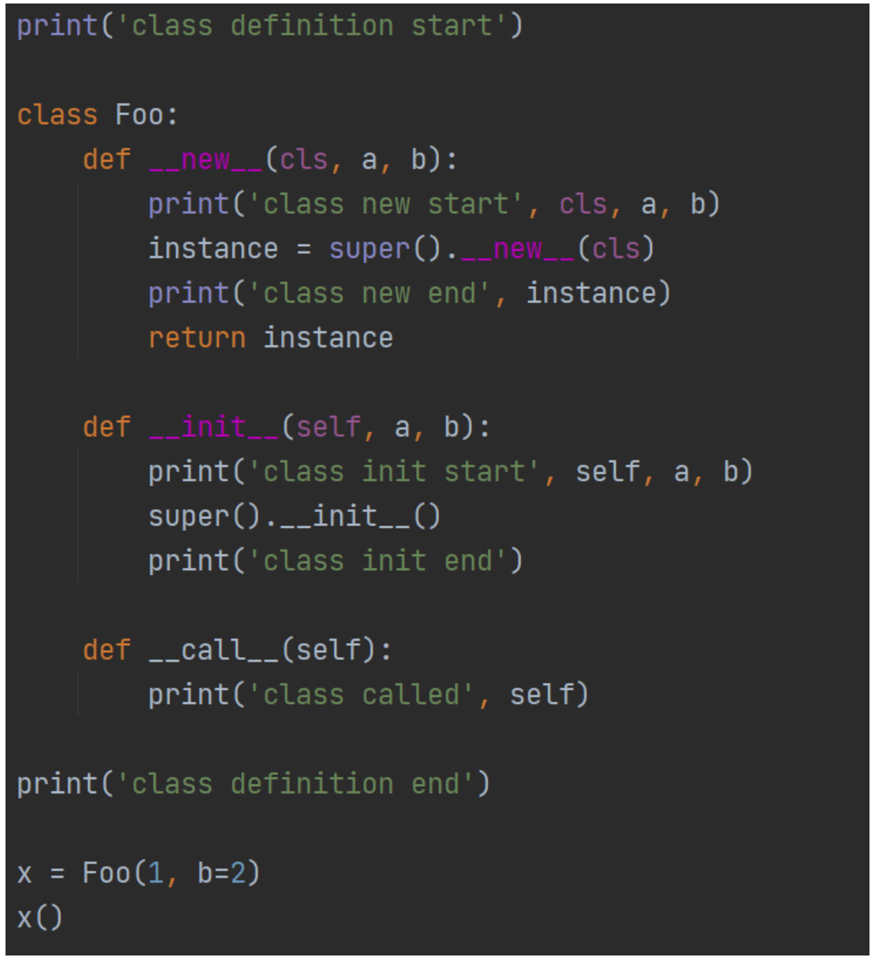
Defining your own “type”: A Metaclass



```
>>> class Meta(type):
>>>     pass
>>> class Foo(metaclass=Meta):
>>>     pass
>>> x = Foo()

>>> type(x)
<class '__main__.Foo'>

>>> type(type(x)), type(Foo)
(<class '__main__.Meta'>, <class '__main__.Meta'>)

>>> type(type(type(x))), type(Meta)
(<class 'type'>, <class 'type'>)
```

$$\begin{tikzcd} \text{type} \arrow[loop above] \\ \text{Meta} \arrow[u] \\ \text{Foo} \arrow[u] \\ \text{x} \arrow[u] \end{tikzcd}$$

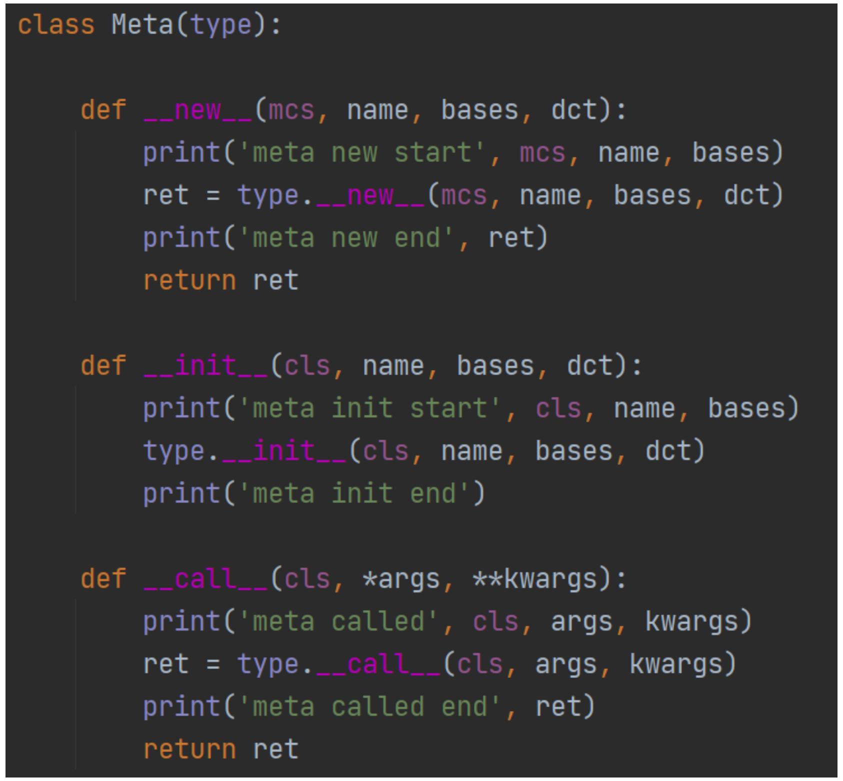
Creating an Instance



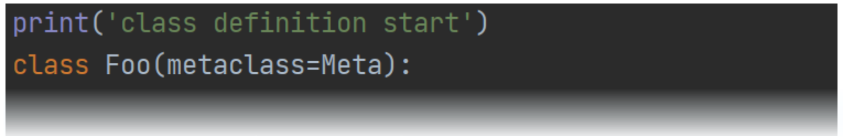
```python
print('class definition start')

class Foo:
    def __new__(cls, a, b):
        print('class new start', cls, a, b)
        instance = super().__new__(cls)
        print('class new end', instance)
        return instance

    def __init__(self, a, b):
        print('class init start', self, a, b)
        super().__init__()
        print('class init end')

    def __call__(self):
        print('class called', self)

print('class definition end')

x = Foo(1, b=2)
x()
```

```
class definition start
class definition end

class new start <class '__main__.Foo'> 1 2
class new end <__main__.Foo object at ...>

class init start <__main__.Foo object at ...> 1 2
class init end

class called <__main__.Foo object at ...>
```

Creating a Class



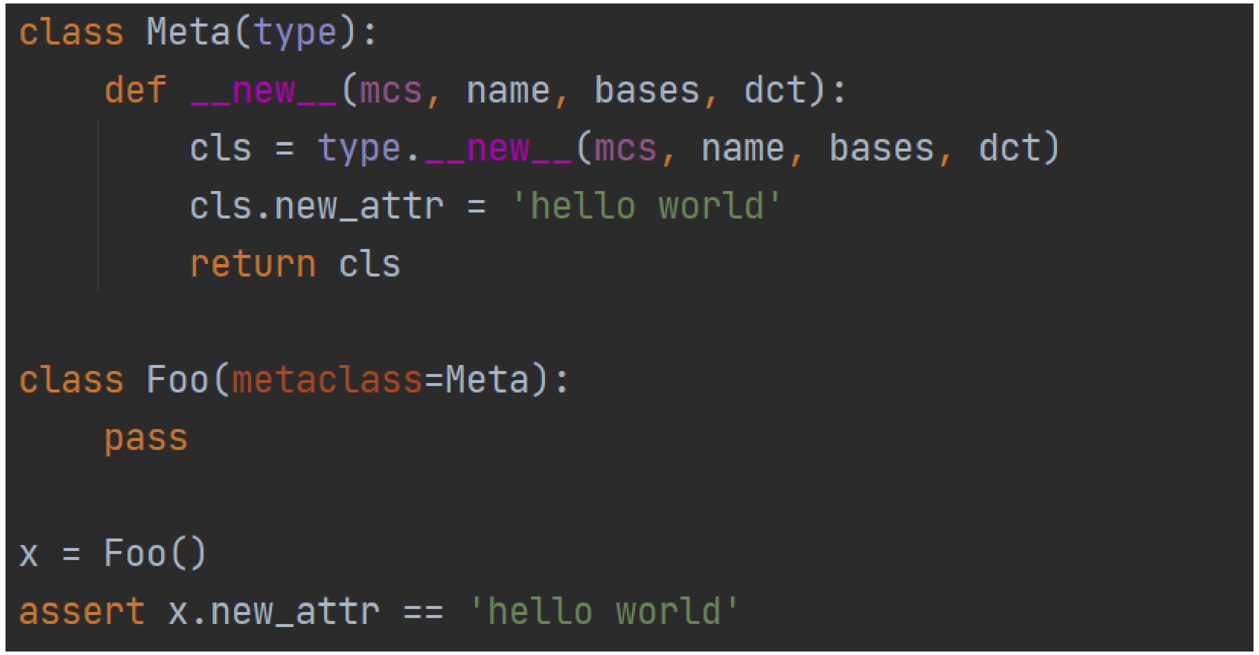
```python
class Meta(type):

    def __new__(mcs, name, bases, dct):
        print('meta new start', mcs, name, bases)
        ret = type.__new__(mcs, name, bases, dct)
        print('meta new end', ret)
        return ret

    def __init__(cls, name, bases, dct):
        print('meta init start', cls, name, bases)
        type.__init__(cls, name, bases, dct)
        print('meta init end')

    def __call__(cls, *args, **kwargs):
        print('meta called', cls, args, kwargs)
        ret = type.__call__(cls, args, kwargs)
        print('meta called end', ret)
        return ret
```



```python
print('class definition start')
class Foo(metaclass=Meta):
```

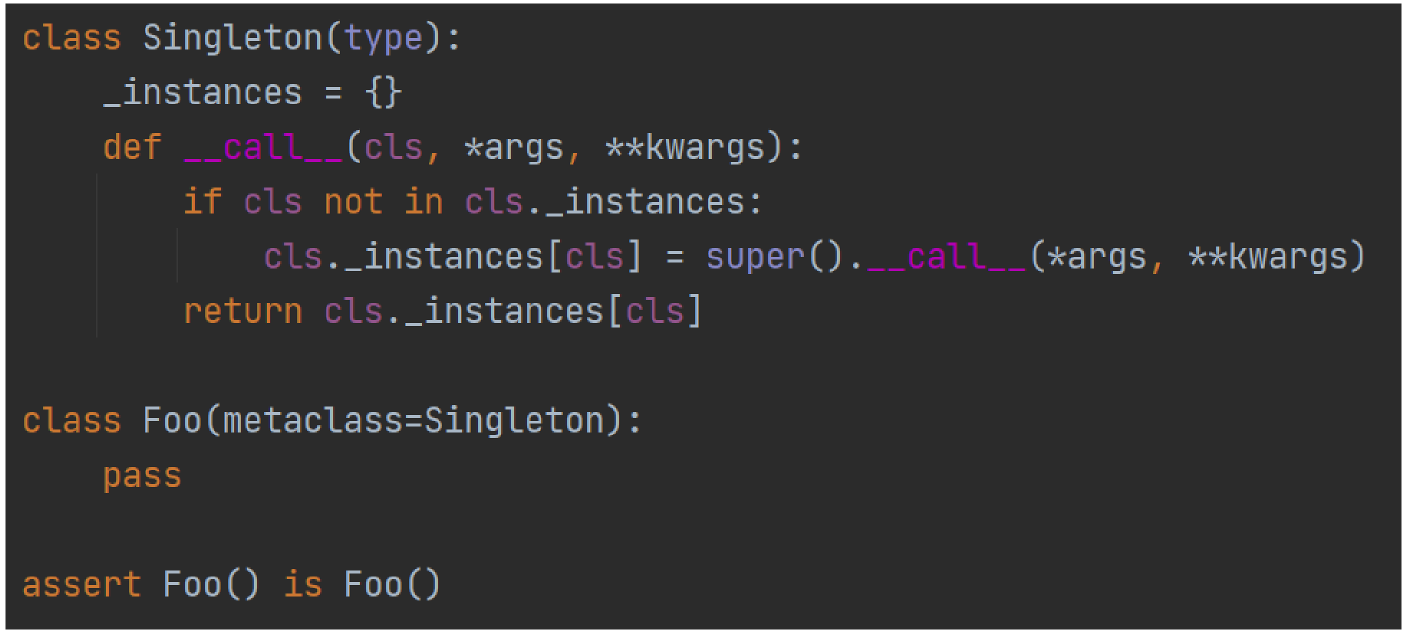
```
class definition start

meta new start <class '__main__.Meta'> Foo ()
meta new end <class '__main__.Foo’>

meta init start <class '__main__.Foo'> Foo ()
meta init end

class definition end

meta called <class '__main__.Foo'> (1,) {'b': 2}

class new start <class '__main__.Foo’> (1,) {'b': 2}
class new end <__main__.Foo object at ...>

class init start <__main__.Foo object at ...> ...
class init end

meta called end <__main__.Foo object at ...>

class called <__main__.Foo object at ...>
```

Dummy Example: Add attribute



```python
class Meta(type):
    def __new__(mcs, name, bases, dct):
        cls = type.__new__(mcs, name, bases, dct)
        cls.new_attr = 'hello world'
        return cls

class Foo(metaclass=Meta):
    pass

x = Foo()
assert x.new_attr == 'hello world'
```

Example: Singleton



```python
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Foo(metaclass=Singleton):
    pass

assert Foo() is Foo()
```

Where are Metaclasses used?

Mostly in libraries that allow you to define APIs as classes
(library modifies your class definition to work with library)

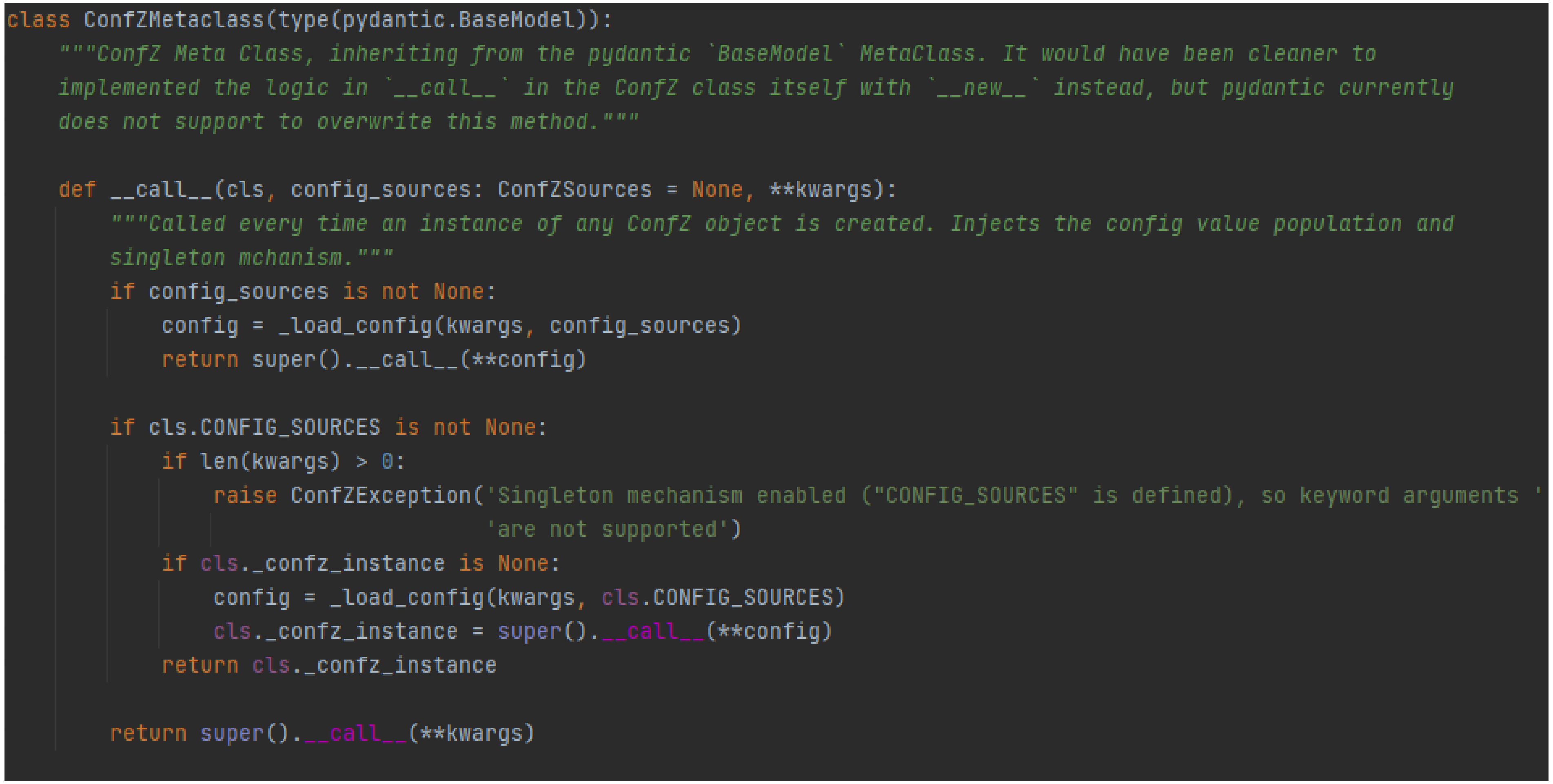
Examples:

- Django ORM
- Pydantic
- ...

Reminder: You most certainly do not need them in your daily work

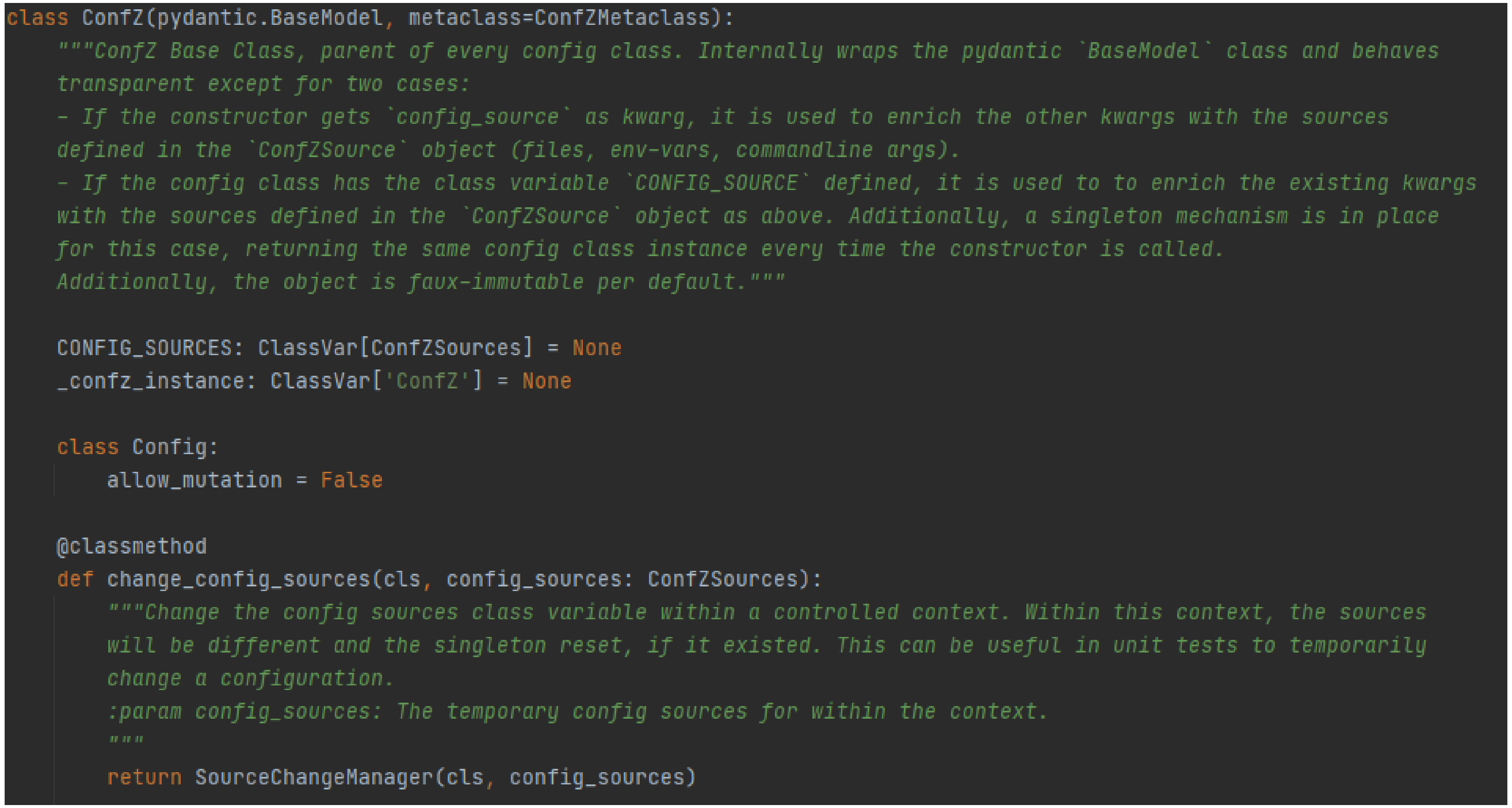
ConfZ: To wrap around Pydantic, had to wrap around its meta-class

Metaclasses in ConfZ



```
class ConfZMetaclass(type(pydantic.BaseModel)):
    """ConfZ Meta Class, inheriting from the pydantic `BaseModel` MetaClass. It would have been cleaner to
    implemented the logic in `__call__` in the ConfZ class itself with `__new__` instead, but pydantic currently
    does not support to overwrite this method."""

    def __call__(cls, config_sources: ConfZSources = None, **kwargs):
        """Called every time an instance of any ConfZ object is created. Injects the config value population and
        singleton mchanism."""
        if config_sources is not None:
            config = _load_config(kwargs, config_sources)
            return super().__call__(**config)

        if cls.CONFIG_SOURCES is not None:
            if len(kwargs) > 0:
                raise ConfZException('Singleton mechanism enabled ("CONFIG_SOURCES" is defined), so keyword arguments '
                                     'are not supported')
            if cls._confz_instance is None:
                config = _load_config(kwargs, cls.CONFIG_SOURCES)
                cls._confz_instance = super().__call__(**config)
            return cls._confz_instance

        return super().__call__(**kwargs)
```

Metaclasses in ConfZ



```python
class ConfZ(pydantic.BaseModel, metaclass=ConfZMetaclass):
    """ConfZ Base Class, parent of every config class. Internally wraps the pydantic `BaseModel` class and behaves
    transparent except for two cases:
    - If the constructor gets `config_source` as kwarg, it is used to enrich the other kwargs with the sources
    defined in the `ConfZSource` object (files, env-vars, commandline args).
    - If the config class has the class variable `CONFIG_SOURCE` defined, it is used to to enrich the existing kwargs
    with the sources defined in the `ConfZSource` object as above. Additionally, a singleton mechanism is in place
    for this case, returning the same config class instance every time the constructor is called.
    Additionally, the object is faux-immutable per default."""

    CONFIG_SOURCES: ClassVar[ConfZSources] = None
    _confz_instance: ClassVar['ConfZ'] = None

    class Config:
        allow_mutation = False

    @classmethod
    def change_config_sources(cls, config_sources: ConfZSources):
        """Change the config sources class variable within a controlled context. Within this context, the sources
        will be different and the singleton reset, if it existed. This can be useful in unit tests to temporarily
        change a configuration.
        :param config_sources: The temporary config sources for within the context.
        """
        return SourceChangeManager(cls, config_sources)
```

zühlke
empowering ideas