programming 3

exercise 4 .

If you want to follow this document's changes, click on the speech bubble next to the **Share** button on the top right, select **Notifications**, and click **All**.

This exercise has 2 parts:

1. [Exercises about context managers and metaclasses](#_zs343f6f01l).
2. [Further developments in our cool system](#_5v62bb3hv6d8).

To begin:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** clone git@github.com:advanced-system-design/exercise-4.git  $ **cd** exercise-4/  $ ./scripts/install.sh  $ **source** .env/bin/activate | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** remote remove origin  $ **git** remote add origin \ git@github.com:advanced-system-design/exercise-4-\*.git | |  |
|  |

And when you're done:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** add .  $ **git** commit -m 'Submitting exercise 4.'  $ **git** push origin master | |  |
|  |

# exercises .

First, enter the **q1/** directory.

1. In **e1.py**, write a context manager that measures the time spent within the context and provides **started**, **stopped** and **elapsed** attributes.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **with** Timer() **as** timer:  ... time.sleep(1)  >>> timer.elapsed  1.0 | |  |
|  |

1. In **e2.py**, write a context manager that suppresses exceptions from a given type. If no types are given, it suppresses all exceptions.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **with** Suppress(**NameError**, **TypeError**):  ... **raise** **NameError**()  # Nothing!  >>> **with** Suppress(**NameError**, **TypeError**):  ... **raise** **TypeError**()  # Nothing!  >>> **with** Suppress(**NameError**, **TypeError**):  ... **raise** **ValueError**()  **ValueError**  >>> **with** Suppress():  ... **raise** **ValueError**()  # Nothing! | |  |
|  |

This actually exists; it's called **contextlib.suppress**, and you should use it instead of **try: … except: pass** blocks.

1. In **e3.py**, write a context manager that captures **stdout** and provides it as its **value** attribute.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| # capture prints  **with** StandardOutput() **as** stdout:  **print**('Hello, world!')  **assert** stdout.value == 'Hello, world!\n'  # or just silence them  **with** StandardOutput():  **print**('Dead men tell no tales') | |  |
|  |

1. In **e4a.py**, write a context manager that creates a temporary file, returns its path, and later deletes it.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **with** TemporaryFile() **as** f:  ... **print**(os.path.isfile(f))  True  >>> os.path.exists(f)  False | |  |
|  |

In **e4b.py**, write a context manager that creates a temporary directory, returns its path, and later deletes it.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **with** TemporaryDirectory() **as** d:  ... **print**(os.path.isdir(d))  True  >>> os.path.exists(d)  False | |  |
|  |

1. In **e5.py**, rewrite the **timer**, **suppress**, **standard\_output**, **temporary\_file** and **temporary\_directory** context managers using **contextlib.contextmanager**.

Which of these were easier to implement as classes, and which as generators?

1. In **e6.py**, implement **contextlib.contextmanager** as **ContextManager**.
2. In **e7.py**, write the **Extended** metaclass, which extends every method of its classes to their thread-safe and an exception-safe versions.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **class** **A**(metaclass=Extended):  ... **def** **f**(self):  ... …  >>> a = A()  >>> a.f()  >>> a.f\_sync() # same as f, but thread safe (synchronized in respect   # to the class's lock)  >>> a.f\_safe() # same as f, but exception safe (doesn't raise exceptions) | |  |
|  |

1. In **e8.py**, write the **Overloaded** metaclass, which supports function overloading.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **class** A(metaclass=Overloaded):  ... **def** **f**(self):  ... **return** 1  >>> **def** **f**(self, x):  ... **return** 2  >>> a = A()  >>> a.f()  1  >>> a.f(**None**)  2 | |  |
|  |

# 

# back to our system

The networking utilities we wrote last time were somewhat incomplete, because we were missing on context managers and class methods. So, in **connection.py**, add:

* A managed context, after which the connection is closed.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **with** connection:  connection.send(b'Hello, world!') | |  |
|  |

* A **connect(host, port)** class method, which connects to the specified host and port, and returns a **Connection** object for this connection.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| connection = Connection.connect('127.0.0.1', 8000)  # or, even better:  **with** Connection.connect('127.0.0.1', 8000) **as** connection:  connection.send(thought.serialize()) | |  |
|  |

And in **listener.py**, add:

* A managed context, in which it starts listening, and by the end of which it stops. So this code would listen for 2 seconds, then stop:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **with** Listener(8000) **as** listener:  time.sleep(2) | |  |
|  |