programming 2

exercise 3 .

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 3 parts:

1. [Questions to test yourself](#_tjmfgj12xerr) — no need to submit those.
2. [Exercises about generators, classes, and descriptors](#_zs343f6f01l).
3. [Using our newly gained understanding of object-oriented programming to tidy up our serialization and networking code](#_5v62bb3hv6d8).

To begin:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** clone git@github.com:advanced-system-design/exercise-3.git  $ **cd** exercise-3/  $ ./scripts/install.sh  $ **source** .env/bin/activate | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** remote remove origin  $ **git** remote add origin \ git@github.com:advanced-system-design/exercise-3-\*.git | |  |
|  |

And when you're done:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** add .  $ **git** commit -m 'Submitting exercise 3.'  $ **git** push origin master | |  |
|  |

# test yourself

Same as before, there's no need to submit those questions.

1. Complete the **?**s:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **class** **A**:  **def** **f**(self):  **return** 1  **def** **g**(self):  **return** 2  **class** **B**(A):  **def** **f**(self):  **return** 3  **class** **C**:  **def** **g**(self):  **return** 4  **class** **D**(A):  **def** **g**(self):  **return** 5  **class** **X**(B, C):  **pass**  x = X()  **assert** x.f() == **?**  **assert** x.g() == **?**  **class** **X**(C, D):  **pass**  x = X()  **assert** x.f() == **?**  **assert** x.g() == **?**  **class** **X**(D, C):  **pass**  x = X()  **assert** x.f() == **?**  **assert** x.g() == **?**  **class** **X**(B, D):  **pass**  x = X()  **assert** x.f() == **?**  **assert** x.g() == **?** | |  |
|  |

1. Complete the **?**s:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **class** **D1**:  **def** **\_\_get\_\_**(self, instance, cls):  **return** 1  **class** **D2**:  **def** **\_\_get\_\_**(self, instance, cls):  **return** 2  **def** **\_\_set\_\_**(self, instance, value):  **pass**  **class** **A**:  z = 3  **class** **B**(A):  x = D1()  y = D2()  **def** **\_\_init\_\_**(self):  self.x = 5  self.y = 6  **def** **\_\_getattr\_\_**(self, name):  **return** 7  b = B()  **assert** b.x == **?**  **assert** b.y == **?**  **assert** b.z == **?**  **assert** b.w == **?**  **assert** B.x == **?**  **assert** B.y == **?**  **assert** B.z == **?**  **assert** B.w == **?** | |  |
|  |

Can you think of a way to make the lines that raise an exception work?

1. Implement **method** and **property** yourself.

# 

# exercises .

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

1. Write a generator that traverse a tree data structure:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| tree = (1, [  (2, [  (4, []),  (5, []),  ]),  (3, [  (6, []),  (7, []),  ]),  ]) | |  |
|  |

In **e1a.py**, implement it using breadth-first search:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **for** value **in** bfs(tree):  ... **print**(value)  1 2 3 4 5 6 7 | |  |
|  |

In **e1b.py**, implement it using depth-first search:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **for** value **in** dfs(tree):  ... **print**(value)  1 2 4 5 3 6 7 | |  |
|  |

1. In **e2.py**, write a generator that traverses all the entries in a directory recursively, and returns an entry object with a **name** and a **type**. Don't use **os.walk**.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ tree fs/  fs  ├── d1  │ ├── d2  │ │ ├── d3  │ │ │ └── f3  │ │ └── d4  │ └── f2  └── f1 | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **for** e **in** walk('fs'):  ... **print**(f'{e.name} ({e.type})')  d1 (directory)  f1 (file)  d2 (directory)  f2 (file)  d3 (directory)  d4 (directory)  f3 (file) | |  |
|  |

Add a **skip()** method to entry objects, so skipped directory entries aren't traversed recursively.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **for** e **in** walk('fs'):  ... **print**(f'{e.name} ({e.type})')  ... **if** e.name == 'd2':  ... e.skip()  d1 (directory)  f1 (file)  d2 (directory)  f2 (file) | |  |
|  |

1. In **e3.py**, write the **TransformDict** class, which receives a transform function, and applies it to normalize its keys.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> td = TransformDict(**lambda** key: key.lower())  >>> td['FOO'] = 1  >>> td['foo'] = 2  >>> td['Foo']  2  >>> td['BAR'] = 3  >>> td  {'foo': 2, 'bar': 3} | |  |
|  |

Support all standard dictionary behaviours: representation, equality, boolean value, getting, setting and deleting an item, length, containment and iteration.

1. In **e4.py**, write the **MultiDict** class, which stores multiple values per key. Setting a value appends it to its key's list; getting and deleting operates on the oldest (first in) value, except for **get\_all(key)** and **delete\_all(key)**, which operate on all values.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> md = MultiDict()  >>> md['x'] = 1  >>> md['x'] = 2  >>> md['x']  1  >>> md.get\_all('x')  [1, 2]  >>> **del** md['x']  >>> md['x']  2  >>> md.delete\_all('x') | |  |
|  |

Support all standard dictionary behaviours: representation, equality, boolean value, getting, setting and deleting an item, length, containment and iteration.

1. In **e5.py**, write the **LazyVariable** class, which receives a name and returns an object able to capture any expression it's involved in as a **LazyExpression** instance, with the **evaluate(\*\*kwargs)** method that resolves itself given some variables' values.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> x = LazyVariable('x')  >>> x  x  >>> x + 1  (x + 1)  >>> 2\*x + 1  ((2 \* x) + 1)  >>> x.evaluate(x=2)  2  >>> y = 2\*x + 1  >>> y.evaluate(x=2)  5 | |  |
|  |

Support representation, addition, subtraction, multiplication and division, as well as the unary plus (**+x**) and minus (**-x**).

1. This exercise is **optional**: make the following "Linux shell" syntax work in Python.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> ls  d1 f1  >>> ls -l  total 0  drwxr-xr-x 2 user user 4096 Nov 10 18:02 d1  -rw-r--r-- 1 user user 1 Nov 10 18:02 f1  >>> wc -c < 'f1'  1  >>> ls -l | wc -c  93 | |  |
|  |

(There's no need to support anything except the commands shown here.)

1. In **e7.py**, Implement the following **TypedProperty** descriptor.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> **class** **A**:  ... x = TypedProperty(int)  ... **def** **\_\_init\_\_**(self, x):  ... self.x = x  >>> a = A(1)  >>> a.x  1  >>> a.x = 'foo'  ValueError: attribute 'x' must be int  >>> **del** a.x  >>> a.x  0 | |  |
|  |

The descriptor receives a type, and when you try to get it, returns the default value of this type. When you try to set this value, it makes sure it's an instance of the specified type, and if not, raises a **ValueError** (the exact error message is not important). Finally, when you try to delete this value, it actually "resets" it, so the next time you access it you'd get the default value again.  
The descriptor should return itself when accessed via the class (e.g. **A.x**).

# back to our system

1. So far, we've been passing thoughts as triplets of a user ID, a timestamp and a string — no longer! It's time to define the **Thought** class, which receives the three values, stores them as the **user\_id**, **timestamp** and **thought** attributes, and provides:

* A handy **\_\_repr\_\_** method, which would return something like:  
  Thought(user\_id=1, timestamp=datetime.datetime(2000, 1, 1, 12, 0), thought="I'm hungry")
* A handy **\_\_str\_\_** method, which would return something like:  
  [2000-01-01 12:00:00] user 1: I'm hungry
* An **\_\_eq\_\_** method, which would return true only for other thought instances with similar attributes.
* The **serialize()** method, which returns bytes representing this thought, ready to be sent over the wire.
* The **deserialize(data)** class method, which given some bytes, decodes them and construct a new thought instance accordingly.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> thought1 = Thought(1, datetime(2000, 1, 1, 12, 0), "I'm hungry")  >>> thought1  Thought(user\_id=1, timestamp=datetime.datetime(2000, 1, 1, 12, 0), thought="I'm hungry")  >>> **print**(thought1) [2000-01-01 12:00:00] user 1: I'm hungry  >>> # continued | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> thought1.serialize()  b"\x01\x00\x00\x00\x00\x00\x00\x00…"  >>> thought2 = Thought.deserialize(\_)  >>> thought1 == thought2  True | |  |
|  |

Implement it in **thought.py**.

1. One of the nuisances that have blighted our codebase since the very beginning is networking code: there's quite a lot of work to be done to get a socket to listen on some port, and even more to receive exactly N bytes. Time to wrap it up in a class!  
   The **Connection** class receives a socket, and provides:

* A handy **\_\_repr\_\_** method, which would return something like:  
  <Connection from 127.0.0.1:65535 to 127.0.0.1:8000>
* The **send(data)** method, which sends all the data over the socket.
* The **receive(size)** method, which receives as many bytes as were specified by size, or throws an exception if the connection was closed before all the data was received.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| data = connection.receive(20)  **assert** **len**(data) == 20 | |  |
|  |

* A **close()** method, which closes the socket.

Implement it in **connection.py**.

The **Listener** class receives a port, and optionally a host (which defaults to **'0.0.0.0'**), a backlog (which defaults to **1000**), and whether the address should be reused (which defaults to **True**). It also provides:

* A handy **\_\_repr\_\_** method, which would return something like:  
  Listener(port=8000, host='0.0.0.0', backlog=1000, reuseaddr=True)
* A **start()** method, which starts listening, and a **stop()** method, which stops listening and closes the socket.
* An **accept()** method, which waits for a connection, accepts it, and returns a **Connection** object.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| listener = Listener(8000)  listener.start()  connection = listener.accept() | |  |
|  |

Implement it in **listener.py**.

Finally, copy over **client.py** and **server.py** from the previous question, and implement them to use our newly written utilities.