programming 1

exercise 2 .

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 looks like a lot; don't panic! It has 3 parts:

1. [Questions to test yourself](#_tjmfgj12xerr) — no need to submit those.
2. [Exercises about functions and decorators](#_8hj70lb30usi) — there are a lot, but they're mostly easy.
3. [Using our newly gained decorating skills to improve our CLI and webserver](#_ak10rlxyux3y).

To begin:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** clone git@github.com:advanced-system-design/exercise-2.git  $ **cd** exercise-2/  $ ./scripts/install.sh  $ **source** .env/bin/activate | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** remote remove origin  $ **git** remote add origin \ git@github.com:advanced-system-design/exercise-2-123456789.git | |  |
|  |

And when you're done:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **git** add .  $ **git** commit -m 'Submitting exercise 2.'  $ **git** push origin master | |  |
|  |

# test yourself

There's no need to submit those questions, but they're here to help you make sure you understand what we talked about in class.

1. Unpack the values **1..5** into the variables **a..e** for the following expressions:
   * [1, 2, 3, 4, 5]
   * [[1], [2], [3], [4], [5]]
   * [1, [2, [3, [4, [5]]]]]
   * [1, 0, [2, [3], 4, 5, 0, 0]]
   * {1, 2, 3, 4, 5}
2. Complete the **?**s:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| numbers = [[1, 2, 3], [1, 2, 3], [1, 2, 3]]  **for** ?:  **assert** x == 1  **assert** y == 2  **assert** z == 3 | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| numbers = [[1, [2, [3]]], [1, [2, [3]]], [1, [2, [3]]]]  **for** ?:  **assert** x == 1  **assert** y == 2  **assert** z == 3 | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| numbers = [[1], [1], [1], [1], [1]]  **for** ?:  **assert** x == 1 | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| numbers = [[1, 2], [1, 2, 3], [1, 2, 3, 4]]  **for** ?:  **assert** x == 1  **assert** y == 2 | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| numbers = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}  **for** ?:  **assert** x\*\*2 == y | |  |
|  |

1. Complete the **?**s:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| a = 1  b = 2  c = 3  **def** f():  **locals**()['a'] = 4  **globals**()['a'] = 5  b = 6  c = 7  **def** g():  **global** b  **nonlocal** c  b = 8  c = 9  **assert** a == **?**  **assert** b == **?**  **assert** c == **?**  g()  **assert** a == **?**  **assert** b == **?**  **assert** c == **?**  f()  **assert** a == **?**  **assert** b == **?**  **assert** c == **?** | |  |
|  |

1. Complete the **?**s:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| a = 1  **if** **True**:  a = 2  **assert** a == **?**  b = 3  **for** b **in** **range**(10):  **print**(b)  **assert** b == **?**  c = 4  **class** **A**:  c = 5  **def** **f**(self):  **return** c  a = A()  **assert** c == **?**  **assert** a.c == **?**  **assert** a.f() == **?** | |  |
|  |

1. Complete the **?**s:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| As = []  **for** i **in** **range**(5):  **class** **A**:  x = i  **def** **f**(self):  **return** i  **def** **g**(self):  **return** self.x  As.append(A)  a = As[2]()  **assert** a.x == **?**  **assert** a.f() == **?**  **assert** a.g() == **?** | |  |
|  |

1. Write a loop that iterates over **['a', 'b', 'c']** and prints:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| c 1  b 2  a 3 | |  |
|  |

1. Write a loop that iterates over **['x', 'y', 'z']** and **['a', 'b', 'c', 'd', 'e']** and prints:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| x e  y d  z c | |  |
|  |

Write a similar loop that prints:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| x a 1  y b 2  z c 3  \_ d 4  \_ e 5 | |  |
|  |

1. Write a loop that, given a number **n**, finds its smallest prime factor in **p**; if the number is prime, **p** should end up as **1**.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| n = **int**(sys.argv[1])  # Your loop here  **print**(p) | |  |
|  |

Rewrite it as a function. Which one is better?

1. Write a loop that, given a number **1 < n < 100**, prints the 10 by 10 multiplication table until it reaches a product equal to **n**, at which point it prints **X** and stops.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ python e9.py 25  1 2 3 4 5 6 7 8 9 10  2 4 6 8 10 12 14 16 18 20  3 6 9 12 15 18 21 24 27 30  4 8 12 16 20 24 28 32 36 40  5 10 15 20 X | |  |
|  |

Rewrite it as a function. Which one is better?

1. **add\_defaults** receives a dictionary and adds some standard keys to it. If no dictionary is specified, it allocates a new one.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **def** **add\_defaults**(config={}):  config['host'] = '0.0.0.0'  config['port'] = 8000  config['log\_path'] = '/tmp/log'  **return** config | |  |
|  |

Can you spot the bug and fix it? Then, take at look at this implementation:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **def** **add\_defaults**(config=**None**):  **if** **not** config:  config = {}  config['host'] = '0.0.0.0'  config['port'] = 8000  config['log\_path'] = '/tmp/log'  **return** config | |  |
|  |

Can you spot the bug and fix it?

1. **get\_permissions** receives a list of root permissions and a list of user permissions, and returns the list of permissions befitting the current user.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **def** **get\_permissions**(root\_permissions, user\_permissions):  **return** os.getuid() != 0 **and** user\_permissions **or** root\_permissions | |  |
|  |

Can you spot the bug and fix it? Can you do it in one line?

1. **count\_args** receives 0-2 arguments and returns how many arguments it received.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **def** **count\_args**(x=0, y=0):  **return** (x **and** 1) + (y **and** 1)  **assert** count\_args() == 0  **assert** count\_args(1) == 1  **assert** count\_args(1, 2) == 2 | |  |
|  |

Can you spot the bug and fix it? Can you do it in one line?

1. How would you refactor the following code?

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **import** datetime **as** dt  events = {  'critical': [  (dt.datetime(2000, 1, 1, 11, 0), 'server crashed'),  (dt.datetime(2000, 1, 1, 12, 0), 'server crashed')  ],  'standard': [  (dt.datetime(2000, 1, 1, 10, 1), 'server started'),  (dt.datetime(2000, 1, 1, 11, 1), 'server started'),  (dt.datetime(2000, 1, 1, 12, 1), 'server started')  ]  }  **def** **report\_events**(events, since=**None**, until=**None**):  **if** **len**(events.keys()) == 0:  **print**('no events')  **return**  **if** 'critical' **in** events.keys() **and** **len**(events['critical']) > 0:  **print**('critical events:')  **for** event **in** events['critical']:  **print**(' %s %s' % (event[0], event[1]))  **else**:  **print**('no critical events')  **if** ((since **is not** **None**) **and** (until **is not** **None**)):  **print**('standard events:')  **if** 'standard' **in** events.keys() **and** **len**(events['standard']) > 0:  **for** event **in** events['standard']:  **if** event[0] >= since **and** event[0] <= until:  **print**(' %s %s' % (event[0], event[1]))  report\_events(events, dt.datetime(2000, 1, 1, 10),  dt.datetime(2000, 1, 1, 12)) | |  |
|  |

After you've given it some thought, you can find my solution [here](https://docs.google.com/document/d/1bUQNrW6dS3UBfxsSdJVbmfkrAvJGW1JEJFjoHMUY3Y0/edit#).

# exercises .

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

1. Write some **one-liners** (note that d-g are **optional** — you **don't** have to do them):  
   In **e1a.py**, implement **file\_sizes**, which returns all the files mapped to their sizes in the current directory.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ ls  directory/ file.txt  $ cat file.txt  Hello, world! | |  |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> file\_sizes()  {'file.txt': 14} | |  |
|  |

In **e1b.py**, implement **sieve\_of\_eratosthenes**, which returns a set of all the primes up to a number.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> sieve\_of\_eratosthenes(100)  {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97} | |  |
|  |

In **e1c.py**, implement **is\_prime**, which returns whether a number is prime.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> is\_prime(10)  False  >>> is\_prime(13)  True | |  |
|  |

In **e1d.py**, simulate [itertools.product](https://docs.python.org/2/library/itertools.html#itertools.product).

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> product('ABCD', 2)  {'AA', 'AB', 'AC', 'AD', 'BA', 'BB', 'BC', 'BD', 'CA', 'CB', 'CC', 'CD', 'DA', 'DB', 'DC', 'DD'} | |  |
|  |

In **e1e.py**, simulate [itertools.permutations](https://docs.python.org/2/library/itertools.html#itertools.permutations).

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> permutations('ABCD', 2)  {'AB', 'AC', 'AD', 'BA', 'BC', 'BD', 'CA', 'CB', 'CD', 'DA', 'DB', 'DC'} | |  |
|  |

In **e1f.py**, simulate [itertools.combinations](https://docs.python.org/2/library/itertools.html#itertools.combinations).

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> combinations('ABCD', 2)  {'AB', 'AC', 'AD', 'BC', 'BD', 'CD'} | |  |
|  |

In **e1g.py**, simulate [itertools.combinations\_with\_replacement](https://docs.python.org/2/library/itertools.html#itertools.combinations_with_replacement).

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> combinations\_with\_replacement('ABCD', 2)  {'AA', 'AB', 'AC', 'AD', 'BB', 'BC', 'BD', 'CC', 'CD', 'DD'} | |  |
|  |

1. In **e2a.py**, write a decorator the traces whenever a function is invoked and whenever it returns.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @trace  ... **def** **inc**(x):  ... **return** x + 1  >>> inc(1)  enter inc  leave inc  2 | |  |
|  |

In **e2b.py**, amend the decorator to report arguments, return values and errors.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @trace  ... **def** **div**(x, y):  ... **return** x / y  >>> div(4, 2)  enter div(4, 2)  leave div(4, 2): 2.0  2.0  >>> div(x=1, y=0)  enter div(x=1, y=0)  leave div(x=1, y=0) on error: division by zero  ZeroDivisionError: division by zero | |  |
|  |

In **e2c.py**, amend the decorator the show nesting using indentation.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @trace  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> fib(3)  enter fib(3)  enter fib(2)  enter fib(1)  leave fib(1): 1  enter fib(0)  leave fib(0): 0  leave fib(2): 1  enter fib(1)  leave fib(1): 1  leave fib(3): 2  2 | |  |
|  |

In **e2d.py**, amend the decorator to receive the printing function as an argument.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @trace(**print**)  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> fib(3)  ... # prints to stdout  >>> @trace(write)  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> fib(3)  ... # writes to a file | |  |
|  |

In **e2e.py**, amend the decorator to receive the printing function as a keyword-only argument, and default to **print** if no such argument is provided.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @trace  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> fib(3)  ... # prints to stdout  >>> @trace(log=fp.write)  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> fib(3)  ... # writes to a file | |  |
|  |

1. In **e3a.py**, write a decorator that caches function results by arguments.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @cache  ... **def** **fib**(n):  ... **print**(f'computing fib({n})...')  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> fib(3)  computing fib(3)...  computing fib(2)...  computing fib(1)...  computing fib(0)...  2  >>> fib(5)  computing fib(5)...  computing fib(4)...  5 | |  |
|  |

In **e3b.py**, amend the decorator so that the cache is exposed as the cache attribute of the function.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @cache  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> fib(3)  2  >>> fib.cache  {(0,): 0, (1,): 1, (2,): 1, (3,): 2}  >>> fib(5)  5  >>> fib.cache  {(0,): 0, (1,): 1, (2,): 1, (3,): 2}, (4,): 3, (5,): 5} | |  |
|  |

In **e3c.py**, amend the decorator to support a maximum cache size, and only keep the last recently added values.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @cache(3)  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> fib(2)  1  >>> fib.cache  {(0,): 0, (1,): 1, (2,): 1}  >>> fib(5)  5  >>> fib.cache  {(3,): 2, (4,): 3, (5,): 5} | |  |
|  |

In **e3d.py**, amend the decorator to receive the maximum size as a keyword-only argument, and default to an unlimited cache size if no such argument is provided.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @cache # unlimited cache  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2)  >>> @cache(max\_size=3) # limited cache  ... **def** **fib**(n):  ... **return** n **if** n < 2 **else** fib(n-1) + fib(n-2) | |  |
|  |

As a bonus, check out [functools.lru\_cache](https://docs.python.org/3/library/functools.html#functools.lru_cache).

1. In **e4.py**, write a decorator that times function execution and prints it.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @time\_execution  ... **def** **wait**(n):  ... time.sleep(n)  >>> wait(1)  wait took 1.01 seconds to execute  >>> wait(2)  wait took 2.01 seconds to execute | |  |
|  |

1. In **e5a.py**, write a decorator that makes a function exception-safe by suppressing them.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @exception\_safe  >>> **def** **div**(x, y):  ... **return** x / y  >>> div(1, 0)  # No error! | |  |
|  |

In **e5b.py**, amend the decorator to receive any number of exceptions, in which case it only suppresses the specified exceptions, and default to suppressing all exceptions if no arguments are provided.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @exception\_safe(**NameError**, **TypeError**)  ... **def** **f**(error):  ... **raise** error()  >>> f(error=**NameError**) # No error.  >>> f(error=**TypeError**) # No error.  >>> f(error=**ValueError**)  ValueError  >>> @exception\_safe  ... **def** **f**(error):  ... **raise** error()  >>> f(error=**NameError**) # No error.  >>> f(error=**TypeError**) # No error.  >>> f(error=**ValueError**) # No error. | |  |
|  |

1. In **e6.py**, write a decorator that synchronized functions with respect to some lock.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> lock = threading.Lock()  >>> @synchronize(lock)  ... **def** **f**():  ... time.sleep(1)  ... **print**('f')  >>> @synchronize(lock)  ... **def** **g**():  ... time.sleep(1)  ... **print**('g')  >>> threading.Thread(target=f).start()  >>> threading.Thread(target=g).start()  # after one second...  f  # after one second...  g | |  |
|  |

1. In **e7.py**, write a decorator that validates function argument and return value types based on the keyword arguments passed to it.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| >>> @validate\_types(x=**int**, y=**int**, return\_value=**int**)  ... **def** **add**(x, y):  ... **return** x + y  >>> add(1, 2)  3  >>> add('foo', 'bar')  ValueError: argument 'x' must be int | |  |
|  |

# back to our system

First, enter the **q2/** directory, and copy over the latest **server.py**, **client.py** and **web.py** from [exercise 1](https://docs.google.com/document/d/1d5gb3X86MSQccRJRG76BoD7dqhnHYH-ROrYC-3eoBXM/edit).

1. In **cli.py**, implement the **CommandLineInterface** class, which provides the following interface:
   * **command(func)** is a decorator which collects a function and makes it part of the command-line interface, automatically inferring its name and signature.  
     The function name is available as its **\_\_name\_\_** attribute, and its arguments' names are available as the **args** of **inspect.getfullargspec**'s result.
   * **main()** deploys the command-line interface, which parses **sys.argv**, uses the first command-line argument to determine which function to invoke, and passes the rest of the command-line arguments as keyword arguments.  
     The rest of the command-line arguments should have to format **key=value**. If something is wrong (for example, the first argument doesn't match any function name, or the other arguments' format is invalid), it should exit with a non-zero status code and print a usage message.

**cli\_example.py** is a simple example of how I'd use this framework:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| cli = CommandLineInterface()  @cli.command  **def** **inc**(x):  **print**(**int**(x) + 1)  @cli.command  **def** **add**(x, y):  **print**(**int**(x) + **int**(y))  **if** \_\_name\_\_ == '\_\_main\_\_':  cli.main() | |  |
|  |

Then, from the terminal, I'd get:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **python** cli\_example.py  USAGE: python example.py <command> [<key>=<value>]\*  $ **python** cli\_example.py inc x=1  2  $ **python** cli\_example.py add x=1 y=2  3  # And some edge-cases:  $ **python** cli\_example.py foo # Invalid command  USAGE: python example.py <command> [<key>=<value>]\*  $ **python** cli\_example.py add 1 2 # Invalid format  USAGE: python example.py <command> [<key>=<value>]\*  $ **python** cli\_example.py add a=1 b=2 # Invalid arguments  USAGE: python example.py <command> [<key>=<value>]\* | |  |
|  |

Finally, integrate this CLI with **server.py**, so that it has a **run** command with the **address** and **data** arguments, and **client.py**, so that it has an **upload** command with the **address**, **user**, and **thought** arguments.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| $ **python** server.py run address="127.0.0.1:5000" data=data/  …  $ **python** client.py upload address="127.0.0.1:5000" user=1 thought="I'm hungry"  … | |  |
|  |

1. In **website.py**, implement the **Website** class, which provides the following interface:
   * **route(path)** is a second-order decorator that receives a path, and returns a decorator, which then collects a function and makes it part of the HTTP server, handling GET requests sent to this path.  
     So for example, **@route('/')** would collect a function and use it to handle GET requests sent to the path **/**.  
     These functions (let's call them handlers) should return two values: the response status code (e.g. 200 or 404) and body.  
     In case the path is parametrized, like with **@route('/users/([0-9]+)')**, matching the GET request path to its handler should be done using regular expressions, with the matched groups as the handler's arguments.
   * **run(address)** runs the HTTP server at the given address, and uses the collected functions to serve their specified paths.

**website\_example.py** is a simple example of how I'd use this framework:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| website = Website()  @website.route('/')  **def** **index**():  **return** 200, '<html>users list</html>'  @website.route('/users/([0-9]+)')  **def** **user**(user\_id):  **if** user\_id **not in** ['1', '2']:  **return** 404, ''  **return** 200, f'<html>user {user\_id}</html>'  **if** \_\_name\_\_ == '\_\_main\_\_':  website.run(('127.0.0.1', 8000)) | |  |
|  |

Finally, integrate this website with your **web.py**.