

Exercise Manual for Course 1906

Advanced Python: Best Practices and Design Patterns

1906/MA/C.1/911/B.1

by Michael Woinoski

Technical Editor:
Alexander Lapajne

© LEARNING TREE INTERNATIONAL, INC.
All rights reserved.

All trademarked product and company names are the property of their respective trademark holders.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or translated into any language, without the prior written permission of the publisher.

Copying software used in this course is prohibited without the express permission of Learning Tree International, Inc.
Making unauthorized copies of such software violates federal copyright law, which includes both civil and criminal penalties.

Legend for Course Icons.....	ii
Hands-On Exercise 1.1: Python Object-Oriented Programming.....	1
Hands-On Exercise 2.1: Implementing Design Patterns in Python.....	11
Hands-On Exercise 3.1: Unit Testing.....	15
Optional Hands-On Exercise 3.2: Unit Testing With Mocks.....	23
Hands-On Exercise 4.1: Error Detection and Debugging.....	25
Hands-On Exercise 5.1: Measuring and Improving Performance.....	35
Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns.....	41
Hands-On Exercise 7.1: Installation and Distribution.....	51
Hands-On Exercise 8.1: Concurrency.....	61
Hands-On Exercise 8.2: Multiprocessing.....	67
Hands-On Exercise 9.1: Interfacing With REST Web Services and Clients.....	73
Hands-On Exercise A.1: Extending Python.....	77
Hands-On Exercise B.1: Debugging With <code>pdb</code>	83



Legend for Course Icons

Standard icons are used in the hands-on exercises to illustrate various phases of each exercise.



Major step



Warning



Action



Hint



Checkpoint



Stop



Question



Congratulations



Information



Bonus



Solution/Answer



Objectives

In this exercise, you will

- Define Python classes and create objects
- Extend classes to define subclasses

Overview

In this exercise, you will define classes and subclasses, and work with instances of those classes. You'll enhance the Python code of `TicketManor`, the course's theme application. `TicketManor` is a web application for purchasing tickets to concerts, movies, and sporting events.

`TicketManor`'s frontend is implemented in HTML and JavaScript, and the backend is implemented with the Python frameworks Pyramid and SQLAlchemy. But you'll be working with application-defined classes that don't require knowledge of any frameworks other than standard Python modules.



Viewing the `TicketManor` application before your changes

1. ☐ Open a command prompt.
2. ☐ Execute the following commands:

```
cd C:\crs1906\exercises\ex01_inheritance
ticketmanor
```



The `ticketmanor` batch file rebuilds the project and then starts a web server for testing the `TicketManor` application.

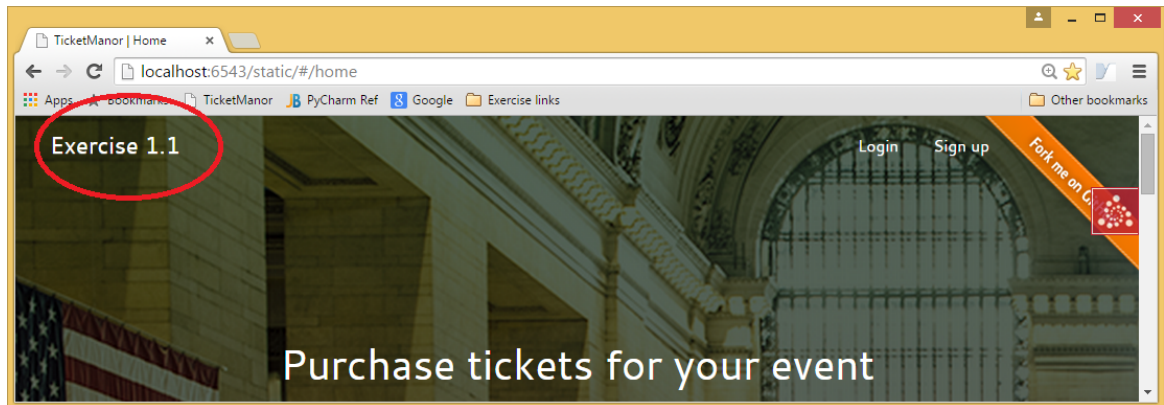


Hands-On Exercise 1.1: Python Object-Oriented Programming (continued)

3. ☐ Wait until the console displays the message "serving on http://0.0.0.0:6543". Then switch to Chrome and click the `TicketManor` toolbar button.



Verify that the page title is **Exercise 1.1**. If not, press `<Ctrl><Shift><R>` to reload the page.



4. ☐ Click the **Concerts** button.
5. ☐ In the search field, enter **Berlin Philharmonic**, then click the **Search** button.



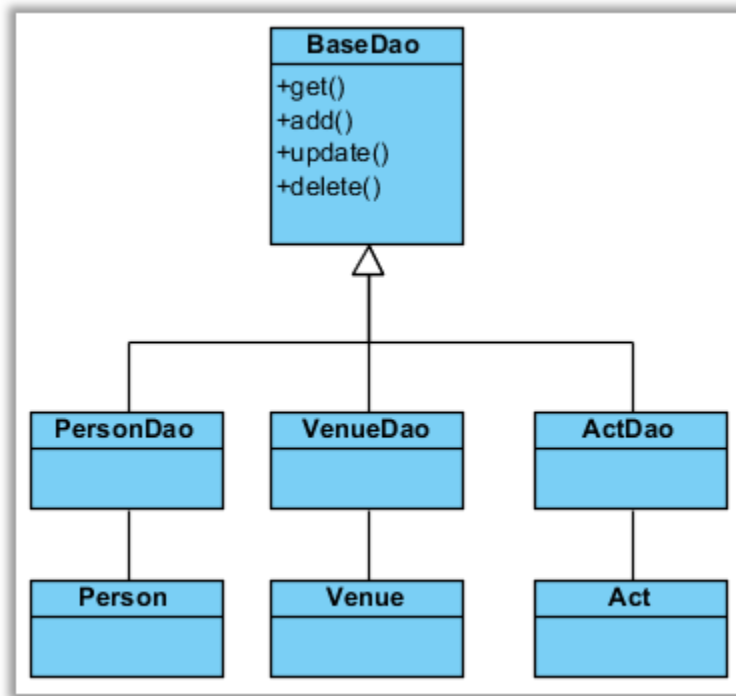
Note that the application does not display any search results. You will implement the classes that query the database for artists that match the entered search terms. Ignore the message "Something went wrong", which is the result of a timeout while downloading the concert news.





Implementing TicketManor's data access objects

Next, you will refactor TicketManor's Data Access Objects (DAOs) to extend a common superclass, `BaseDao`. The following UML diagram shows the relationships between the subclasses of `BaseDao` and the entity classes that they manage.



6. ☐ Switch to PyCharm.
7. ☐ Confirm that the current PyCharm project is `ex01_inheritance`; if not, select **File | Close**, then select **File | Open** and choose `C:\crs1906\exercises\ex01_inheritance`. Click **OK**.
8. ☐ Expand the folder, `ex01_inheritance/ticketmanor/models`, and open the file, `person.py`.



The `Person` class represents a domain object stored in the database. Database operations for `Person` instances are implemented by the `PersonDao` class.



Hands-On Exercise 1.1: Python Object-Oriented Programming (continued)

9. ☐ Expand the folder, `ex01_inheritance/ticketmanor/models/persistence`, and open the file, `person_dao.py`.
10. ☐ Search for comments that contain `TODO`. Perform the `TODO` steps to complete the implementation of the `PersonDao` class.

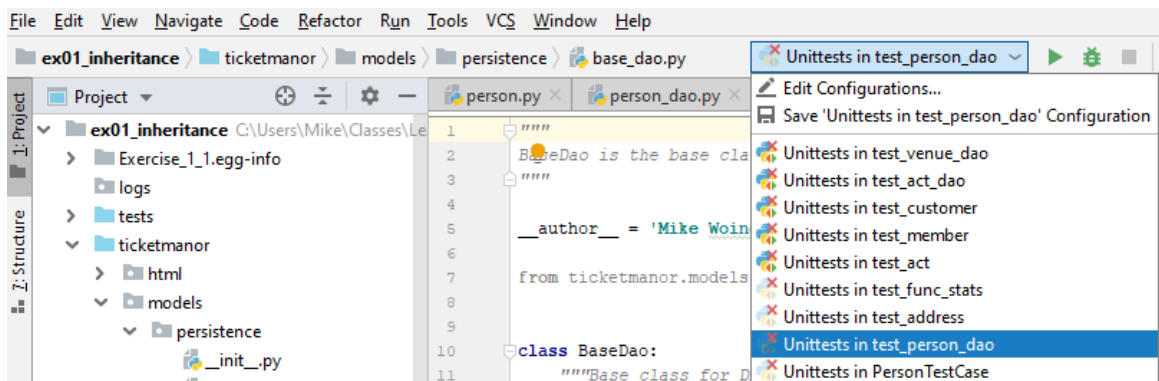


Some of the `TODO` steps require you to copy code from `person_dao.py` to `base_dao.py`.



*To view the hints for an exercise, open a web browser and navigate to the home page, then click the appropriate link under the **Exercise Hints** heading.*

11. ☐ Edit the file `base_dao.py` and complete any remaining `TODO` steps. Save your changes.
12. ☐ In PyCharm's **Run Configurations** drop-down, select **Unittests in test_person_dao**, then click the **Run** button next to the menu.



Verify that the test runner displays a green bar to indicate that all tests succeeded. If the test runner displays a red bar, fix your code and run the unit tests again.

13. ☐ Open the file `venue_dao.py` in the folder `ex01_inheritance/ticketmanor/models/persistence`. Complete the `TODO` steps in `venue_dao.py`.
14. ☐ In PyCharm's **Run Configurations** drop-down, select **Unittests in test_venue_dao**, then click the **Run** button next to the menu.





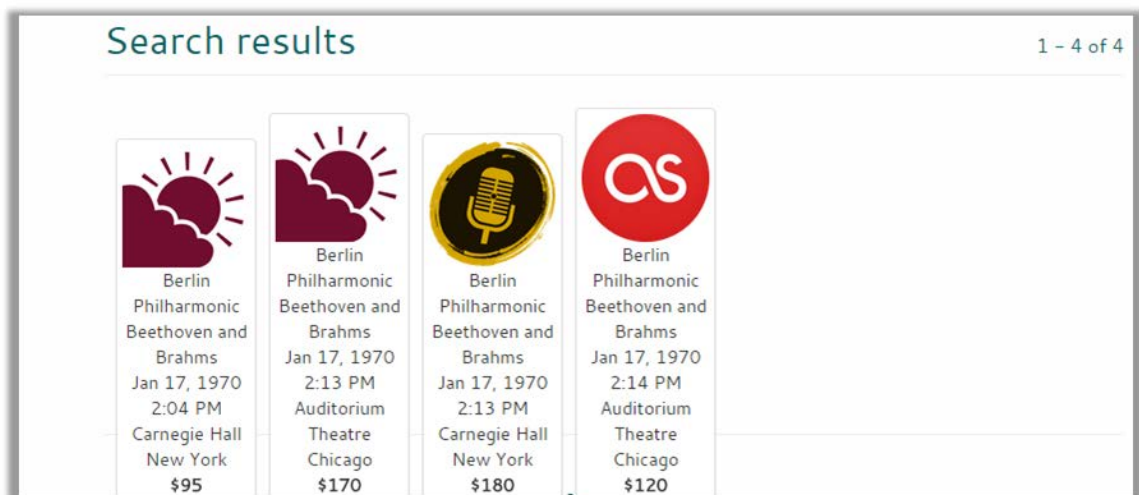
Confirm that the unit tests for `venue_dao.py` succeed.

15. ☐ Open the file `act_dao.py` in the folder `ex01_inheritance/ticketmanor/models/persistence`. Complete the TODO steps in `act_dao.py` and save your work.
16. ☐ In PyCharm's **Run Configurations** drop-down, select **Unittests in test_act_dao**, then click the **Run** button next to the menu. Verify that all unit tests pass.
17. ☐ Switch to the command prompt.
18. ☐ If a server is running, press `<Ctrl><C>` to stop it, then execute the following command to start a new server:

`ticketmanor`
19. ☐ Switch to Chrome and click the `TicketManor` toolbar button.
20. ☐ Click the **Concerts** button, then enter **Berlin Philharmonic** in the Search input and click the **Search** button.



Verify that you get search results. (Your output may be different from the screenshot below.)



Hands-On Exercise 1.1: Python Object-Oriented Programming (continued)



If you make changes to your code, you may need to restart the server to pick up the changes.



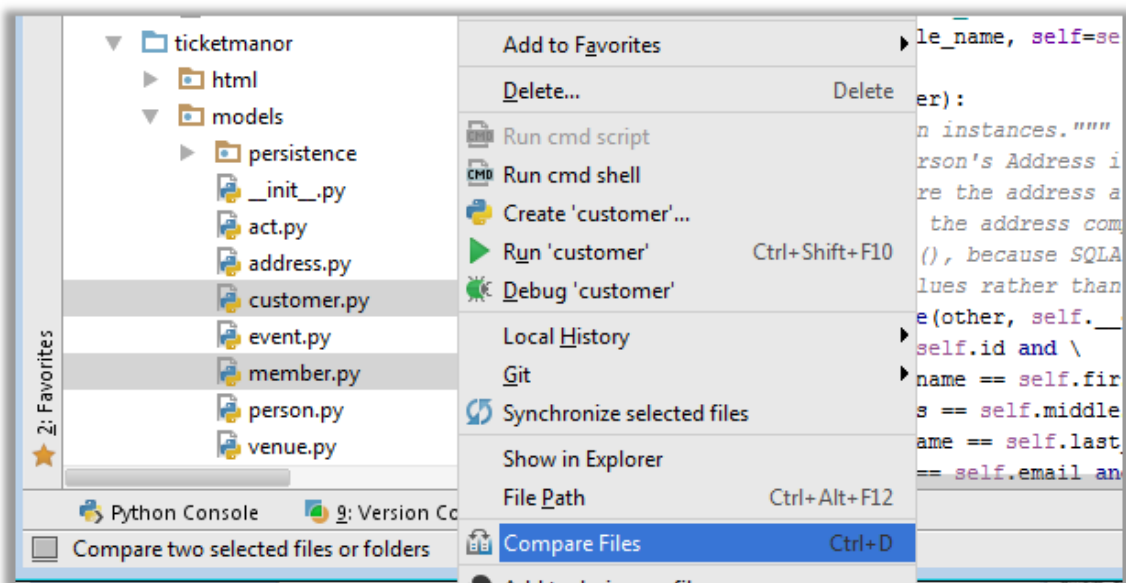
Congratulations! You have enhanced an application's design and eliminated duplicate code by defining a superclass and extending it.



If you have time, try to create and extend a new superclass

In this section, you will eliminate duplicate code by refactoring existing classes to use a common superclass.

21. ☐ In PyCharm, expand the `ticketmanor/models` folder. Use `<Ctrl>-click` to select the files `customer.py` and `member.py`.



22. ☐ Right-click `customer.py` and select **Compare Files**.



Hands-On Exercise 1.1: Python Object-Oriented Programming (continued)



Note that much of the code in the two files is duplicated. You will move the duplicate code into a base class named `User`.

```
person.py (C:/crs1906/exercises/ticketmanor_webapp/ticketmanor/models)
# user_profile = relationship("UserProfile", uselist=False)
# Don't define init(): it breaks SQLAlchemy's magic

@hybrid_property
def name(self):
    # Client code accesses the hybrid property as an attr
    middle_name = self.middles + " " if self.middles is
    return "{self.first_name} {self.last_name}"
    .format(middle_name, self=self)

def _eq_(self, other):
    """Compare Person instances"""
    return isinstance(other, self.__class__) and \
        other.id == self.id and \
        other.first_name == self.first_name and \
        other.middles == self.middles and \
        other.last_name == self.last_name and \
        other.email == self.email and \
        other.street == self.street and \
        other.city == self.city and \
        other.state == self.state and \
        other.country == self.country and \

customer.py (C:/crs1906/exercises/ticketmanor_webapp/ticketmanor/models)
__tablename__ = 'customers'
id = Column('id', Integer, ForeignKey('people.id'), pri

@hybrid_property
def name(self):
    # Client code accesses the hybrid property as an attr
    middle_name = self.middles + " " if self.middles is
    return "{self.first_name} {self.last_name}"
    .format(middle_name, self=self)

def _eq_(self, other):
    """Compare Customer instances"""
    return isinstance(other, self.__class__) and \
        other.id == self.id and \
        other.first_name == self.first_name and \
        other.middles == self.middles and \
        other.last_name == self.last_name and \
        other.email == self.email and \
        other.street == self.street and \
        other.city == self.city and \
        other.state == self.state and \
        other.country == self.country and \
```

- 23. ☐ Close the diff window, then open `customer.py` and complete the TODO steps. Save your work.
- 24. ☐ Open `user.py` and complete the TODO steps. Save your work.
- 25. ☐ In PyCharm's **Run Configurations** drop-down, select **Unittests in test_customer**, then click the **Run** button next to the menu.



Verify that all unit tests still pass.

- 26. ☐ Open `member.py` and complete the TODO steps. Save your work.
- 27. ☐ In PyCharm's **Run Configurations** drop-down, select **Unittests in test_member**, then run the tests.



Verify that all unit tests still pass.



Congratulations! You have used inheritance to eliminate duplicate code.



Hands-On Exercise 1.1: Python Object-Oriented Programming (continued)



If you have more time, try to gain experience with dictionary comprehensions

In this section, you will replace list and dictionary processing in for loops with comprehensions.

- 28. ☐ Open the file, `ticketmanor/scripts/func_stats.py`, and complete the steps marked BONUS TODO.
- 29. ☐ In PyCharm's **Run Configurations** drop-down, select **Unittests in test_func_stats**, then click **Run**.



Verify that all unit tests still pass.

- 30. ☐ Open the file, `ticketmanor/models/act.py`, and complete the steps marked BONUS TODO.
- 31. ☐ In PyCharm's **Run Configurations** drop-down, select **Unittests in test_act**, then run the tests.



Verify that all unit tests still pass.

- 32. ☐ Open the file, `ticketmanor/models/address.py`, and complete the steps marked BONUS TODO.
- 33. ☐ In PyCharm's **Run Configurations** drop-down, select **Unittests in test_address**, then run the tests.



Verify that all unit tests still pass.





Congratulations! You have replaced looping with list and dictionary comprehensions.



This is the end of the exercise.





Objectives

In this exercise, you will

- Define a skeleton algorithm based on the Template Method design pattern
- Write an abstract base class and extend it with concrete subclasses
- Replace list iteration with a generator function

Overview

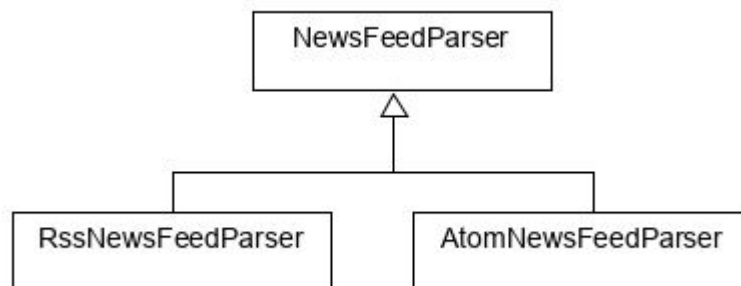
In this exercise, you will implement the Template design pattern in the news feed reader class hierarchy that we discussed in lecture. In addition, you will write and use an abstract base class. Finally, you will gain experience with a common Python idiom by writing a generator function.



Implementing the Template Method design pattern

The news display on the `TicketManor` search page is generated by REST web services. News might be downloaded in one of two formats, RSS or Atom. Both formats are based on XML and both have similar content, but there are minor differences in the XML. For example, for the date of a news item, RSS uses `<pubDate>` while Atom uses `<updated>`.

There are currently two classes, `RssNewsFeedParser` and `AtomNewsFeedParser`, that handle the different news feed formats. You will extract a common superclass that implements the Template Method design pattern to eliminate duplication between the news feed parser classes.



1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex02_template_method`, then click **OK**.



Hands-On Exercise 2.1: Implementing Design Patterns in Python (continued)

2. ☐ Open the file `ticketmanor/rest_services/feed_reader/rss_news_feed_parser.py` and complete the TODO steps in the file.



The `RssNewsFeedParser` class parses a news feed in the RSS XML format. Ignore the steps marked BONUS TODO for now. You'll complete those steps in the bonus section.

3. ☐ Complete the remaining TODO steps in `news_feed_parser.py`. (Ignore the BONUS TODO steps for now.)



Be sure you replace all occurrences of ". . ." with a line of Python code.

4. ☐ In the **Run** menu, select **Unittests in test_rss_news_feed_parser** and click the **Run** button.



Verify that all unit tests pass.

5. ☐ Open the file, `ticketmanor/rest_services/atom_news_feed_parser.py`, and complete the TODO steps in the file. (Ignore the BONUS TODO steps for now.)



The `AtomNewsFeedParser` class parses a news feed in the AtomPub XML format.

6. ☐ In the **Run** menu, select **Unittests in test_atom_news_feed_parser** and click the **Run** button.



Verify that all unit tests pass.

7. ☐ Switch to the command prompt. If a server is running, press <Ctrl><C> to stop it.

8. ☐ Execute the following commands to start a new server:

```
cd \crs1906\exercises\ex02_template_method
ticketmanor
```

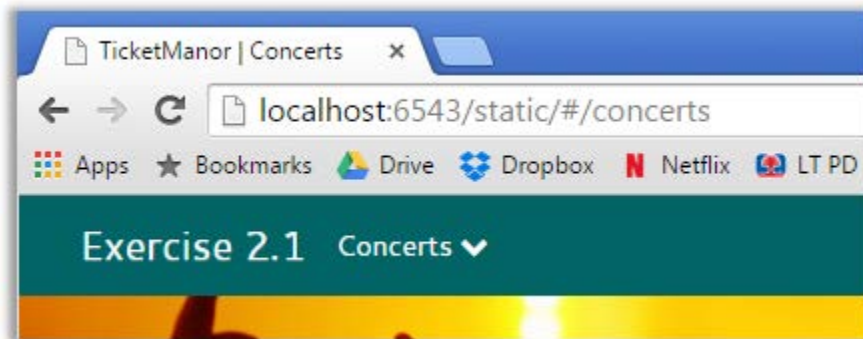


If a Windows Security Alert pops up, click Allow Access.



Hands-On Exercise 2.1: Implementing Design Patterns in Python (continued)

9. ☐ Switch to Chrome and click the **TicketManor** button in the toolbar.
10. ☐ Verify that the page title is **Exercise 2.1**. If not, press `<Ctrl><Shift><R>` to reload the page.



11. ☐ Click the **Concerts** button and verify that the page displays news items.



Congratulations! You have implemented the Template Method design pattern.



If you have time, try to define an abstract base class

In this section, you will convert the `NewsFeedParser` to an abstract base class.

12. ☐ Edit `news_feed_parser.py` and complete the steps marked BONUS TODO. (Ignore the steps marked BONUS TODO 2 for now.)



Use `<Ctrl><Shift><N>` to quickly navigate to a file.

13. ☐ In the **Run** menu, select **Run news_feed_parser** and click the **Run** button.



Verify that the console displays a `Success` message.



Hands-On Exercise 2.1: Implementing Design Patterns in Python (continued)

- 14. ☐ Run the `test_rss_news_feed_parser.py` and `test_atom_news_feed_parser.py` unit tests again, and verify that all tests pass.
- 15. ☐ Switch to Chrome and reload the Concerts search page. Verify that the news display still works.



Congratulations! You have defined an abstract base class.



If you have more time, try to define a generator function

The `NewsFeedParser` method `parse_xml_content()` always returns a list of all news items. This is inefficient if the list is potentially very large. In this section, you will convert the `parse_xml_content()` method into a generator function that yields a single news item each time it is called.

- 16. ☐ Edit `news_feed_parser.py` and complete the steps marked BONUS TODO 2 in the `parse_xml_content()` method.
- 17. ☐ Run the unit tests in `test_rss_news_feed_parser.py` and verify that all tests pass.
- 18. ☐ Switch to the `TicketManor` page in Chrome and verify that concert new items still load as they did before.



Congratulations! You have defined an efficient generator function.



This is the end of the exercise.



Objectives

In this exercise, you will

- Write unit tests for Python classes
- Run unit tests with `unittest`, `Nose`, and `Pytest`
- Generate unit test code coverage reports

Overview

In the previous exercise, you ran unit tests for `RssNewsFeedParser` and `AtomNewsFeedParser`. In this exercise, you will write the unit tests themselves. You will write and run unit tests using techniques and frameworks discussed in lecture, including the `unittest` module and the `Nose` and `Pytest` frameworks.



Writing and running unit tests

In this section, you'll write unit tests for the `NewsFeedParser` class. First, you'll examine `NewsFeedParser`, then you'll complete its unit tests.

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex03_unit_testing`, then click **OK**.
2. ☐ Open the file, `ticketmanor/rest_services/feed_reader/news_feed_parser.py`, and complete the **TODO** steps in the file.
3. ☐ Open the file, `tests/rest_services/test_rss_news_feed_parser.py`, and complete the **TODO** steps in the file.
4. ☐ Scroll to the first test case in `test_rss_news_feed_parser.py` (the method `test_get_news_music()`). Right-click within the body of the method. Note that the **Run** item on the pop-up menu is `Run 'Unittests for test_r...'`. When you right-click within the body of a test method, PyCharm will run only that test method.

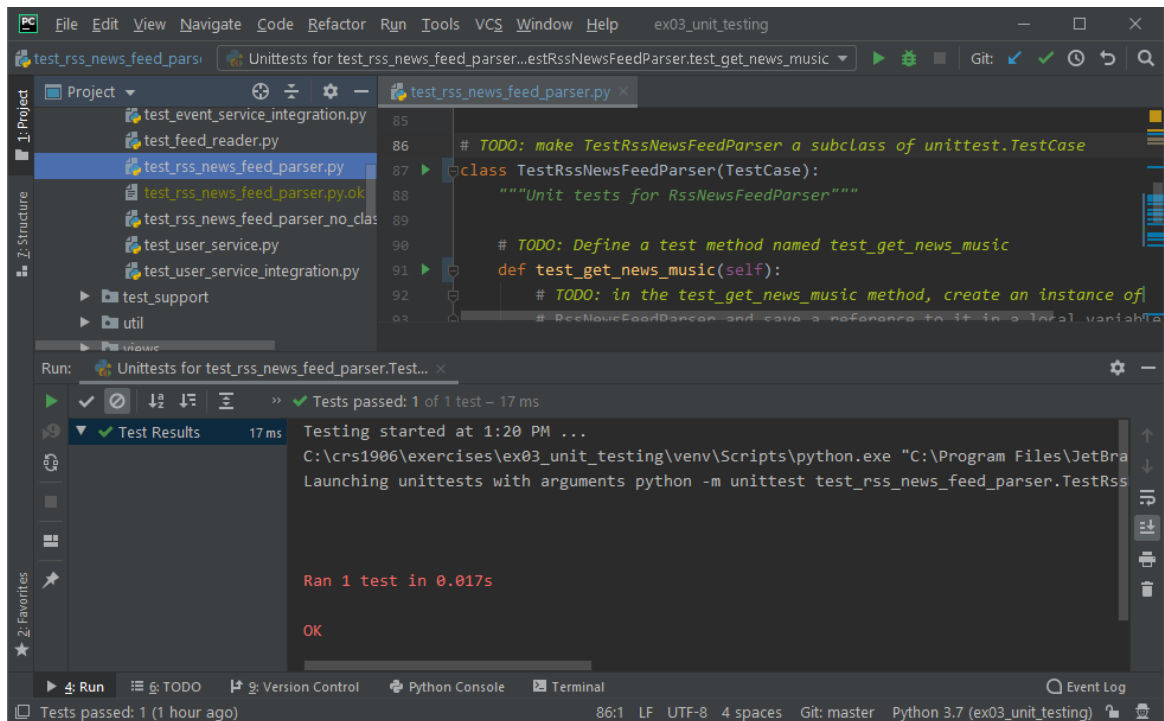


Hands-On Exercise 3.1: Unit Testing (continued)

5. ☐ Select Run 'Unittests for test_r...'.



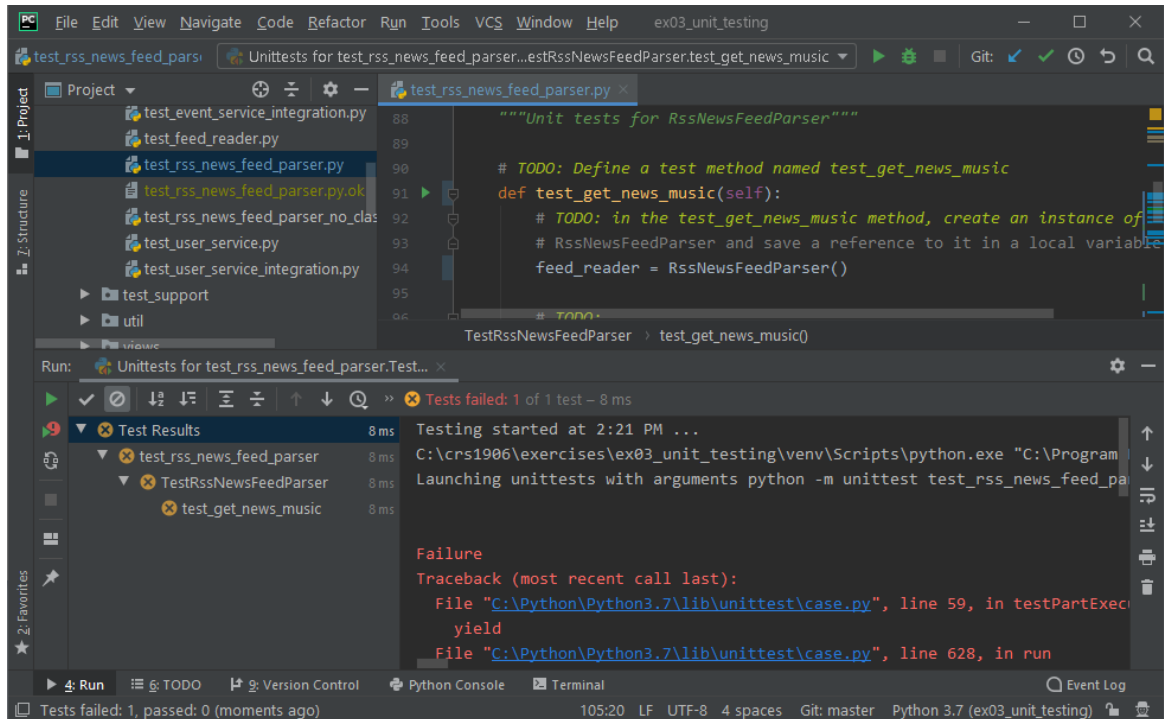
Verify the test case passes by checking for a green checkmark in the PyCharm test runner:



Hands-On Exercise 3.1: Unit Testing (continued)



If the test runner displays a red checkmark, the test case failed. Fix the problem and re-run the test case.



6. ☐ Scroll to the top of `test_rss_news_feed_parser.py` and right-click above the class definition. When you right-click outside an individual test case, PyCharm will run all test cases in the module.
7. ☐ Select Run 'Unittests in test_rs...'



Verify that all test cases pass.

8. ☐ Switch to the command prompt. If a server is running, press `<Ctrl><C>`. Then, execute the following commands:

```
deactivate
cd \crs1906\exercises\ex03_unit_testing
```



Hands-On Exercise 3.1: Unit Testing (continued)

9. ☐ Execute the following command to use the `unittest` module to run your unit tests (enter the command all on one line):

```
python -m unittest tests\rest_services  
\test_rss_news_feed_parser.py
```



Verify that all unit tests pass.

10. ☐ Now execute the following command to have Pytest run your unit tests (enter the command all on one line):

```
pytest -v tests\rest_services\test_rss_news_feed_parser.py
```



Examine the output of the `pytest` command.



Congratulations! You have written and executed automated unit tests using Python's standard `unittest` module and Pytest.



If you have time, try writing test cases without a `TestCase` subclass

In this section, you will define test cases as plain functions that use the `assert` statement.

11. ☐ Open the file, `tests/rest_services/test_rss_news_feed_parser_no_class.py`, and complete the TODO steps in the file.
12. ☐ Use Pytest to run your unit tests (enter the command all on one line):

```
pytest -v tests\rest_services  
\test_rss_news_feed_parser_no_class.py
```



Pytest can run test cases defined as plain functions with `assert` statements.



13. ☐ Switch to PyCharm and compare `test_rss_news_feed_parser_no_class.py` with `test_rss_news_feed_parser.py`.



Right-click the editor tab for one of the files and select Split Vertically so you can view the files side by side.



Note that the "no class" version is simpler and easier to read.



Congratulations! You have simplified your unit tests using features of Pytest.



Leveraging test discovery

The `unittest` module and Pytest give similar results when executing a single test. However, Pytest supports test discovery, which makes it much easier to run multiple tests.

14. ☐ Use Pytest to execute all unit tests under the current directory by running the following commands:

```
set PYTHONPATH=tests
pytest tests
```



Note that Pytest discovers and runs all tests under the `tests` directory, not just a single test file as in the previous steps.



Some tests will fail. We injected errors into the code being tested so you can examine the error reporting of Pytest. The final lines of output should be similar to the following. (Note that your results may be slightly different.)

```
===== 2 failed, 94 passed, 28 warnings in 15.14s
=====
```

15. ☐ Pytest's output includes warnings from third-party libraries. Because you don't own the code that causes the warnings, you can tell Pytest to ignore them by adding the option, `--disable-warnings`, to the `pytest` command:



Hands-On Exercise 3.1:

Unit Testing

(continued)

```
pytest --disable-warnings tests
```



It's not a good practice to ignore warnings. However, it's a useful option when you need to simplify a very verbose report.

16. ☐ To simplify Pytest's report even more, add the `-q` option:

```
pytest -q --disable-warnings tests
```



Congratulations! You have executed unit tests using test discovery.



Generating test coverage reports

Pytest can generate reports that describe how much of the code in the modules being tested is actually run during your tests.

17. ☐ Run the following command to generate a test coverage report (enter the command all on one line):

```
pytest --cov-report term --cov ticketmanor.rest_services tests  
\rest_services\test_rss_news_feed_parser.py
```



What percentage of the code in `rss_news_feed_parser.py` was executed?

18. ☐ Run the following command to add line numbers of untested code to the report (enter the command all on one line):

```
pytest --cov-report term-missing --cov  
ticketmanor.rest_services tests\rest_services  
\test_rss_news_feed_parser.py
```



Which lines of `rss_news_feed_parser.py` were not executed?



An HTML coverage report can give a "big picture" overview of test case coverage.



19. ☐ Run the following commands to generate an HTML test coverage report:

```
pytest -q --cov-report html --cov ticketmanor tests
```

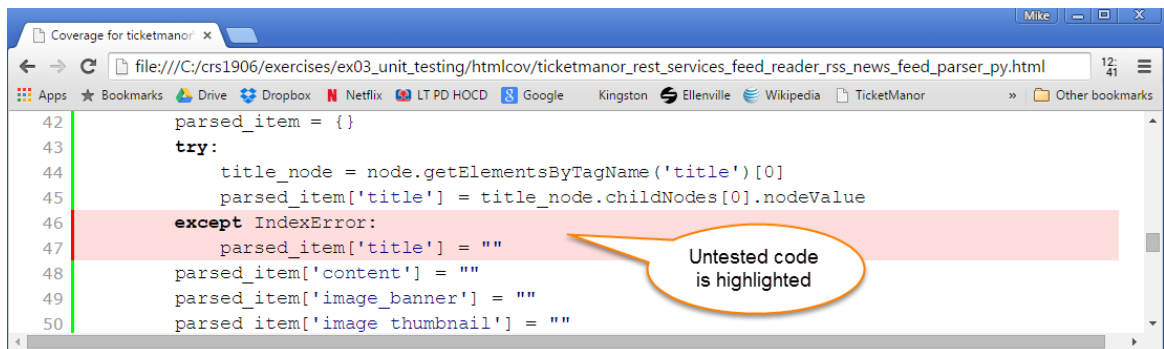
20. ☐ Open File Explorer and navigate to `C:\crs1906\exercises\ex03_unit_testing\htmlcov`.

21. ☐ Double-click the file, `index.html`.

22. ☐ Click the link for `ticketmanor\rest_services\feed_reader\act.py`.



Untested lines of code are highlighted. This makes it easy to see where you need to add test cases.



If you have more time, try achieving 100% test coverage



What percentage of the code in `observer.py` was executed?

23. ☐ Modify `test_observer.py` to achieve 100% test coverage of `observer.py`.



Hands-On Exercise 3.1: Unit Testing (continued)



Congratulations! You have generated test coverage reports to identify code that is not being tested sufficiently.



This is the end of the exercise.



Objectives

In this exercise, you will

- Use mock objects in unit tests to satisfy dependencies
- Test exception handling by programming mock objects to throw exceptions

Overview

In this exercise, you will implement strategies for unit testing using mock objects.

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex03_with_mocks`, then click **OK**.

2. ☐ Open the file, `ticketmanor/rest_services/feed_reader/feed_reader.py`, and complete the TODO steps in the file.



For these TODO steps, you will just read the existing code. You won't make any changes in `feed_reader.py`.

If the project directory contains a file named `ticketmanor_db.sqlite`, delete that file.

3. ☐ Open the file, `tests/rest_services/test_feed_reader.py`, and complete the TODO steps in the file.
4. ☐ In the **Run** menu, select **Unittests in test_feed_reader** and click the **Run** button.



The call to `fetch_news_items()` will display an exception traceback; but as long as you get a green checkmark, the test case passed.



Verify that all test cases pass.

5. ☐ Inject an error in the `fetch_news_items()` method in `feed_reader.py` by deleting the `max_items` argument from the call to the news feed parser's `get_news()` method.
6. ☐ Run the unit tests again and verify that some test cases fail.



Congratulations! You have used mock objects to satisfy dependencies in unit tests.



Optional Hands-On Exercise 3.2: Unit Testing With Mocks (continued)



If you have time, try testing exception handling

In this section, you will test the application's exception handling. You will program mock objects to raise exceptions and verify that the application code handles the exceptions correctly.

7. ☐ Open the file, `ticketmanor/rest_services/user_service.py`, and complete the TODO steps in the file. For these TODO steps, you will just read the existing code. You won't make any changes in `user_service.py`.
8. ☐ Open the file, `tests/rest_services/test_user_service.py`, and complete the TODO steps in the file.
9. ☐ In the **Run** menu, select **Unittests in test_user_service** and click the **Run** button.



Verify that all test cases pass.



Congratulations! You have used mock objects to verify that application code handles exceptions correctly.



This is the end of the exercise.



Objectives

In this exercise, you will

- Log messages from Python modules
- Verify code with Pylint
- Trace program execution with the PyCharm IDE

Overview

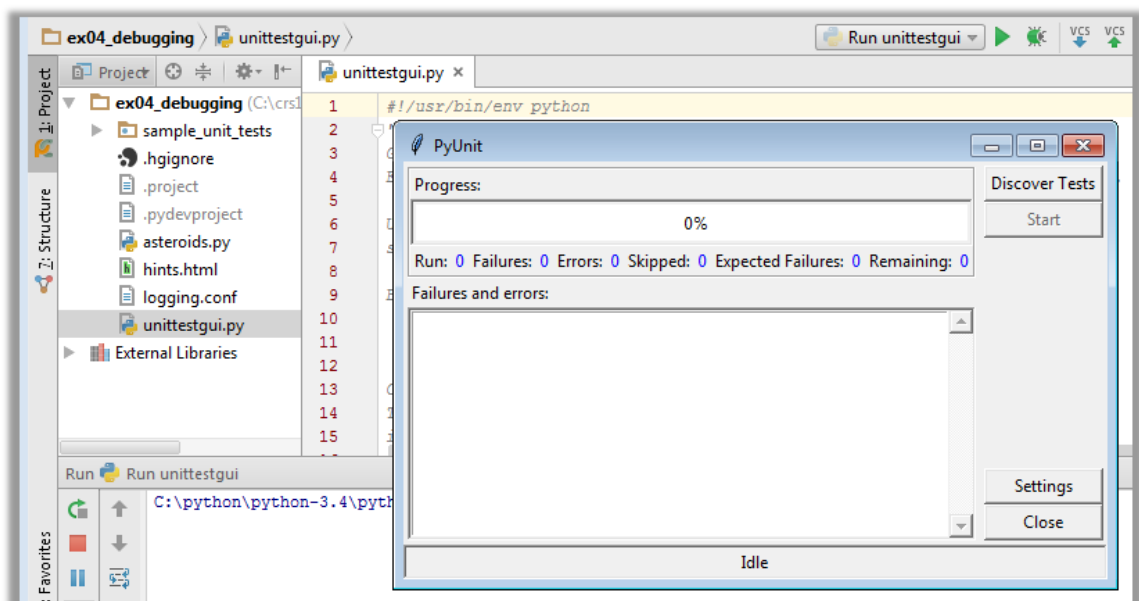
In this exercise, you will apply several of the error detection and debugging techniques.



Running unittestgui.py

You will add logging to `unittestgui.py`, a tkinter script that implements a simple GUI unit test runner. First, you will run the script to get familiar with its operation. Then you will add logging to the script.

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex04_debugging`; then, click **OK**.
2. ☐ In the Project window, right-click `unittestgui.py` and select **Run unittestgui**.



3. ☐ In the new application window, click **Discover Tests**.



Hands-On Exercise 4.1: Error Detection and Debugging (continued)

4. ☐ Select `C:\crs1906\exercises\ex04_debugging\sample_unit_tests` and click **OK**.



Note that the `sample_unit_tests` directory is preselected in the file chooser dialog. When you run the application again, simply click OK when the file dialog appears.

5. ☐ Click **Start**.
6. ☐ When the Progress indicator reaches 100%, double-click one of the error messages.
7. ☐ Click **Close** to dismiss the dialog.
8. ☐ Click **Close** to exit the script.

Now that you are familiar with the script's operation, you will add logging to it.



Adding logging to `unittestgui.py`

9. ☐ Edit `unittestgui.py` and complete the TODO steps.
10. ☐ Edit `logging.conf` and complete the TODO steps.
11. ☐ Execute `unittestgui.py` and run unit tests as you did in the previous steps.



Remember to close the `unittestgui` application after running the unit tests.

12. ☐ Open `unittestgui.log`.



Verify that you see the output of the logging method calls you added.

13. ☐ Edit `logging.conf` and change the logging level of the `unittestgui` logger from `DEBUG` to `WARNING`.
14. ☐ Execute `unittestgui.py` again, then open `unittestgui.log`.





How is the logging output different this time?

- 15. ☐ Edit `logging.conf` and change the logging level of the `unittestgui` logger to `DEBUG`.
- 16. ☐ Modify the `file_formatter` configuration to add a timestamp to log messages.



Refer to the course notes for Logging Configuration File `logging.conf`.

- 17. ☐ Execute `unittestgui.py` again, then open `logs/unittestgui.log`.



Verify that the new logging messages contain the timestamp.



Congratulations! You have adding logging to a Python application.



Hands-On Exercise 4.1: Error Detection and Debugging (continued)



If you have time, try checking code with Pylint

In this section, you will analyze code using Pylint.



Running Pylint

18. ☐ Open a command prompt and run the `unittestgui.py` script:

```
cd \crs1906\exercises\ex04_debugging
python unittestgui.py
```



Note that the Python interpreter doesn't display any warnings or errors when the script runs.

19. ☐ Run `pylint` on `unittestgui.py`:

```
pylint unittestgui.py
```

20. ☐ Scroll through the generated output.



Pylint flags code with poor style (for example, methods without doc strings; identifiers that don't follow Python conventions) as well as potential bugs (for example, unused variables and arguments, which often indicate typos).





Disabling Pylint messages

For now, let's ignore the messages about TODO comments, invalid names, and unnecessary pass statement; for example:

```
unittestgui.py:44:2: W0511: TODO: import the logging package
(fixme)
...
unittestgui.py:50:0: C0103: Constant name "logger" doesn't
conform... (invalid-name)
...
unittestgui.py:90:8: W0107: Unnecessary pass statement
(unnecessary-pass)
```

21. ☐ Note the identifiers in parentheses at the end of the above message (fixme, invalid-name, and unnecessary-pass).
22. ☐ Pass the message identifiers as arguments to pylint's `--disable` option:

```
pylint --disable=fixme,invalid-name,unnecessary-pass
unittestgui.py
```



Are the "fixme", "invalid name", and "unnecessary pass" messages gone?

☐ Yes ☐ No



Creating a Pylint configuration file

The list of `pylint` options can get very long, so it's convenient to save the configuration in a `pylint` configuration file. You can run Pylint with the `--generate-rcfile` option to generate a template configuration file. Any options that precede the `--generate-rcfile` option will be added to the generated file.

23. ☐ Run the following command to generate a `pylintrc` file:

```
pylint --disable=fixme,invalid-name,unnecessary-pass --
generate-rcfile > pylintrc
```



Hands-On Exercise 4.1: Error Detection and Debugging (continued)

24. ☐ Run `pylint` again with no options:

```
pylint unittestgui.py
```



Verify that the `pylint` output does not include "fixme", "invalid name", or "unnecessary pass" messages.



`Pylint` read the `pylintrc` file and applied its configuration settings.



Congratulations! You have used Pylint to analyze code for poor style and potential bugs.



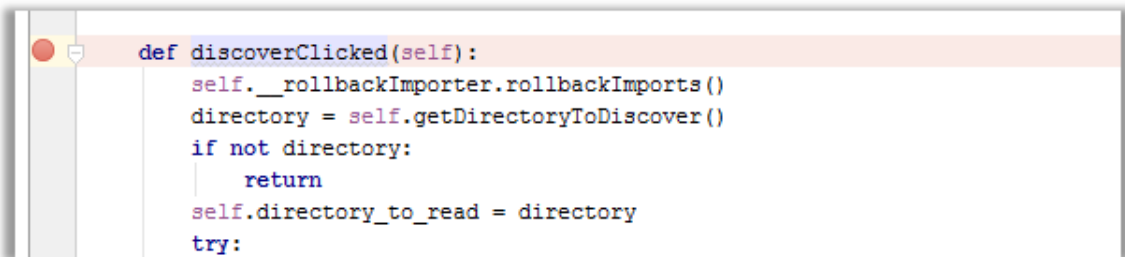
Debugging with PyCharm

In this section, you will use PyCharm's debugger to step through code.

25. ☐ In PyCharm, open `unittestgui.py`.
26. ☐ Set a breakpoint on the `discoverClicked()` method at or near line 118.



Click the line in the editor, then click in the margin just to the right of the line number itself.

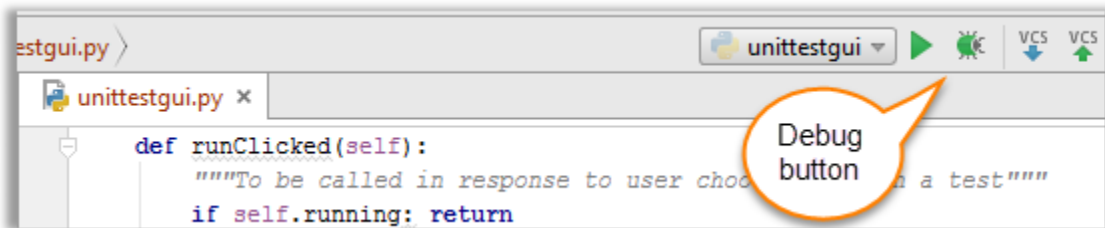


```
def discoverClicked(self):  
    self.__rollbackImporter.rollbackImports()  
    directory = self.getDirectoryToDiscover()  
    if not directory:  
        return  
    self.directory_to_read = directory  
    try:
```



Hands-On Exercise 4.1: Error Detection and Debugging (continued)

27. ☐ In PyCharm's Run menu, select the **Run unittestgui** menu item, then click the **Debug** button

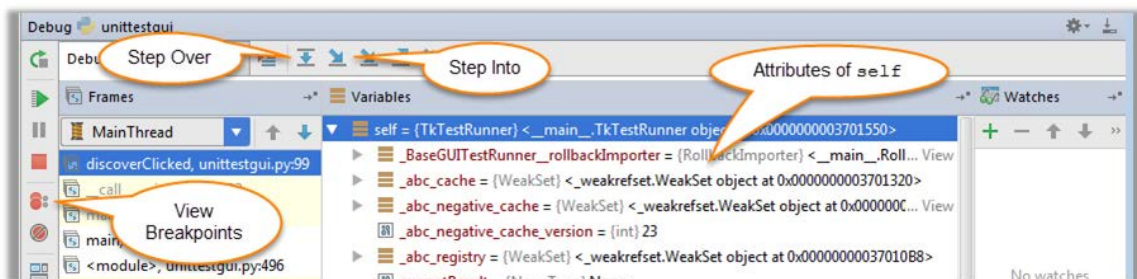


28. ☐ When the unit test GUI pops up, click **Discover Tests**.



Note that when the breakpoint is reached, the debugger window pops up in PyCharm.

29. ☐ Click **Step Over** once to begin single-stepping through the `discoverClicked()` method.
30. ☐ In the Variables pane of the debugger window, expand the variable `self` and examine its attributes.

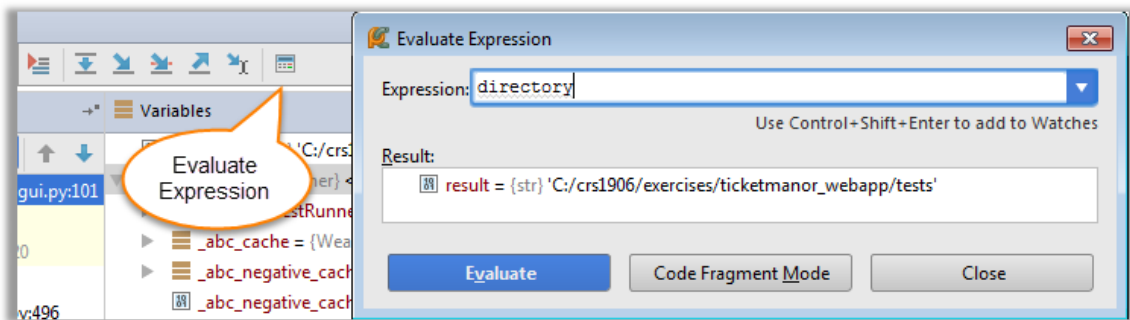


31. ☐ In the Variables window, collapse the `self` attributes.
32. ☐ On the debugger toolbar, click **Step Over** to execute the call to `rollbackImports()`.
33. ☐ Click **Step Into** to step into the call to `getDirectoryToDiscover()`.
34. ☐ Click **Step Over** to execute the call to `askdirectory()`.
35. ☐ Switch to the `unittestgui` application. In the **Browse for Folder** dialog, select `C:\crs1906\exercises\ex04_debugging\sample_unit_tests`.



Hands-On Exercise 4.1: Error Detection and Debugging (continued)

- 36. ☐ Switch to PyCharm and click **Step Over**.
- 37. ☐ In the Debug window toolbar, click **Evaluate Expression**.



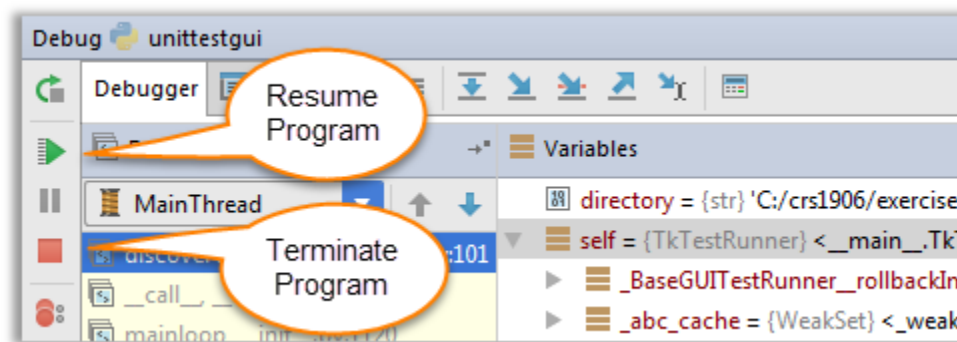
- 38. ☐ Enter `directory` and click **Evaluate**.
- 39. ☐ Experiment with the debugger. Keep stepping over and stepping into method calls. Examine the state of new and changed variables in the Variable window.



*Note that the Variables window highlights changed variables in blue. If you step into code from a library (for example, `Lib/tkinter`), click **Step Out** until you return to the `unittestgui.py` script.*



*If the unit test GUI program appears to hang, click the debugger's **Resume Program** button.*



- 40. ☐ When you are finished stepping through the script, click the GUI's **Close** button.





Congratulations! You have used the PyCharm debugger to single step through a program and examine its execution state.



If you have more time, try debugging a Python script

The script `asteroids.py` is a simple Python version of the classic Asteroids video game. There's a bug in the code that causes the program to crash.

- 41. ☐ Use the PyCharm debugger to diagnose and fix the problem.
- 42. ☐ Now play Asteroids!
 - Fire engine: `<up arrow>`
 - Turn ship: `<left arrow>`, `<right arrow>`
 - Shoot photon torpedo: `x`
 - Quit: `q`



Congratulations! You have debugged a program with PyCharm.



This is the end of the exercise.





Objectives

In this exercise, you will

- Use profilers to monitor and measure execution of Python applications
- Replace the standard CPython implementation with PyPy

Overview

In this exercise, you will profile the execution of a program that solves sudoku puzzles. Then you will compare the program's performance with the standard CPython interpreter implementation and the PyPy implementation.



Profiling with the `cProfile` module

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex05_performance`, then click **OK**.

2. ☐ Switch to a command prompt. If a virtual environment is active, deactivate it:

`deactivate`

3. ☐ Run the `sudoku.py` program and examine its output:

```
cd \crs1906\exercises\ex05_performance
python sudoku.py sudoku_input.txt
```



Now you will use the `cProfile` module to measure the performance of `sudoku.py`.

4. ☐ Execute the sudoku solver with `cProfile`:

```
python -m cProfile -s tottime sudoku.py sudoku_input.txt >
profile1.txt
```

5. ☐ Open `profile1.txt`.



It may take a few seconds before PyCharm displays `profile1.txt` in the project view.



Hands-On Exercise 5.1: Measuring and Improving Performance (continued)



Which function had the largest total execution time (not including subfunction calls)?

6. ☐ Execute the sudoku solver with `cProfile` again, this time sorting by number of calls:

```
python -m cProfile -s ncalls sudoku.py sudoku_input.txt >
profile2.txt
```

7. ☐ Open `profile2.txt`.



Which function had the highest total number of calls?



Congratulations! You have profiled a program's execution using the `cProfile` module.



If you have time, try to run Python scripts with PyPy

You often get big performance improvements by executing your scripts with PyPy instead of the standard Python interpreter (CPython). For this exercise, you'll compare the performance of the Sudoku solver using PyPy and CPython.

8. ☐ Execute the sudoku solver with PyPy:

```
use_pypy
pypy3 sudoku.py sudoku_input.txt | head -3
```



Based on the script's output, how long did it take to solve 32 puzzles?

9. ☐ Run the script with the standard Python interpreter.

```
use_python37
python sudoku.py sudoku_input.txt | head -3
```





Based on the script's output, how long did it take to solve 32 puzzles?



Using PyPy can result in dramatic performance improvements.



Congratulations! You have used PyPy to boost the performance of your Python scripts.



If you have more time, try profiling a program with SnakeViz

In this section, you'll profile the sudoku solver with SnakeViz.

10. ☐ Execute the sudoku solver with `cProfile`, saving its output in `sudoku.prof`:

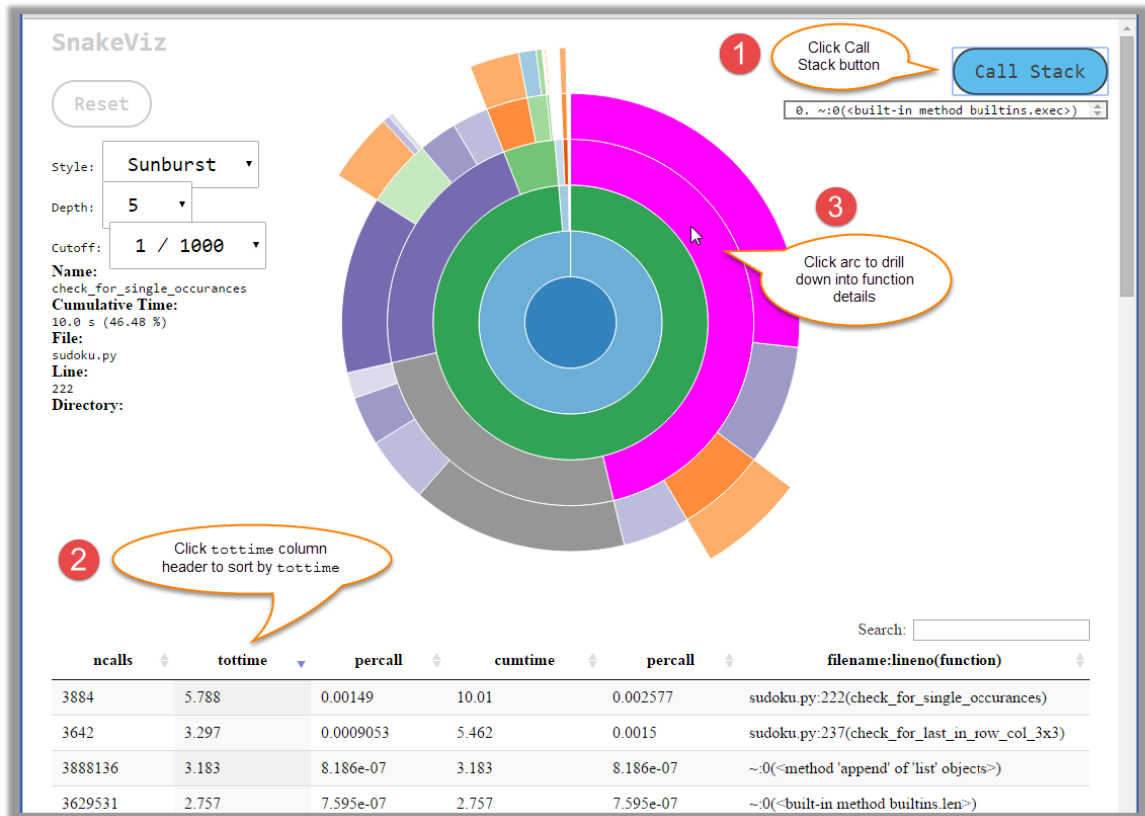
```
python -m cProfile -o sudoku.prof sudoku.py sudoku_input.txt
```



Hands-On Exercise 5.1: Measuring and Improving Performance (continued)

11. ☐ Start SnakeViz. In the Style dropdown, select **Sunburst**.

```
snakeviz sudoku.prof
```



12. ☐ In the SnakeViz browser window, click the **Call Stack** button.
13. ☐ In the function detail under the starburst, click the `tottime` column header twice to sort by largest total time.
14. ☐ In the starburst, click the third arc from the center to drill down to the details of the `check_for_single_occurrences()` function.
15. ☐ Hover over each of the arcs in the starburst to see the execution details of each called function.
16. ☐ In the Call Stack display, click the last item (numbered 0) to return to the top-level starburst.
17. ☐ Experiment with SnakeViz by clicking various arcs in the starburst.



18. ☐ When you are finished experimenting, switch to the command prompt and press `<Ctrl><C>` to shut down the SnakeViz server.



Congratulations! You have analyzed a program's execution profile with the SnakeViz GUI.



If you have more time, try comparing the performance of different coding techniques

Which are faster: loops or list comprehensions? Use one of the profiling techniques to investigate.

19. ☐ Edit `func_stats.py` and examine the code under the BONUS TODO comment.



Note that the code creates a list with an explicit loop.

20. ☐ Profile the execution of `func_stats.py` using `timeit` or `cProfile`.



What was the total time spent in the function `get_function_stats()`?

21. ☐ Edit `func_stats.py`, comment out the code under the BONUS TODO comment, and uncomment the code under the second BONUS TODO comment.

22. ☐ Use the same profiler to profile `func_stats.py` again.



What was the total time spent in the function `get_function_stats()`?

23. ☐ In this case, which is faster: explicit looping or a list comprehension?



Hands-On Exercise 5.1: Measuring and Improving Performance (continued)



Congratulations! You have used a profiler to compare run times of different coding techniques.



This is the end of the exercise.



Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns

Objectives

In this exercise, you will

- Implement the Decorator design pattern for profiling the execution of selected methods
- Create loosely coupled many-to-many relationships with the Observer design pattern
- Access remote objects using the Proxy design pattern

Overview

In this exercise, you will implement the Decorator, Observer, and Proxy design patterns in Python.



Applying a profiling decorator

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex06_design_patterns`, then click **OK**.

2. ☐ Edit `measure.py` and examine the `measure()` function.



This is a profiling decorator, similar to the decorator discussed in lecture. However, instead of reporting on each method call, this decorator accumulates the total number of calls and total execution time of each function that it is applied to.

3. ☐ Examine the code of the `measure()` function. Be sure you understand how the `measure()` function works.

4. ☐ Examine the code of the `get_function_stats()` function. This function returns a list of statistics for all functions decorated with the `@measure` decorator. Be sure you understand how the `get_function_stats()` function works.



Now you will modify `sudoku.py` so that you can measure its performance.

5. ☐ Edit `sudoku.py` and complete the TODO steps.



Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns (continued)

6. ☐ Switch to a command prompt, change directory, and run the `sudoku.py` program:

```
cd C:\crs1906\exercises\ex06_design_patterns
python sudoku.py sudoku_input.txt
```



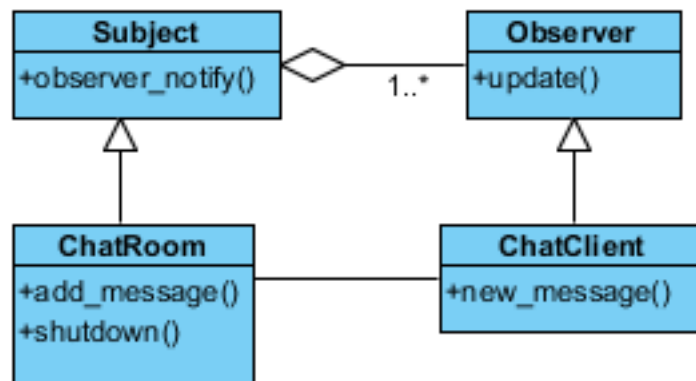
Which function had the largest average execution time?



If you have time, try implementing a `ChatRoom` based on the Observer pattern

In the first part of this exercise, you will implement a simple chat room that supports multiple chat clients. The design is based on the Observer pattern. The loose coupling of the Subject and Observers will make it possible to change to the implementation of one without affecting the other.

The following UML diagram shows the relationships between classes:



7. ☐ Edit the file, `observer.py`, and examine the code.



Note the definition of the `Subject` and `Observer` classes. The code is similar to the code discussed in the lecture, except the `Subject`'s `observer_notify()` method pushes data to the client.



Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns (continued)

8. ☐ Edit `chat_room.py` and perform the TODO steps. (Ignore the steps marked BONUS TODO for now.)



Note that the `ChatRoom` class (a subclass of `Subject`) knows nothing about the implementation of the `Observers`.

9. ☐ Edit `chat_client.py` and perform the TODO steps. (Ignore the steps marked BONUS TODO for now.)



Note that the `ChatClient` class, a subclass of `Observer`, knows nothing about the implementation of the `Subject`.



The `Observer` pattern allows the `ChatClient` and `ChatRoom` class to be very loosely coupled, which makes the classes much easier to maintain and update.

10. ☐ In the PyCharm Run menu, select **Unittests in test_chat_observer** and click the **Run** button.



Verify that all unit tests pass.



Congratulations! You have achieved loose coupling between classes with the Observer design pattern.



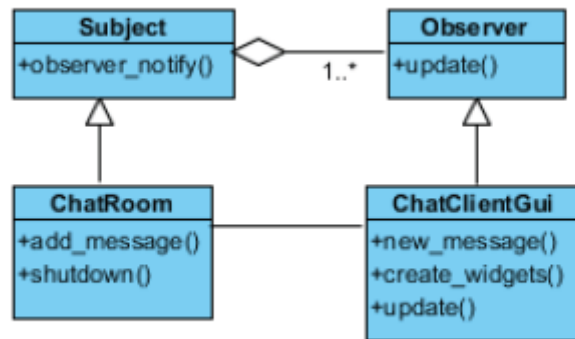
Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns (continued)



If you have more time, try implementing a GUI chat client

Next, you will use the code from the previous section's `ChatClient` to implement a GUI chat client. Because the design is based on the Observer pattern, the code for the `ChatRoom` class won't need any changes even though its concrete observers are completely different from those in the previous section.

The following UML diagram shows the relationships between classes:

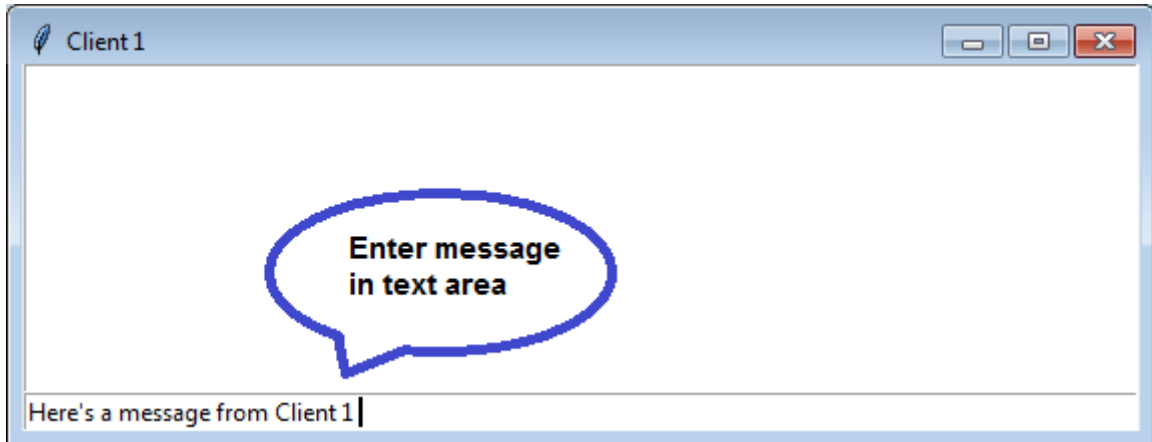


11. ☐ Edit the file `chat_gui_client.py` and perform the TODO steps.
12. ☐ In the PyCharm Run menu, select **Run chat_gui_client** and click the **Run** button.



Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns (continued)

13. ☐ Click the text entry at the bottom of a client window. Enter some text, then press <Enter>.



Verify that the message is broadcast to all client windows.



Because of the loose coupling between `Subject` and `Observer`, the `ChatRoomSubject` can support a completely different type of observer with no changes.



Congratulations! You have demonstrated that the Observer design pattern makes it easy to replace Observers with no effect on Subjects.



Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns (continued)

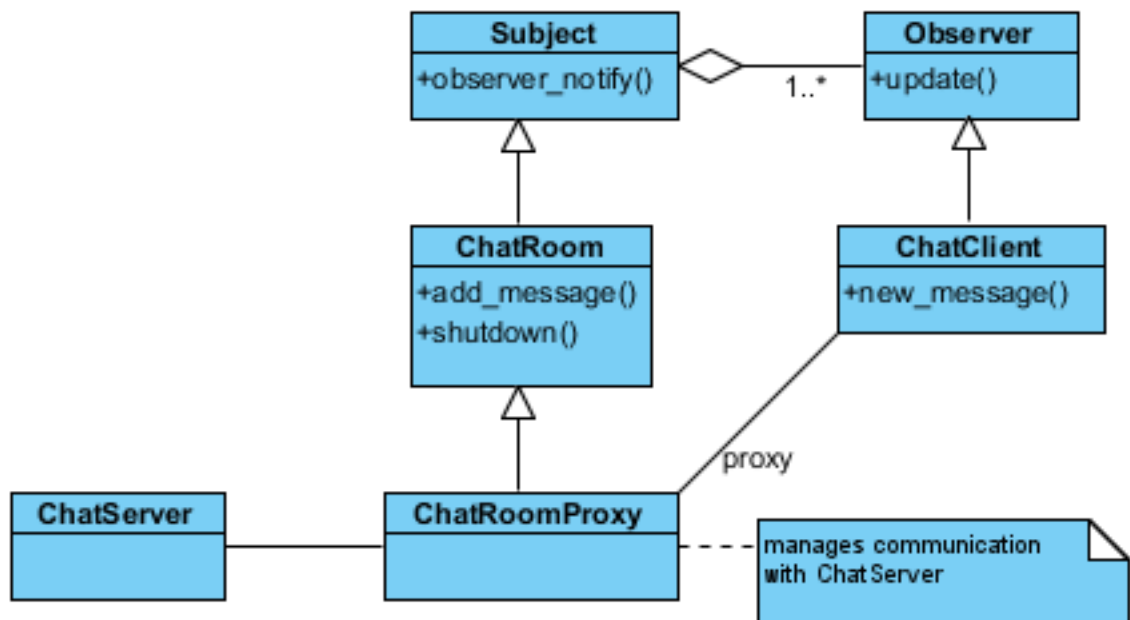


If you have more time, try implementing the Proxy design pattern

The simple chat room you implemented in the previous section only supports `Observers` that exist in the same Python interpreter. In this section, you will implement a chat server that supports clients running in remote Python interpreters—or even running in completely different languages!

Although the chat room implementation is more complex because of the network interaction, the Proxy design pattern allows the client code to work with no changes. The client will interact with a remote proxy that hides the details of the network interaction.

The following UML diagram shows the relationships between classes. Note that the `ChatClient` interacts with the `ChatRoomProxy` rather than the remote `ChatServer`:



14. ☐ Edit `chat_server.py` and perform the TODO steps.
15. ☐ Edit `chat_room_proxy.py` and perform the TODO steps.
16. ☐ Edit `chat_room.py` and perform the BONUS TODO steps.



Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns (continued)

17. ☐ Edit `chat_client.py` and perform the BONUS TODO steps.



Note that the only change required is to import the new proxy class. The chat room proxy implements the same interface as the original chat room, so the client code requires no changes at all.

18. ☐ In the PyCharm Run menu, select **Run chat_server** and click **Run**.

19. ☐ Open a command prompt and start an instance of the modified chat client (enter your name when prompted):

```
cd \crs1906\exercises\ex06_design_patterns
python chat_client.py
```

20. ☐ Open a second command prompt and start another instance of the modified chat client:

```
cd \crs1906\exercises\ex06_design_patterns
python chat_client.py
```

21. ☐ Enter some text in the first command prompt window and press <Enter>.



Verify that the message is displayed in the second command prompt window.

22. ☐ Enter several messages in both windows.



Verify that the chat interaction works correctly.

23. ☐ Press <Ctrl>C in the chat client windows to shut down the clients.



Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns (continued)

24. ☐ Switch to PyCharm. In the Run window (bottom left-hand corner), click the red square **Stop** button to shut down the chat server.



Congratulations! You have implemented a remote Proxy to hide the details of network interaction from a client.



If you have more time, try configuring a decorator with a configuration file

In practice, you may need to toggle a decorator on and off based on the environment. For example, you may want to enable the profile decorator for your development environment, but disable it for QA and production environments. In this section, you'll configure the decorator using a configuration file.

25. ☐ Switch to PyCharm and edit `measure_configurable.py`. Complete the steps marked **BONUS TODO**.
26. ☐ Edit `measure.ini` and complete the steps marked **BONUS TODO**.



By changing the configuration setting in `measure.ini`, you can enable or disable the profiling decorator.

27. ☐ Edit `sudoku.py` and change the `from measure import` statement at the top of the file to the following:

```
from measure_configurable import measure, get_function_stats,  
decorator_enabled
```



Hands-On Exercise 6.1: Applying the Decorator, Observer, and Proxy Design Patterns (continued)

28. ☐ Run `sudoku.py` with the profiling decorator enabled:

```
python sudoku.py sudoku_input.txt
```



Verify that the decorator gathers statistics as before.

29. ☐ Now edit `measure.ini` and change the value of `enabled` to **False**

30. ☐ Run `sudoku.py` with the profiling decorator disabled:

```
python sudoku.py sudoku_input.txt
```



Verify that the decorator didn't gather any statistics.



Congratulations! You have enabled and disabled a decorator using a configuration file.



This is the end of the exercise.





Objectives

In this exercise, you will

- Establish isolated Python environments with `venv`
- Install modules from the PyPI repository using `pip`
- Package Python modules for distribution

Overview

In this exercise, you will create virtual environments and install packages from PyPI. You will also package a module as both a source distribution and a wheel distribution.



Working with virtual environments

For this exercise section, you will work with the original version of the sudoku solver from Chapter 5.

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex07_distributing`, then click **OK**.
2. ☐ Open a command prompt and execute the `sudoku.py` script:

```
cd \crs1906\exercises\ex07_distributing
python sudoku.py sudoku_input.txt
```



Did the script complete successfully?

3. ☐ Switch to PyCharm and edit `sudoku.py`.



Notice all the syntax errors reported by PyCharm.



Which version of Python does the script need?



Hands-On Exercise 7.1: Installation and Distribution (continued)



When you ran the script, which version of Python did you use?



Execute this command: `python -V`



This script needs to run under Python 2.7, but your default environment uses Python 3.7. You often need to support different versions of Python to run different applications. The best way to handle this situation is to use virtual environments.



Creating a virtual environment

4. ☐ Switch to the command prompt.
5. ☐ Set a variable with the name of a new directory:

```
set VENV=\crs1906\exercises\ex07_distributing\py27venv
```

6. ☐ Create a new Python 2.7 virtual environment with `venv`:

```
\Python\Python2.7\Scripts\python.exe -m venv %VENV%
```



Important: Run `venv` from the installation of the Python version you want for the new virtual environment.

7. ☐ Open a file explorer and view the contents of `\crs1906\exercises\ex07_distributing\py27venv\Scripts`.



Note that `venv` copied the Python 2.7 interpreter and libraries to the new directory.

8. ☐ View the contents of `\crs1906\exercises\ex07_distributing\py27venv\Lib\site-packages`.



Note that the only modules installed in `site-packages` are `pip`, `setuptools`, `wheel`, and `easy_install` (the predecessor of `pip`).



`venv` creates a minimal Python environment. You can install additional modules in the virtual environment using `pip`.



9. ☐ Activate the new virtual environment:

```
%VENV%\Scripts\activate
```



Note that the command prompt changes to the name of the virtual environment.

10. ☐ Verify that the `python` command will run a Python 2.7 interpreter:

```
python -V
```

11. ☐ Execute the `sudoku` script again:

```
python sudoku.py sudoku_input.txt
```



Did the script complete successfully this time?



Any modules you install in the virtual environment will not affect any other Python installations. This makes virtual environments ideal for testing new modules.

12. ☐ Deactivate the virtual environment:

```
deactivate
```



Now which version of Python is run by the `python` command?



Congratulations! You have created a virtual environment.



Hands-On Exercise 7.1: Installation and Distribution (continued)



If you have time, try installing modules with `pip`

You will now install a module from PyPI into a virtual environment.

13. ☐ Activate the Python 2.7 virtual environment that you created before:

```
%VENV%\Scripts\activate
```

14. ☐ Start a Python interpreter and try to import the `requests` module.

```
python
import requests
```



Does the import succeed?



You need to install the `requests` package.

15. ☐ Exit the interpreter and execute the following command to install the `requests` module:

```
pip install requests
```



Note that `pip` automatically installs packages that are required by the `requests` package.

16. ☐ Start a Python interpreter and execute the following commands:

```
import requests
help(requests)
r = requests.get('http://www.google.com')
r.text[:250]
quit()
```





If the call to `get.requests()` fails with the message 'Max retries exceeded', just try it again.

If your site uses an HTTP proxy, add the proxy configuration to the `requests.get()` call. For example:

```
http_proxy = 'http://wwwproxy.se.axis.com:3128'  
proxy_dict = {'http': http_proxy}  
r = requests.get('http://www.google.com',  
proxies=proxy_dict)
```



Now the `requests` module is installed in the virtual environment.

17. ☐ Execute the following command to uninstall the `requests` module (when `pip` prompts you with `Proceed (y/n)?`, enter **y**):

```
pip uninstall requests
```



Can you import the `requests` module from a Python interpreter?

18. ☐ Did you get a warning message when you ran `pip`? (You are using `pip` version ..., however version ... is available)

If so, upgrade to the newest version of `pip` with the following command:

```
python -m pip install --upgrade pip
```



Note that you should not run `pip install --upgrade pip`, because the first thing `pip` would do is uninstall itself!

19. ☐ Deactivate the virtual environment:

```
deactivate
```



Congratulations! You have installed and uninstalled a module using `pip`.



Hands-On Exercise 7.1: Installation and Distribution (continued)



If you have more time, try building and installing a module

In this section, you will follow PyPA recommendations for best practices for building and installing a new Python module. The module is named `simple_tz`, and it provides a simple interface to the `pytz` module. `pytz` provides robust time-zone support for the `datetime` module. However, `pytz`'s API is a bit verbose, so our `simple_tz` module will provide a simple function to convert a time string from one time zone to another.

20. ☐ Switch to PyCharm and expand the `simple_tz` folder.



This is the top-level directory of the module you will build.

21. ☐ Expand the nested `simple_tz` directory.



This directory defines the `simple_tz` package. Note that the module's directory structure follows the PyPA recommendations.

22. ☐ Edit `tz.py` and perform the TODO steps.
23. ☐ Edit `setup.py` in the parent `simple_tz` folder and perform the TODO steps.
24. ☐ Open a command prompt and execute the following command to build the module's source and wheel distributions:

```
cd \crs1906\exercises\ex07_distributing\simple_tz
python setup.py sdist bdist_wheel
```



`setup.py` created a `dist` directory and wrote the distribution files to that directory.

25. ☐ List the contents of the `dist` directory.



Which files did `setup.py` create?



Now you'll test the wheel distribution by installing it in a new virtual environment.



26. ☐ Create a new Python 3.7 virtual environment.

27. ☐ Activate the new environment.

28. ☐ Use `pip` to install the wheel file into the virtual environment:

```
cd dist
pip install simple_tz-1.0.0-py2.py3-none-any.whl
```



Note that `pip` installed the `pytz` module, which was listed as a dependency in `setup.py`.

29. ☐ Start a Python interpreter and test the module's installation by entering the following commands:

```
from simple_tz import tz
help(tz)
tz.convert('2020-12-31 16:00:00', 'PST', 'CET')
```

If you make changes to files in the package, you must rebuild the distribution and upgrade the installed package using the following command:

```
pip install -U simple_tz-1.0.0-py2.py3-none-any.whl
```



Were you able to import the new module and call its function?



After you install a module, you can use it exactly as you would a module you installed from PyPI.

30. ☐ Exit the Python interpreter and deactivate the virtual environment.

31. ☐ Congratulations! You have built and installed a module.



Hands-On Exercise 7.1: Installation and Distribution (continued)



If you have more time, try setting up a local repository

Creating a local repository of Python modules and packages allows your organization to share modules among development teams using standard distribution and installation techniques. Users of the modules can simply run `pip` to install modules from your local repository, exactly as they install modules from PyPI.

32. ☐ Set up a local Python repository using the `pypiserver` package.



If needed, refer to the course notes and review "Setting Up a Local Repository" and "Installing Packages from Local Repositories".

33. ☐ Copy the wheel and source distributions that you created earlier to the `pypiserver` distribution directory.



The distributions are in the directory `simple_tz\dist`.

34. ☐ Start the `pypiserver`. If you get a popup from Windows Firewall, click **Allow Access**.

35. ☐ Open a new command prompt and change the directory to `\crs1906\exercises\ex07_distributing`.

36. ☐ Create a Python 3.7 virtual environment.



Remember to activate the environment.

37. ☐ Install your module from the local `pypiserver`.

38. ☐ Start the Python interpreter from the virtual environment.

39. ☐ Import your module.



40. ☐ Execute the imported function:

```
python
from simple_tz import tz
tz.convert('2020-12-31 15:00:00', 'PST', 'CET')
```



Verify that the function executes without errors.

41. ☐ Close the new command prompt, and shut down `pypiserver` in the original command prompt with `<Ctrl><C>`.
42. ☐ Deactivate the virtual environment.



Congratulations! You have set up a local Python module repository.



This is the end of the exercise.





Objectives

In this exercise, you will

- Create and manage multiple threads of control with the `Thread` class
- Launch and manage subprocesses using the `subprocess` module
- Synchronize threads with locks

Overview

In this exercise, you will write multithreaded code to speed up an I/O-bound application. You will also guard against potential race conditions using locks.



Writing multithreaded code

Although multithreading can improve application performance dramatically, it can also lead to bugs that are very difficult to reproduce and fix. So before adding threads to your code, you need to analyze it carefully for thread safety.

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex08_concurrency`, then click **OK**.
2. ☐ Edit `chat/chat_server.py`.



This is the socket-based chat server from Exercise 6.1.



Review the code in `chat_server.py`. Are the methods of the `ChatServer` class thread-safe?



Answer:

No. The shared `chat_sockets` attribute is the source of the thread-safety problems. Because it is defined as a class attribute, it is shared by all instances of the `ChatServer` class. If two request threads access `chat_sockets` at the same time, the data structure may become corrupted.



Hands-On Exercise 8.1:

Concurrency (continued)



Where in the code are the thread-safety problems?



Hint...

There are four.



Answer:

1. In the `handle()` method, the statement
`ChatServer.chat_sockets.add(self.request)`
2. In the `handle()` method, the statement for `socket` in
`ChatServer.chat_sockets`
3. In the `handle()` method, the statement
`ChatServer.chat_sockets.remove(self.request)`
4. In the `shutdown()` method, the statement for `socket`
in `cls.chat_sockets`



How can you ensure that two threads can't access `chat_sockets` at the same time?



Answer:

Add a lock and make sure each thread acquires the lock before accessing `chat_sockets`.

3. ☐ In the `ChatServer` class in `chat_server.py`, add a class attribute of type `Lock()`



Hint...

Add a line of code immediately after the class definition:
`chat_sockets_lock = Lock()`



4. ☐ For each of the four statements with thread-safety problems, add a statement to ensure that threads acquire the lock before continuing.



Hint...

Review Python Idiom: Managing Locks in the course notes.

To indent a block of code, highlight the code and press <Tab>.

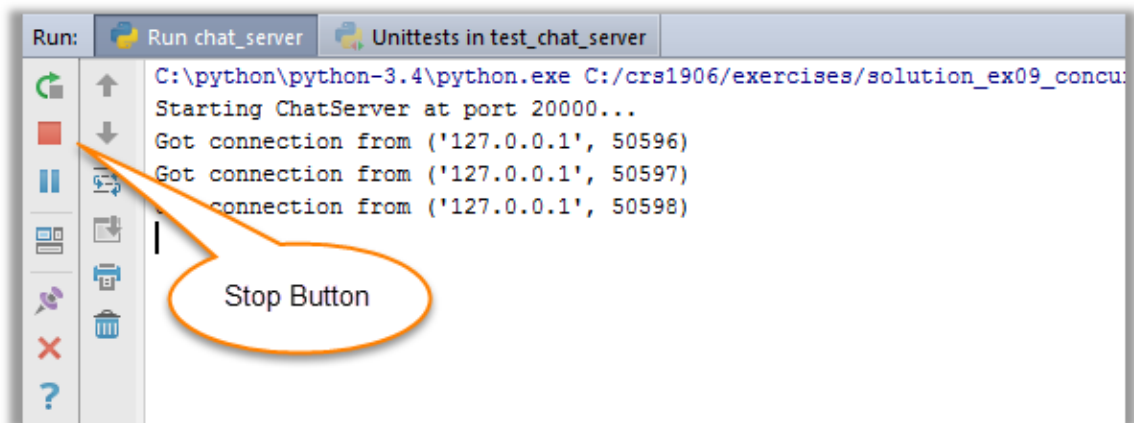
```
with ChatServer.chat_sockets_lock:
    ... # statement that needs to be protected
```

5. ☐ In PyCharm's Run menu, select **Run chat_server** and click **Run**.
6. ☐ In PyCharm's Run menu, select **Unittest test_chat_server** and click **Run**.



Verify that all unit tests pass.

7. ☐ Shut down the chat server:
- Click the **Run chat_server** tab at the bottom of the PyCharm window
 - Click the red square **Stop** button



Congratulations! You have written a multithreaded application. You have also ensured thread safety using a lock.

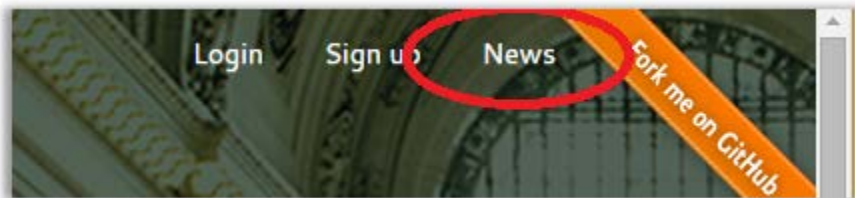


Hands-On Exercise 8.1: Concurrency (continued)



If you have time, try writing a multithreaded RSS feed reader

8. ☐ Open Chrome. On the toolbar, click **Exercise Links**, then select **Exercise 8.1 Web App**.
9. ☐ At the upper right-hand corner of the TicketManor home page, click the **News** link.



The TicketManor application downloads three RSS news feeds to populate this page. Currently, the code downloads the feeds in a serial manner: it doesn't start downloading the second feed until the first download is complete.



Is this application CPU-bound or I/O-bound?



Answer:

This is an I/O-bound application.



Which is more likely to improve this application's performance, concurrency with threads or parallelism with processes?



Answer:

Concurrency with threads usually helps I/O-bound applications.





If you have more time, try adding threads to the `TicketManager` news feed reader

- 10. ☐ Switch to PyCharm and open `ticketmanor/rest_services/feed_reader/all_news_feed_reader.py`.
- 11. ☐ Edit `all_news_feed_reader.py` and complete the TODO steps.
- 12. ☐ Switch to the command prompt. If a server is running, press `<Ctrl><C>` to stop it, then execute the following commands to start the `TicketManor` application:

```
cd C:\crs1906\exercises\ex08_concurrency
ticketmanor
```

- 13. ☐ Switch to Chrome. In the toolbar, click the `TicketManor` button. If Windows Firewall pops up a Security Alert dialog, click **Allow Access**.



*Verify that the title in the page's upper left-hand corner is **Exercise 8.1**. If not, press `<Ctrl><Shift><R>` to reload the page.*

- 14. ☐ Click the **News** link in the upper right-hand corner.



Verify that all three news feeds are displayed. If the news is not displayed:

1. *Check the command prompt window for error messages from the server.*
2. *Check the network icon in the Windows system tray. If necessary, right-click the icon and run the network troubleshooter.*



Congratulations! You have improved an application's performance by adding threads.



This is the end of the exercise.





Objectives

In this exercise, you will

- Parallelize application execution with process pools and Executor classes
- Simplify parallel algorithms by using Queue objects

Overview

In this exercise, you'll use multiprocessing to parallelize the execution of an application.



Writing an application with multiprocessing

In the first section of this exercise, you'll compare the execution times of serial and parallel implementations of an algorithm. You'll be working with a Monte Carlo approximation of the value of pi. The Monte Carlo technique uses the distribution of random values in its calculation.

First, you'll complete a multiprocessing version of the function. Then you'll compare the execution times of your multiprocessing version with a serial computation.

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex08_multiprocessing`, then click **OK**.
2. ☐ Edit `pi_monte_carlo/pi.py` and complete the TODO steps.
3. ☐ In PyCharm's Run menu, select **Run pi** and click **Run**.



Verify that `pi.py` ran successfully. With a relatively small number of samples, the value of pi produced by the Monte Carlo strategy is not very accurate. :)



Congratulations! You have used multiprocessing to parallelize the execution of an application.



Hands-On Exercise 8.2: Multiprocessing (continued)



If you have time, try comparing serial and parallel computations

Now you'll compare execution times of several versions of the calculation.

4. ☐ Add the following statements to the end of the `pi.py` script to time the execution of the `pi_async()` function:

```
from timeit import timeit
time = timeit('pi_async()',
              setup='from __main__ import pi_async',
              number=1)
print('pi_async execution time:', time)
```



Note that `__main__` has two leading and two trailing underscores.

5. ☐ Run `pi.py` again.



What was the execution time of the `pi_async()` function?

6. ☐ Edit `pi.py` and examine the `pi_serial()` function.



This is a serial version of the Monte Carlo strategy.

7. ☐ Copy and paste the `timeit()` call at the end of the script. Change all occurrences of `pi_async` to `pi_serial`.



Hint...

```
time = timeit('pi_serial()',
              setup='from __main__ import pi_serial',
              number=1)
print('pi_serial execution time:', time)
```

8. ☐ Run `pi.py` again.





What were the execution times of the two functions?



The parallel processes improve the performance of this calculation. Depending on the system load, the parallel version may be nearly twice as fast.



Congratulations! You have compared the performance of serial and parallel computations.



If you have more time, try comparing threads and processes

Now you'll compare the performance of the process-based application to a thread-based version.

9. ☐ Edit `pi.py` and scroll to the `pi_async()` function.
10. ☐ Replace the `ProcessPoolExecutor` with a `ThreadPoolExecutor`. The `ThreadPoolExecutor` constructor requires a `max_workers` argument. Give `max_workers` the value `ntasks`.



Hint...

Review the course notes for Best Practices on Concurrency.

11. ☐ Run `pi.py` again.



What were the execution times?



The multithreaded version runs slower than the serial version.



Hands-On Exercise 8.2: Multiprocessing (continued)



Why did multiprocessing help the performance, and multithreading hurt the performance?



Answer:

This is a CPU-bound application. Processes can use multiple cores simultaneously, but Python threads share one core.

12. ☐ In `pi.py`, replace the `ThreadPoolExecutor` with the original `ProcessPoolExecutor`, and then experiment with different values of the `ntasks` variable. See if you can find the number that gives the best performance.



Congratulations! You have compared the performance of processes and threads.



If you have more time, try using a `ThreadPoolExecutor` in a news feed reader

In this section, you will modify the news feed reader from Exercise 8.1 to use a `ThreadPoolExecutor` to manage the threads that read news feeds.

13. ☐ Edit the file `ticketmanor/rest_services/feed_reader/all_news_feed_reader.py`. Complete the TODO steps.
14. ☐ Switch to a command prompt. If a server is running in the command prompt, press `<Ctrl><C>` to terminate the server. Then execute the following commands:

```
cd \crs1906\exercises\ex08_multiprocessing
ticketmanor
```

15. ☐ Switch to Chrome and click the **TicketManor** button on the toolbar.



*Verify the page title is **Exercise 8.2**. If not, press `<Ctrl><Shift>R` to reload the page.*



16. ☐ Click the **News** link.



Verify that the News page still displays three types of news. If the news is not displayed:

- 1. Check the command prompt window for error messages from the server.*
- 2. Check the network icon in the Windows system tray. If necessary, right-click the icon and run the network troubleshooter.*



Congratulations! You have used a `ThreadPoolExecutor` to manage concurrent threads in a news feed reader.



This is the end of the exercise.





Objectives

In this exercise, you will

- Send REST requests from a Python client
- Consume JSON response data
- Build a Python implementation of a REST web service

Overview

In this exercise, you will write code that sends HTTP requests to a REST service.



Writing a REST client

In the first exercise section, you will use the `requests` module to send requests to a REST service provided by the `TicketManor` web application. The `TicketManor` frontend, implemented with the AngularJS JavaScript framework, uses REST requests to communicate with the backend Python services.

Your Python code will interact with the `TicketManor` REST service that manages users of the application. The data in body of the request and the response will be JSON.

1. ☐ Open a command prompt and execute the following commands to start the `TicketManor` web app:

```
cd C:\crs1906\exercises\ticketmanor_webapp
ticketmanor
```

2. ☐ Open Postman. Fetch the `TicketManor` user, `ned.flanders@gmail.com`, by sending a `GET` request to the following URL:

```
http://localhost:6543/rest/users/ned.flanders@gmail.com
```



What is the type of the response?



You sent a `GET` request to the `TicketManor` REST service, which queried its database and returned a JSON response. If the `TicketManor` service doesn't respond within a few seconds, switch to Chrome, click the `TicketManor` button in the toolbar, and press `<Ctrl><Shift>R` to reload the web app.



Hands-On Exercise 9.1: Interfacing With REST Web Services and Clients (continued)

3. ☐ Now you will write a Python client of the TicketManor REST service. In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\ex09_rest_services`, then click **OK**.



You will implement the REST client calls in unit test cases in `test_user_rest_service_json.py`. The test cases will send requests to the REST service using different URIs and HTTP methods.

4. ☐ Edit `test_user_rest_service_json.py` and complete the TODO steps.
5. ☐ In PyCharm's Run menu, select **pytest in test_user_rest_service_json** and click the **Run** button.



Verify that all unit tests pass.



Congratulations! You have written a Python client for a REST service.



If you have time, try implementing a REST service

In this section, you will implement a REST service using the Flask framework.

6. ☐ Switch to the command prompt. If a server is running, press `<Ctrl><C>` to shut it down.
7. ☐ In PyCharm's Project window, expand the `user_service` folder.
8. ☐ Edit `rest_server.py` and complete the TODO steps.
9. ☐ In PyCharm's Run menu, select **Run rest_server** and click the **Run** button.



Verify that PyCharm's Run output window displays a message that includes `Running on http://127.0.0.1:5000`



Hands-On Exercise 9.1: Interfacing With REST Web Services and Clients (continued)

10. ☐ In PyCharm's Run menu, select **pytest in test_rest_server** and click the **Run** button.



Verify that all test cases pass.

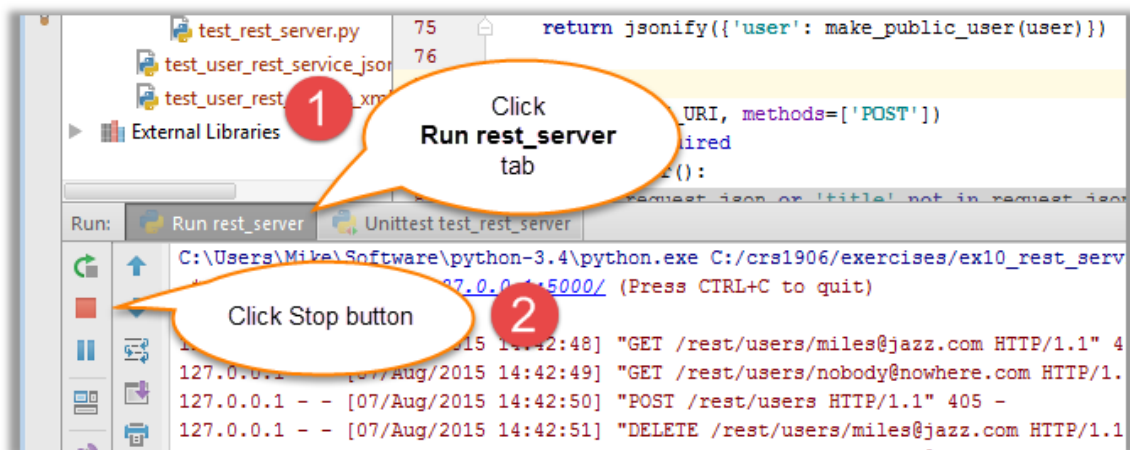


If test cases fail, check the logging output in the service's Run window console:

The screenshot shows the PyCharm Run window with two tabs: 'Run rest_server' and 'Unittests in test_rest_server'. The 'Unittests in test_rest_server' tab is active, displaying the following output:

```
Creating user miles@jazz.com
-----
127.0.0.1 - - [22/Oct/2016 16:27:01] "POST /rest/users HTTP/1.1" 201 -
127.0.0.1 - - [22/Oct/2016 16:27:02] "PUT /rest/users/miles@jazz.com HTTP/1.1" 201 -
Traceback (most recent call last):
  File "C:\python\python-3.4\lib\site-packages\flask\app.py", line 1836,
    return self.wsgi_app(environ, start_response)
```

11. ☐ When you are finished testing, shut down the REST server:
- In the Run window, click the **Run rest_server** tab, then
 - Click the **Stop** button



Hands-On Exercise 9.1: Interfacing With REST Web Services and Clients (continued)



Congratulations! You have implemented a REST service in Python.



This is the end of the exercise.



Objectives

In this exercise, you will

- Call C functions from Python applications using the `ctypes` module
- Extend Python's functionality with C extension modules
- Simplify the implementation of C extension modules using Cython

Overview

In this exercise, you will use several different techniques for calling C functions to boost Python application performance.



Calling C functions using `ctypes`

For this exercise section, you will call C functions from the `mandelbrot.py` script that you worked with in Chapter 5. You'll also run the script with and without the C function calls to compare the performance of the application.

1. ☐ In PyCharm, select **File | Open** and navigate to `C:\crs1906\exercises\appA_extending_python`, then click **OK**.
2. ☐ Edit `mandelbrot_ctypes/mandelbrot.py` and complete the TODO steps.



This is a pure Python version of the script, which performs all calculations in Python.

3. ☐ Edit `mandelbrot_ctypes/calc_z_ser.h` and complete the TODO steps.
4. ☐ Edit `mandelbrot_ctypes/calc_z_ser.c` and complete the TODO steps.
5. ☐ Edit `mandelbrot_ctypes/mandelbrot_ctypes.py` and complete the TODO steps.



This is a version of the script that calls the C function in `calc_z_ser.c` to perform the main calculation.



Hands-On Exercise A.1: Extending Python (continued)

6. ☐ Close the old command prompt and open a new command prompt. Execute the following commands to create a DLL for the `calc_z_ser()` function:

```
cd \crs1906\exercises\AppA_extending_python
activate
cd mandelbrot_ctypes
make
```

7. ☐ Do a listing of the directory with `dir` or `ls`.
8. ☐ Verify that a DLL was created.
9. ☐ Execute the `mandelbrot_ctypes.py` script, which now calls a C function:

```
python mandelbrot_ctypes.py 600 500
```



Verify that the script runs successfully.

10. ☐ Compare the performance of the pure Python version and the version that calls the C function:

```
python mandelbrot.py 600 500
python mandelbrot_ctypes.py 600 500
```



What are the times of the `calculate_z_serial()` function calls for the two different versions?



Although the translation to C makes the function harder to understand, test, and maintain, it gives better performance compared to the pure Python version.



Congratulations! You have called C code from a Python application and analyzed the performance difference.





If you have time, try writing a C extension module

Calling C functions directly from Python scripts, as we did in the first section of this exercise, requires adding a lot of setup code in the calling Python script. If your project has many calls to the same C functions, you can eliminate most of the repetitive setup code by writing an extension module in C.

In this exercise, you'll create a C extension module for the `calc_z_ser()` function from the previous exercise.

11. ☐ Edit `mandelbrot_ext_module/calc_z_ser.h` and complete the TODO steps.
12. ☐ Edit `mandelbrot_ext_module/calc_z_ser.c` and complete the TODO steps.
13. ☐ Edit `mandelbrot_ext_module/calc_z_ser_ext_mod.c` and complete the TODO steps.
14. ☐ Edit `mandelbrot_ext_module/setup.py` and complete the TODO steps.
15. ☐ Open a command prompt and execute the following commands to create an extension module for the `calc_z_ser()` function:

```
cd \crs1906\exercises\AppA_extending_python
\mandelbrot_ext_module
python setup.py build_ext --inplace
```



Do a listing of the directory and confirm that a DLL was created.

16. ☐ Edit `mandelbrot_ext_module/mandelbrot_ext_mod.py` and complete the TODO steps.
17. ☐ Execute the `mandelbrot_ext_mod.py` script:

```
python mandelbrot_ext_mod.py 600 500
```



Verify that the script runs successfully.



Hands-On Exercise A.1: Extending Python (continued)

18. ☐ Compare the performance of the pure Python version and the version that calls the C function:

```
python mandelbrot.py 600 500  
python mandelbrot_ext_mod.py 600 500
```



What are the run times of the calculations for the two different versions?



Writing a C extension module requires more knowledge of C than simply calling a C function. However, it can greatly simplify the calling Python code, and may be worth the effort for C functions that your project uses frequently.



Congratulations! You have write a C extension module for Python.



If you have more time, try creating an extension module with Cython

In this section, you will translate a Python module to Cython and analyze the performance of a program that uses the new module.

Cython is a language that lets you write C extension modules with a Python-like syntax. You can often get big performance gains with relatively minor changes.

19. ☐ Edit `mandelbrot_cython/mandelbrot_cython.py` and complete the TODO steps.
20. ☐ Edit `mandelbrot_cython/calc_z_ser.pyx` and complete the TODO steps.



Note that the Cython module uses a file extension of `.pyx` instead of `.py`.

21. ☐ Edit `mandelbrot_cython/setup.py` and complete the TODO steps.



22. ☐ Execute the following command to compile the Cython module:

```
cd \crs1906\exercises\AppA_extending_python\mandelbrot_cython  
python setup.py build_ext --inplace
```



Do a listing of the directory and confirm that a C source file and a .pyd file (Python dynamic link library) were created.

23. ☐ Execute the Python script that uses the Cython module:

```
python mandelbrot_cython.py 600 500
```



Verify that the script runs successfully.

24. ☐ Compare the performance of the pure Python version and the version that uses the Cython module:

```
python mandelbrot.py 600 500  
python mandelbrot_cython.py 600 500
```



What are the times of the `calculate_z_serial()` function calls for the two different versions?



Writing extension modules in Cython is usually much simpler than writing pure C extension modules, and Cython can often improve performance considerably compared to a pure Python program.



Congratulations! You have implemented a C extension module using Cython.



This is the end of the exercise.





Objectives

In this exercise, you will

- Trace program execution with the standard `pdb` debugger

Overview

Debugging with an IDE is generally easier than debugging with `pdb`. However, there are times when `pdb` is useful; for example, for post-mortem debugging (that is, for debugging program crashes).

The script `asteroids.py` is a simple Python version of the classic Asteroids video game. There's a bug in the code that causes the program to crash. You're going to use `pdb` to diagnose and fix the problem.



Debugging with `pdb`

1. ☐ Open a command prompt and execute the following commands:

```
cd \crs1906\exercises\appB_advanced_features
python asteroids.py
```



Note that the program crashes.

2. ☐ To start post-mortem debugging, run the program again from the command line with Python's `-i` option to enter interactive mode when the program terminates:

```
python -i asteroids.py
```



Note the interpreter prompt ">>>" is displayed after the traceback.

3. ☐ Execute the following commands to start `pdb`'s port-mortem debugging:

```
import pdb
pdb.pm()
```



Hands-On Exercise B.1: Debugging With pdb (continued)

4. ☐ Now the prompt is `(Pdb)`, indicating that you are in the `pdb` debugging environment. Execute the `list` command to get a listing of the code in the current stack frame:

```
list
```

```
179 def intersect(object1, object2):
180     -> dist = math.sqrt((object1.xcor() - object2.xcor()) ** 2 +
181                       (object1.ycor() - object2.ycor()) ** 2)
182
183     radius1 = object1.getRadius()
184     radius2 = object2.getRadius()
185
(Pdb)
```

5. ☐ Display the values of the `object1` and `object2` arguments to the `intersect()` function:

```
object1
object2
```



What is the value of object2?

6. ☐ Where are you in the call stack? To find out, enter the `where` command:

```
where
```

7. ☐ Enter the `up` command to move up the call stack to the caller of the `intersect()` method:

```
up
```

8. ☐ Get a listing of the calling method:

```
list
```





Can you spot the problem?

```
(Pdb) list
218     ship.move()
219
220     gameover = False
221     for asteroid in asteroids:
222         asteroid.move()
223     ->     if intersect(ship, gameover):
224             turtle.color('red')
225             turtle.write("BOOM!!!", font=("Arial", 30),
226                         gameover = True
227
```



Answer:

Aha! On line 223, the call to `intersect()` is passing `gameover` instead of `asteroid`.

9. ☐ Exit the debugger with the following commands:

```
q
exit()
```

10. ☐ Edit `asteroids.py` and change line 223 to the following:

```
if intersect(ship, asteroid):
```

11. ☐ Save the file.

```
python asteroids.py
```



Hands-On Exercise B.1: Debugging With `pdb` (continued)

12. ☐ Now play Asteroids!

- Fire engine: `<up arrow>`
- Turn ship: `<left arrow>`, `<right arrow>`
- Shoot photon torpedo: `x`
- Quit: `q`



Congratulations! You have debugged a program with `pdb`.



This is the end of the exercise.

