**Assignment T5: Second Iteration**

**Team Name:** Alchemist

**Members:**

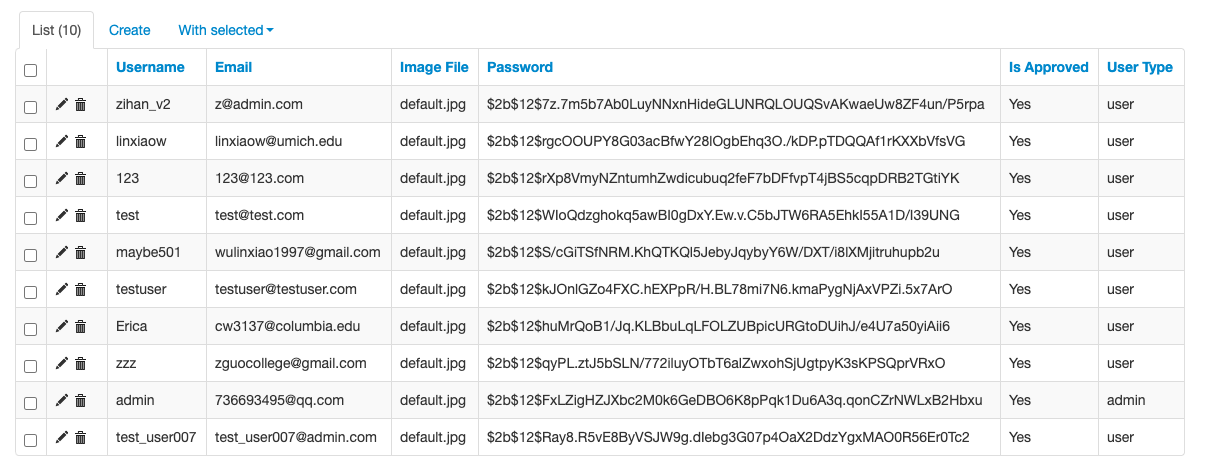
[Zihan Guo](mailto:zg2391@columbia.edu) , [Erica Chenchen Wei](mailto:cw3137@columbia.edu) [Jialiang Zhang](mailto:jz3283@columbia.edu) [Linxiao Wu](mailto:lw2944@columbia.edu)

Part 1. User story

*Revise your user stories, from the first iteration and/or project proposal, to reflect what is now fully implemented, tested and working. Do not include anything that is not actually evident in your github repository and/or you will not be able to show in your final demo.*

*Compare the user stories given in the second iteration report to those given in the first iteration report (part 2). If there are any differences you did not expect, you should discuss the discrepancies with the team. There is no grade for this part, except do not grade the rest of the report if the user stories are missing.*

Between Release 1 and Release 2, we have made the following changes:

1. [Jialiang Zhang](mailto:jz3283@columbia.edu) has added **OAUTH 2.0** support.
2. [Jialiang Zhang](mailto:jz3283@columbia.edu) has improved our **admin panel** such that admin users are able to approve/disapprove new user registration requests. In addition, an admin user can make another user an admin.
   1. 
3. [Zihan Guo](mailto:zg2391@columbia.edu) has set-up CI/CD using **AWS CodePipeline** and **AWS Elastic Beanstalk**
4. [Erica Chenchen Wei](mailto:cw3137@columbia.edu) improved integration of our flask application with dask application using **werkzeug.middleware.dispatcher**
5. [Linxiao Wu](mailto:lw2944@columbia.edu) improved strategy failure events using pop-up alerts; in addition, we improved our U.I. by displaying the error trace and style check failure reason on the same page.

**Users: Investment Professionals**

**<Strategy Upload>:** As an investment professional, I want to easily upload all trading strategies so that I can get results and compare them smoothly. My conditions of satisfaction are <

Common case:

1. Investment professionals should log in before they can upload their strategies.
2. Investment professionals upload the valid trading strategies(.py) file and then they have options to upload or not.
3. After uploading, they can choose to use which strategy to run backtesting.
4. After backtesting is done, then can view the result and save to compare with others.

Special cases:

1. Investment professionals upload an invalid strategy, which can’t be compiled, a warning message will prompt up to let them know.
2. Investment professionals do not have an account

**Users: Administrators**

<**Access Control**>: As an <admin>, I want to <manage users’ access to the system> so that this user can log in to the application and access relevant information. My conditions of satisfaction are <

common case:

Admin approves the user's login and adds the user to the User table. Admin disapproves a user’s login by removing the user from the User table.

Special cases:

1. Admin tries to approve a user's login request when the email matches an existing record in the user database.

Part 2. Equivalence Partitions & Boundary Conditions

*Write a test plan that explains the equivalence partitions and boundary conditions necessary to unit-test each of the major subroutines in your system (methods or functions, excluding constructors, getters/setters, helpers, etc.) and then implement your plan. Associate the names of your specific test case(s) with the corresponding equivalence partitions and boundaries (if applicable). Your test suite should include test cases from both valid and invalid equivalence partitions, and just below, at, and just above each equivalence class boundary (or inside vs. outside the equivalence class when boundary analysis does not apply). Note the same test case might apply to multiple equivalence classes. Say there is a method whose input should be an integer between 1 and 12. Then there is an equivalence class 1-12, an equivalence class <1 (or <=0), and an equivalence class >12 (or >= 13). A test case with input 0 would be just below the lower boundary of the equivalence class 1-12 and also at the high boundary of the equivalence class <1.*

*Include the link to the folder in your github repository that contains your automated test suite.*

We have equivalence partitioning & boundary condition tests for our major module:

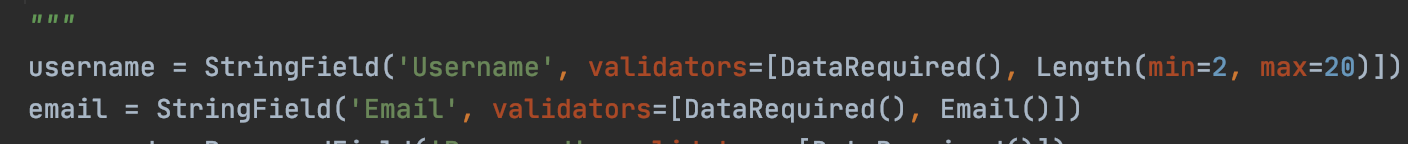
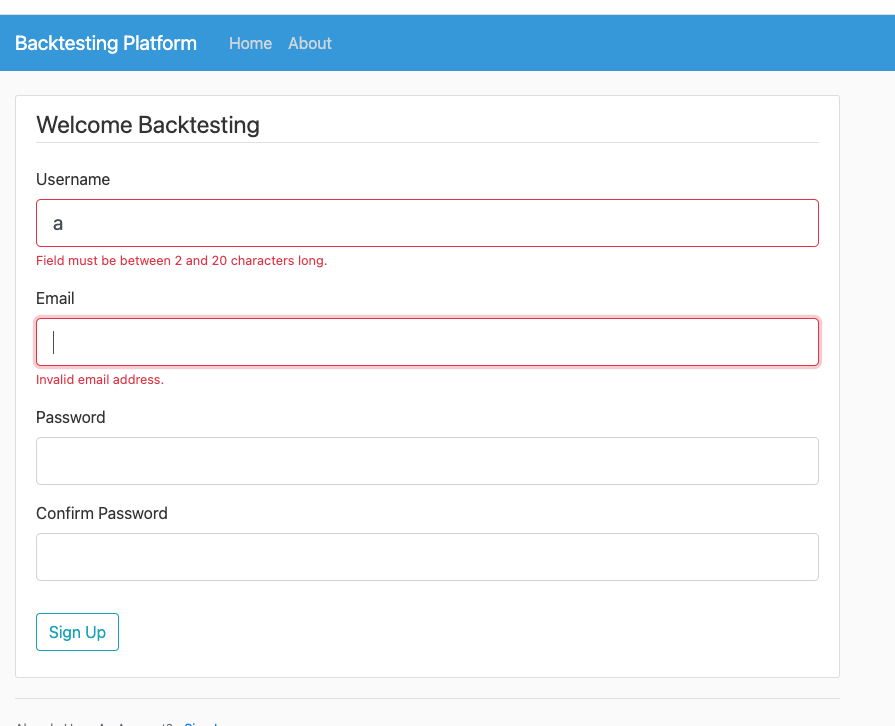
1. **Registration & login:** we have both username check and password check and test
2. **Upload**: we have file name, type, length check and test
3. **Backtesting:** Not applicable because this module does not accept user input
4. **Visualization**: strategy id check and test

Names of test case: Username length

We consider a user's username when a new user tries to register, the length of username should be in the range from 2 to 20. The equivalence class is [2, 20] inclusively. Here are the test conditions when the username length drops in [0, 2), [2, 20], (20, 50] :

1. Length of username less than 2 is considered invalid. Username cannot be a string that is less than two characters. [when username is in [0, 2)]
2. Length of username from 2 to 20 is considered valid. Users can choose a username that contains at least 2 characters and at most 20 characters. [when username is in [2, 20]]
3. Length of username greater than 20 is considered invalid. Users cannot choose a username that contains more than 20 characters. [when username length is in (20, 50]]

Case 1:



Case 1 & Case 3:

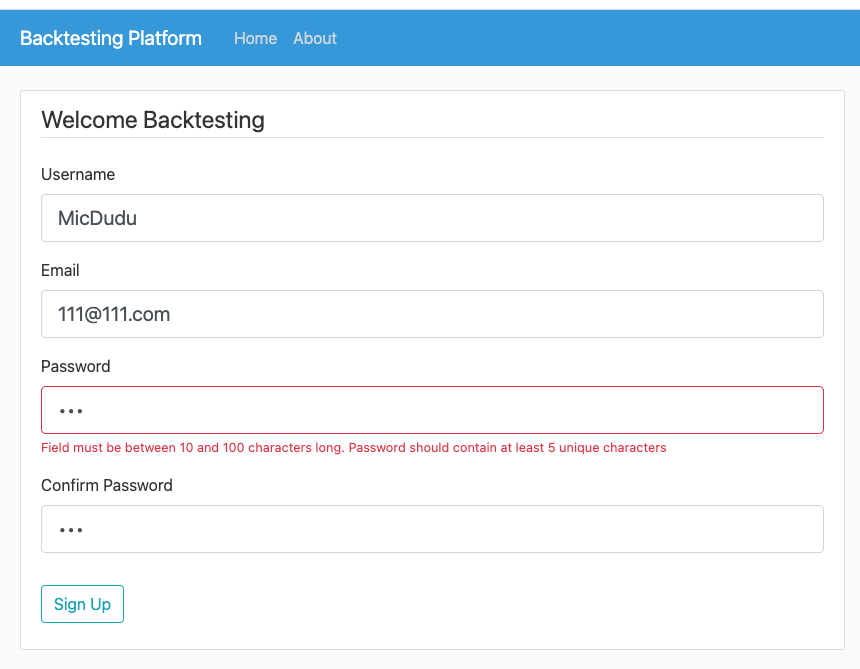


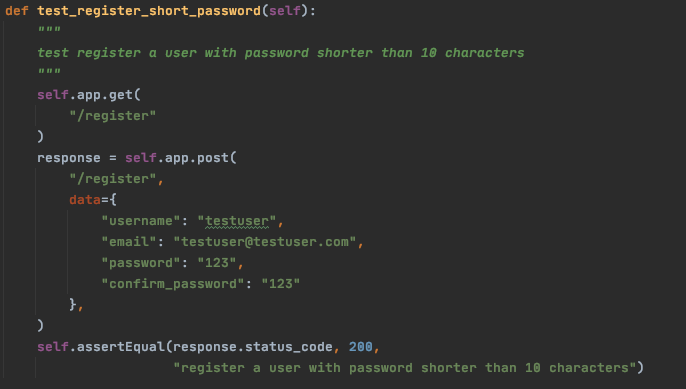
Case 2 (valid):



Names of test case: Password length

We consider a user's password when a new user tries to register, the length of password should be in the range from 10 to 100. The equivalence class is [10, 100] inclusively. Here are the test conditions when the password length drops in [0, 10), [10, 100], (100, 500] :

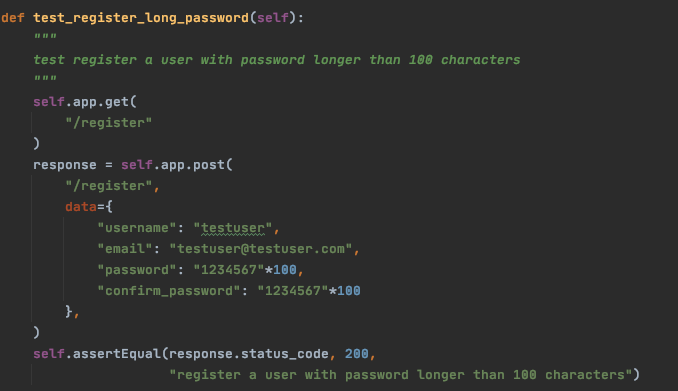
1. Length of password less than 10 is considered invalid. Password cannot be a string that is less than ten characters. [when password is in [0, 10)]
2. Length of password from 10 to 100 is considered valid. Users can choose a password that contains at least 10 characters and at most 100 characters. [when password is in [10, 100]]
3. Length of password greater than 100 is considered invalid. Users cannot choose a password that contains more than 100 characters. [when password length is in (100, 500]]

Case 1: 

Case 2 (valid):



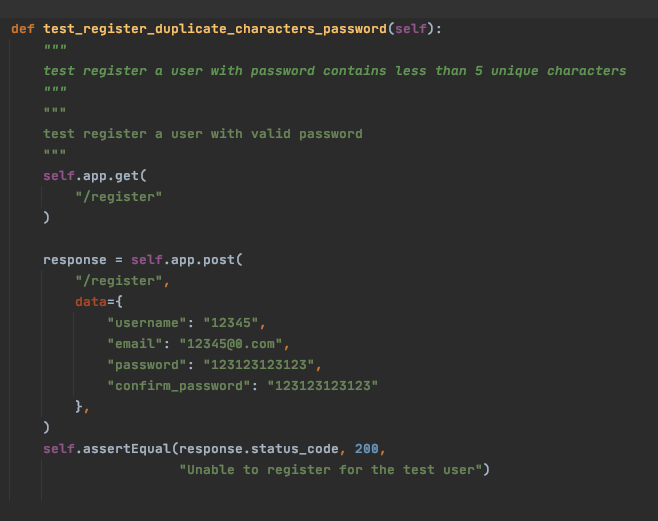
Case 3:

Names of test case: Password Uniqueness 

We also consider the password content when a new user tries to register, the password should contain 5 or more unique characters. Here are the test conditions when the number of unique characters in the password drops in [0, 5), [5, 100]:

1. The number of unique characters in the password less than 5 is considered invalid. Password cannot be a string that contains less than 5 unique characters. [when number of unique character is in [0, 5)]
2. The number of unique characters in the password at least 5 is considered valid. Password should be a string that contains 5 or more unique characters. [when number of unique character is in [5,100]]

Case 1:



Case 2 (valid):



Names of test case: Upload Filename length

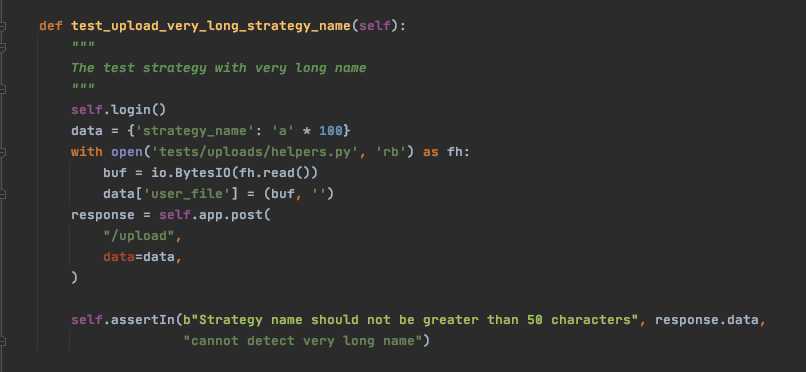
We corporate this case in acceptance testing on the uploading files, where equivalence class for input filename length is (0, 50) exclusively. We consider three partitions of the filename length , [0], (0, 50), and [50, 100]

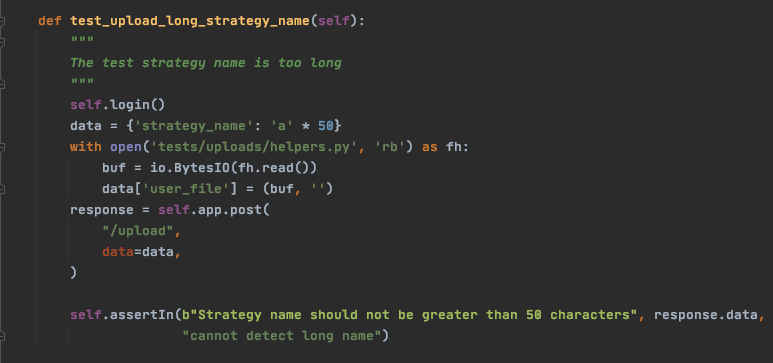
1. If the user doesn’t give an empty filename, which the filename length is 0, it will give a warning and the user can’t upload the file with an empty filename. We test with “” filename. This is also a boundary condition. We also tested boundaries by passing in filename lengths of 49, 50 and 51 respectively since 50 is our boundary for the upper bound.
2. If the user enters a filename length in [50, inf) , it will give a warning and the user can’t enter a filename greater or equal to 50 characters. We test with a filename length of 100.
3. If the user enters a filename length in (0, 50), the upload should be successful. We test with length 30.

Case 1: without strategy name



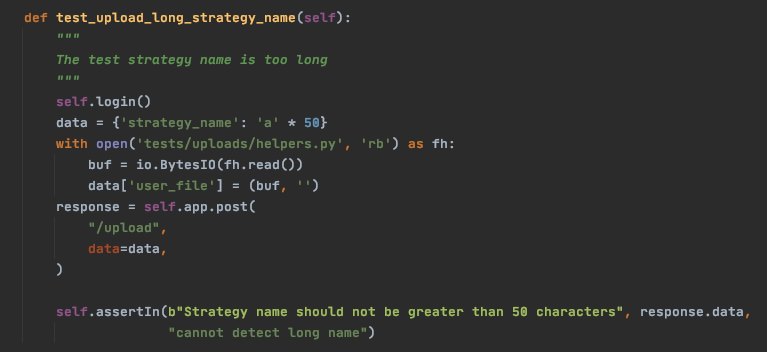
Case2.1 (length is 100):



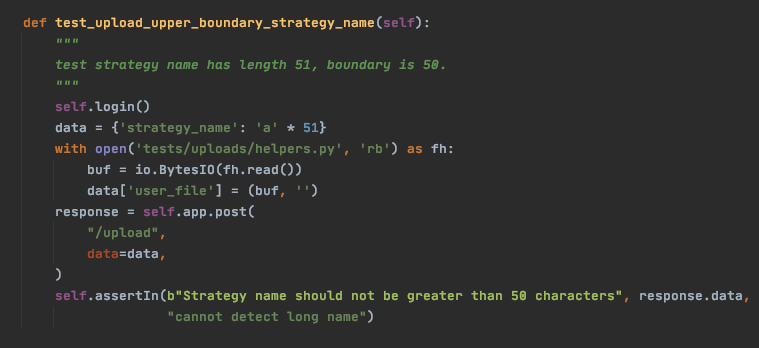
Case 1 (length is 50):

Case 3 (length = 30) partition range: 0 < length < 50



Upper Boundary Tests****

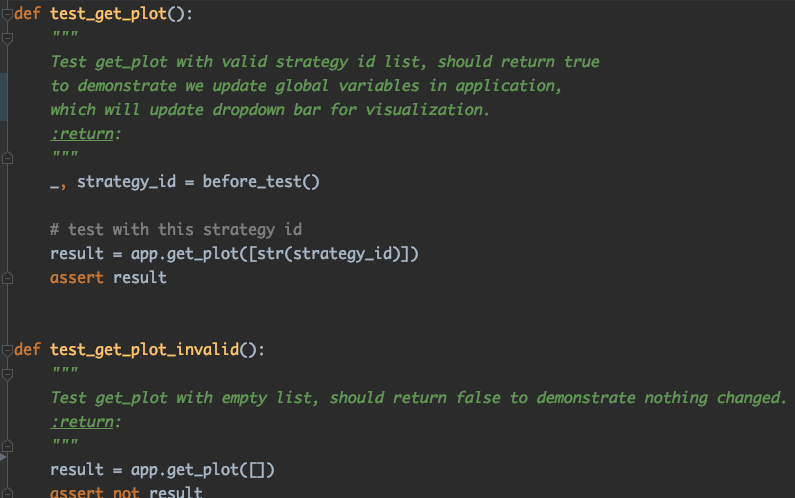
****

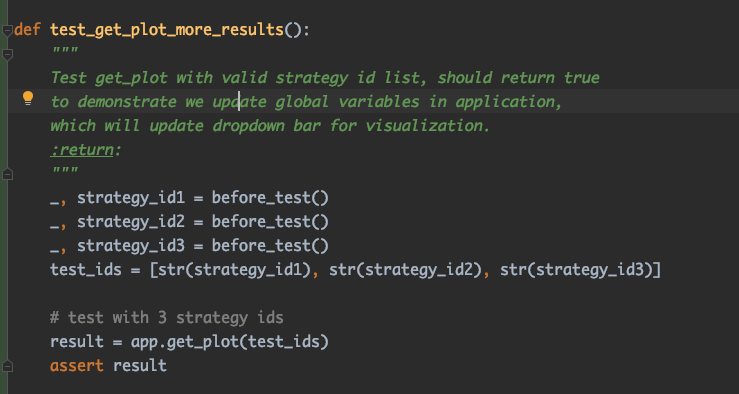
****

Names of test case: Backtesting Visualization

For the unit testing on the results page, we consider two boundaries: valid number of strategy\_ids = 0 and valid number of strategy\_ids > 0. Since we don’t have an upper limit for how many files a user can upload and get backtest results right now, we don’t have an upper boundary condition.

1. If the user doesn’t have any valid backtesting results, then we do nothing on the visualization page. len(strategy\_ids) = 0 is a boundary condition here.
2. If the user has any valid backtesting results, then we update the options in the dropdown bar so that the user can select which result to visualize. We test with only one strategy id, which is a boundary condition.
3. If the user has any valid backtesting results, then we update the options in the dropdown bar so that the user can select which result to visualize. We test with a list of 3 strategy ids, which partition case in (0, inf)

Case 1 & Cas2: ****

Case 3: 

Part 3. Branch Coverage

*Measure the branch coverage achieved by your automated test suite. This requires using a coverage tool appropriate for your programming language and platform. Branch coverage should strive to achieve 100%, but may not reach 100%. Add more test cases until you reach at least 90%. Each additional test case should try to force a particular branch that was not previously covered. Tell us what branch coverage you finally did achieve. If the coverage tool reports less than 100% (that is, between 90% and 99%), discuss one example of a branch that your test cases did not cover and explain why it is difficult to test this branch. If you were unable to reach 90%, explain why not.*

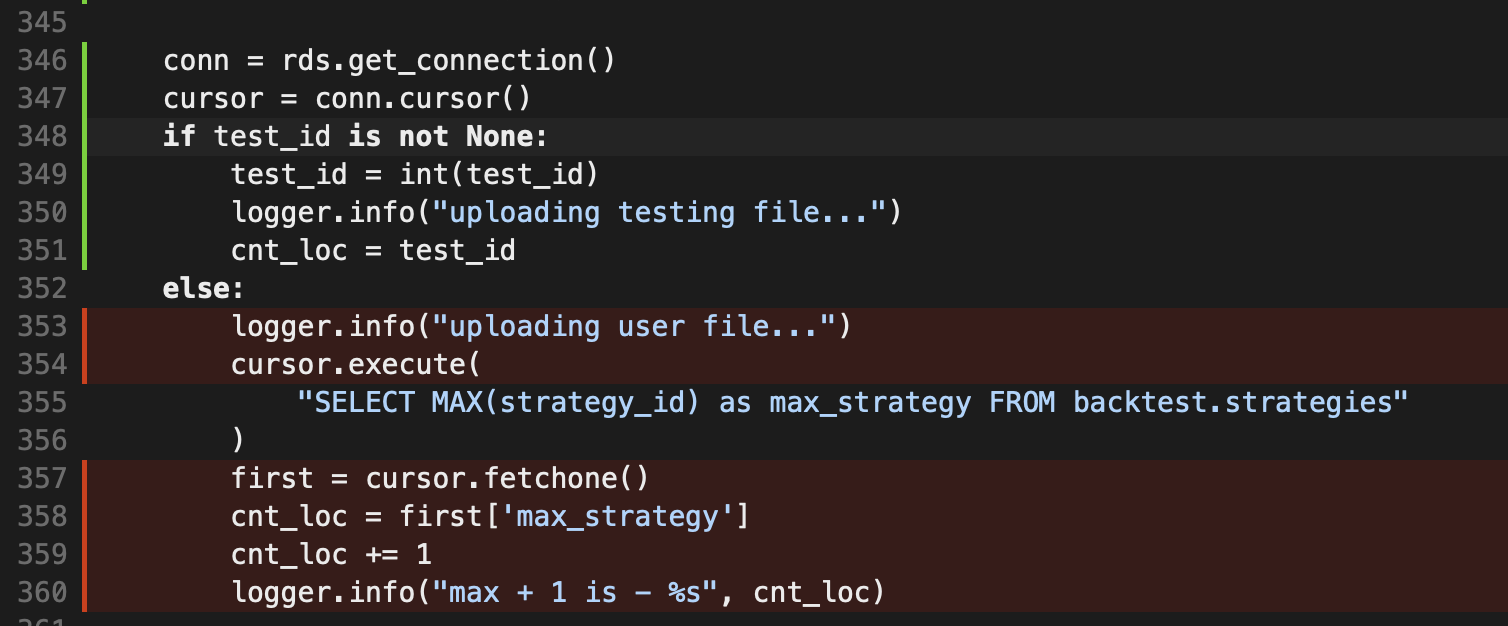
*Include the link to the folder in your repository that contains the coverage test reports. Note this means you need to configure your coverage tool to produce reports that can be saved as files in your repository.*

We have an overall coverage of 92%.

The reason why we cannot cover some branches/code snippets is because

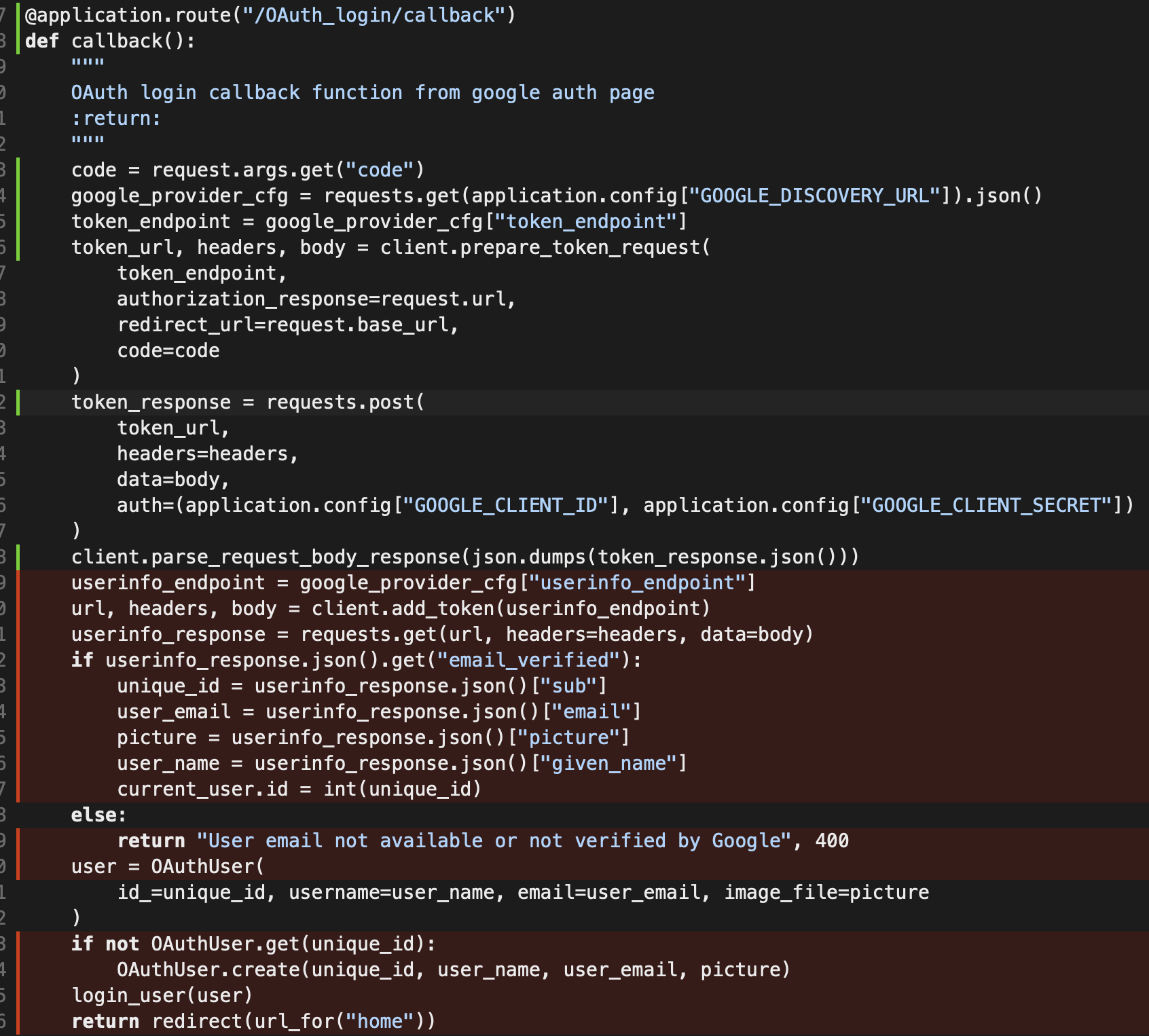
1. Some branches requires authentication token that we cannot mock, for example, Google API
2. We separate the test branch and production branch for some use cases for a deterministic testing pattern.
3. For the forget password feature, we would need to set-up email access to retrieve the reset password token. Which is very convoluted so we assume Google API is robust in delivering the password reset token.

Example for division of test branch and production



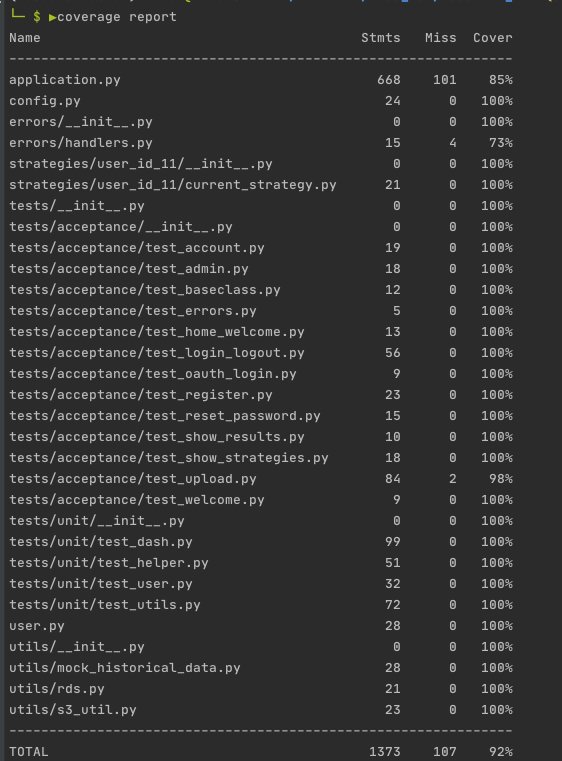
If branch leads to test part, which can be tested, but else goes to production, so it is not testable

Example for security token:



To reach the red (uncovered) part, we need to have the authentication token, which is not possible

The image to our coverage report is :



The link to our detail reports is: <https://github.com/gzhami/research_lab/blob/dev/submissions/coverage_report.txt>

Part 4. Continuous Integration

*As part of the release effort, you need to institute continuous integration for your codebase. The idea is to integrate automated build and test with your version control repository using Travis CI or a similar tool, so that build and test is automatically initiated whenever there is a new commit to the main branch of your github repository. Make sure to ask for help far in advance of the assignment deadline if you have trouble getting CI to work.*

*Include links to the files in your repository that configure the CI tool(s) and a folder in your repository that includes the CI reports. Note this means you need to configure your CI tool to produce reports that can be saved as files in your repository.*

*5 points for CI configuration (e.g., .travis.yml file), 0 points if not found.*

*5 points for a folder in the github report containing CI reports, or any other mechanism allowing you (the grader) to access the CI reports, that demonstrates the CI indeed works and is being used. This should be 0 if CI does not appear to be working.*

We use Github Action instead of Travis CI because Github Action is Github-native and easy to use.

We use Git Action to test if everything compiles on Linux. In addition, we deploy our code using AWS CodePipeline and AWS Elastic Beanstalk. It gets automatically deployed every time we push to dev/master. For details, please see screenshot below.

