Improve backend test coverage and upgrade it to support Python 3

Basic Information

- Name: Rishav Chakraborty
- Institute: University Institute of Engineering and Technology, Panjab University
- Major: Computer Science and Engineering (Second Year)

Why am I interested in working with Oppia?

I believe education is the right to everyone but many can't afford it especially in the developing and underdeveloped countries. Oppia's mission is to "provide high quality education to those who lack access to it." I wish to contribute to it with all my efforts so that students all across the world has access to high-quality scalable online lessons.

I used many online learning platforms but found Oppia to be the most unique. Its explorations and one-on-one learner/tutor conversation makes the learner focus and participate actively. This system also beats the education system followed in schools and colleges.

Furthermore, I find the Oppia community to be super friendly. They helped me at every issue I faced and are always ready to guide me. Contributing to Oppia has enhanced my technical knowledge and also taught me the importance of working and coordinating in a team. It had been a great learning experience for me so far with Oppia. I would continue to contribute to Oppia even after the GSoC period ends.

What interests me about this project? Why is it worth doing?

The project aims to improve backend test coverage to 100%, and then migrate the backend codebase to be simultaneously compatible with both Python 2 and Python 3. It also includes converting the bash scripts to Python. This project is really important for the following reasons:

I. 100% test coverage is needed to guard against regressions in the future which can happen when developers rewrite code and break things because there are no tests for that part of the code.
II. Converting the bash scripts would make Oppia less OS-dependent and would reduce the number of scripting languages that Oppia uses.

III. Python3 is the future of Python and Python2 will not be maintained past 2020. Hence its very important to port the existing code to Python3.

This project requires proficiency in bash and Python scripting and also a knowledge of working of the backend files. Thus, I would get to learn a lot of things while working on this project. It would be great fun to work on this project. I am also deeply interested in networking and from the conversion of the scripts, I would learn a lot more about it. Completing the project would also make me more proficient in Python and back-end testing.

Prior Experience

I have been working with Python for the last two years. I am a backend developer and work primarily with Google App Engine and its framework webapp2 .I also have experience with Django. I have been involved in web development since the past year.

Some of my projects include:

- ❏ At Rajasthan Hackathon 5.0, I led a team of 4 members where we made a system for automatic accident detection using various sensors and alarming the nearest hospital of the location using Google App Engine.
- ❏ I worked in a college project to make an adaptive intelligent tutoring system for students based on Bloom's taxonomy. My task was to create the whole website, the database and also to write the functions which would automatically check the student's answers to questions. I used Django and Stanford NLP to do my work.
- ❏ I made a blog for adding entries and posts for users as a summer project for college. The website is live here.
- ❏ I made other small front-end projects like a website for adding image filters and others for drawing different patterns using pure JavaScript.
- ❏ I also contributed to Sympy(a symbolic mathematics library for Python) in the past year and attended some technical conferences like Pycon India, etc.

Source code for all my projects can be found at my GitHub profile .

Links to PRs and issues to Oppia

Some of my PRs are:

- Lint check to ensure that each Angular file contains exactly one component: #6202

- Lint checks to run only for the files changed by the user: #6033
- Tests for memcache services: #5355
- Changed startup script to open browser after starting server: #5431
- Reduce code redundancy in backend testing: #5838

Here is the complete set of PRs opened by me.

I also created a total of 10 issues in Oppia till date. They can be seen here.

Project plan and implementation strategy

The project is divided into 3 major parts. Below are the detailed explanations of each of these.

1. Milestone 1

1.1. Improve test coverage of 85% of the backend code

For porting the code to be compatible with both Python2 and Python3, we need to ensure first that it has a very good coverage. It is because to be sure in our test suite that any failures that appear after having tools rewrite the code are actual bugs in the tools and not in our code. The final aim is to merge a PR which brings backend coverage to 100% by end of Milestone 2.

The files and their coverage which need to be covered can be seen in Codecov's dashboard.

The following is the general strategy in writing the tests:

1. setup - this is where we prepare any inputs/environment needed for the test.
2. baseline verification - check the values without performing any action. This can be done by `self.get_json(url)` for files in **core/controllers** and by directly calling the function to be tested for other files.
3. action - perform the action or function call that leads to the expected change. This can be done by `self.post_json(url)` for files in **core/controllers** and by directly calling the function which will update its value for other files.
4. endline verification - check that the values in the baseline verification have changed accordingly. This can be checked again by

`self.get_json(url)` for files in **core/controllers** and by directly calling the function to be tested for other files.

Missing tests which are common among files are where:

- Exception is raised

  The tests for catching exception can be written as:

```
with self.assertRaises(Exception):
                response = self.get_json('/mock/%s' %
self.exploration_id)
```

- Code depends on constants in `constants.js`:

```
with self.swap(constants, 'ENABLE_NEW_STRUCTURE_EDITORS',
True):
                response = self.get_json(url)
```

- Some functions are not used anywhere in the codebase and hence leads to decrease in coverage. Those functions would be removed. I will find these functions as and when I will come across these when adding the backend tests. For example when writing the tests for **core/controllers/creator_dashboard.py** I came across a function `_get_intro_card_color` which I found was not used anywhere in the codebase so I removed it.
- When writing the backend tests, the controllers and the functions to be tested would also include strings with unicode characters. These functions would take unicode strings(for example Hindi etc) as parameters to check if we get the desired behaviour. This is important as the behaviour of str/unicode would change when the porting of Python2 to Python3 would occur(further details can been seen in milestone 2.2) and it is important to ensure that the functions still work and are not broken. For example in **core/domain/fs_domain_test.py:**

```
self.fs = fs_domain.AbstractFileSystem(
            fs_domain.ExplorationFileSystem('exploration/eid'))

def test_get_and_save(self):
        self.fs.commit(self.user_id, 'abc.png',
'file_contents')
        self.assertEqual(self.fs.get('abc.png'),
'file_contents')
        self.fs.commit(self.user_id, 'abc.png', 'Л о р е м')
        self.assertEqual(self.fs.get('abc.png'),
```

```
utils.convert_to_str('Л о р е м'))
```

In the above function the behaviour for `fs.commit` should also be working for the unicode string 'Лорем' just like it works for the string ‘file_contents’. This way the functions would include tests for unicode strings.

The list of files that need to be covered is a little big and thus I hope to start making PRs which will add tests for some of the files soon before the community bonding period begins.

2. Milestone 2

2.1. Improve test coverage of the remaining backend code

Since the list of files to be covered for backend testing is quite big I will add tests for the remaining 15% of the backend code in this milestone.

2.2. Port to Python3 from Python2.

This is the main part of the project. I will use **Futurize** for making the codebase compatible with Python3. The Python 3 version which would be ported to is Python 3.6 as Python 3.7 is only available on Xenial build images in Travis and the Travis yaml file uses trusty In order to install Chrome stable. Python 3.6 images are also available on CircleCI. I prefer **Futurize** over other tools like **Modernize** as Futurize does its best to make Python3 idioms and practices exist in Python2, for example backporting the bytes type from Python 3 to have the semantic parity between the major versions of Python. Also Futurize has an excellent documentation to help the conversion. Below is a tabular comparison between **Futurize** and other alternatives:

Futurize	Modernize
Futurize converts either Python2 or Python3 code into (almost) standard Python 3 code, with the `future` module as a run-time dependency.	Python-modernize converts Python2 code into a common subset of Python 2 and 3, with the `six` module as a run-time dependency.

The two steps in the conversion process are:

- Run the first stage of the conversion process with:

  ```
  futurize --stage1 -a core/**/*.py
  ```

  This applies fixes that modernize Python 2 code without changing the effect of the code. This will not introduce any non-trivial bugs into the code as it will modernize Python2 code only with no compatibility with Python3. We are going to ensure this by:

  - ➔ running our backend tests on the entire codebase. And since by then the coverage would be 100%, we can be sure that it didn't introduce any bugs. Even if in the “worst case scenario” some bugs are introduced in the codebase, the backend tests would fail and I will fix those bugs immediately.
  - ➔ making sure that no one else introduces non-portable py2-to-py3 code which would be handled by the lint checks.
  - ➔ manually testing certain core user journeys to make sure nothing is broken. This can be done by working closely with the QA team.

  The changes are those that bring the Python code up-to-date without breaking Python2 compatibility. The resulting code will be modern Python 2.7-compatible code plus __future__ imports which includes:

  ```
  from __future__ import absolute_import
  from __future__ import division
  from __future__ import print_function
  ```

- Run stage 2 of the conversion process with:

  ```
  futurize --stage2 -a core/**/*.py
  ```

  This stage adds a dependency on the future package. The goal for stage 2 is to make further mostly safe changes to the Python 2 code to use Python3 style code that then still runs on Python2 with the help of the appropriate builtins and utilities in future. We can't guarantee that the change would be fully safe as the porting will introduce some bugs/feature-changes some of which are discussed below. I will have to manually fix the bugs that comes on porting. I will also add new backend tests for any such bugs that are found.

The **-a** flag denotes **--all-imports** which add all future imports to each file. This is required as after porting developers need not worry about which future import to add to a new file that has been created.

- After stage 2, we need to convert the binary data to the unicode wherever needed. In Python2 we could use the str type for both text and binary data. Under Python 3, binary files and text files are clearly distinct and mutually incompatible. Hence to represent strings in Python2 we will be using `unicode` while representing them in Python3 we will be using `str`. To represent binary data we will be using `bytes` for both Python 2 and 3.

  By passing the **--unicode-literals** flag to the futurize script, all string literals that were not explicitly marked up as b'' will mean text (Python3 str or Python2 unicode). For example:

  ```
  text_1 = u'Л о р е м'
  text_2 = 'example_text_1'
  text_3 = b'example_text_2'
  ```

  After running the futurize script with the above flag, `text_1` and `text_2` both would be of type ‘unicode’ in Python2 and type ‘str’ in Python3 while `text_3` would be of type ‘bytes’ in both the Python versions.

  As of now, I have not fully migrated the codebase but I tested it by porting the files in the `scripts/` folder and some files in `core/controllers/` and `core/domain/`. The changes required to convert binary to str are mostly trivial and I will fix them as and when they occur. Some examples of the changes required are:

  - ➔ After the conversion of the **pre_commit_linter,** the file-paths are needed to be converted to str form.

    ```
    all_filepaths = [filename.decode('utf-8') for
    filename in all_filepaths]
    ```

  - ➔ To convert some content to str form, we would have to replace the **str()** function with **utils.convert_to_str().** This is because all the strings in Python2 would basically be of type `unicode`. For example `str(content)` would be replaced by `utils.convert_to_str(content)`.

- Make changes to open files using **io.open()** instead of just **open()** as the **io** module is consistent from Python2 to Python3 while the built-in **open()** function is not (in Python3 it's actually io.open()). In Python2, the built-in **open** and **io.open** were different (**io.open** was newer and supported more things like explicit encoding, etc). In Python3, **open** and **io.open** are the same thing (they got rid of the old built-in **open**). Since **open** is different in Python 2 and 3, we will have to use **io.open** which is consistent across the two versions.

  The built-in **open** function in Python2 looks like:

  ```
  open(name[, mode[, buffering]])
  ```

  While **io.open** looks like:

  ```
  io.open(file, mode='r', buffering=-1, encoding=None,
  errors=None, newline=None, closefd=True)
  ```

These changes would be part of the **python_utils.py** file so that we could choose later which one will be easier to migrate. The file would contain both the functions, i.e **open** and **io.open**. Hence our code would look like:

```
with python_utils.open(file) as f:
    # do stuff to f
```

- After the above steps, we will have code that has to choose what to do based on what version of Python is running. The best way to do this is with feature detection of whether the version of Python running under supports what we need.

  For example, in **scripts/pre_commit_linter** the code for redirecting stdout would be as follows:

  ```
  try:
      from StringIO import StringIO
  except ImportError:
      from io import StringIO
  _TARGET_STDOUT = StringIO()
  ```

  This is because:

- The **StringIO** module is gone in Python3 and hence we need **io.StringIO**
- Redirecting stdout with **io.StringIO** in Python2 does not work as it is an in-memory stream for unicode text.

Another example for this is in **scripts.docstrings_checker.get_args_list_from_function_definition,** we will have to use the following code:

```
try:
                return [a.arg for a in
function_node.args.args if a.arg not in
                                              args_to_ignore]
                           except AttributeError:
                               return [a.id for a in
function_node.args.args if a.id not in
                                            args_to_ignore]
```

The above change is required as to get the arguments name in a function in Python2 we need to use **function_node.args.args.id** but in Python3 we need to use **function_node.args.args.arg**

This sort of feature detection would be kept in a separate file called **python_utils.py** so that we don't have this sort of construct in any other file. We don't want it to be a pattern throughout the rest of the codebase. The functions in the file can go like this:

```
def import_string_io():
    try:
        from StringIO import StringIO
    except ImportError:
        from io import StringIO
    return StringIO()

def get_args_of_function(function_node):
    try:
              return [a.arg for a in function_node.args.args if
a.arg not in
```

```
                                    args_to_ignore]
                except AttributeError:
                    return [a.id for a in
function_node.args.args if a.id not in
                                    args_to_ignore]
```

The other files can directly call these functions whenever required.

I will add the other feature detections in this file as and when they occur in the codebase.

- I have checked the https://caniusepython3.com website to detect which dependencies also need to be ported. As much as I can check for now we don't have to migrate our third_party libraries but if the need comes I will migrate them. There may be some libraries that needs to be ported and the caniusepython3 does not name all the libraries that needs to be addressed(i.e it can't tell whether its a blocking dependency or not). This is the reason I have allotted quite a time for this milestone. But we do need to install our pip packages with pip3 in addition to pip. For that in **install_prerequisites.sh** I will add `sudo apt-get install python3-pip` and in **install_third_party.sh** I will add another line in function **pip_install** to install the packages in Python3 environment as well. The above function would become:

  ```
  function pip_install {
    # Attempt standard pip install, or pass in --system if
  the local environment requires it.
    # See https://github.com/pypa/pip/issues/3826 for
  context on when this situation may occur.
    pip install "$@" || pip install --system "$@"
    pip3 install "$@" || pip3 install --system "$@"
  }
  ```

  For mac developers, I will update the wiki page here to include instructions on installing pip3. The command `brew install python3` would install Python3 and also install pip3 automatically.

  Some libraries that are not compatible, for example **beautifulsoup4** can be made compatible by adding the required download URL in **manifest.json** file. There are two URLs for this library(one for Python2 and the other for Python3).

3. Milestone 3

3.1. Put lint checks to ensure that the backend code always remains compatible with both Python2 and Python3

- Once we have fully translated our code to be compatible with Python3, we must make sure our code doesn't regress and stop working under Python2.

  For this we can use **Pylint's --py3k** flag to lint our code to receive warnings when the code begins to deviate from Python3 compatibility. This also prevents from having to run Futurize over our code regularly to catch compatibility regressions. It checks for:

  ➔ Print statements
  ➔ Division without __future__ statement (old-division)
  ➔ Outdated imports, i.e built-in modules that are no longer present in Python3.

  I would add this flag in **pre_commit_linter** where pylint is run:

```
pylinter = lint.Run(
                ['--py3k'] + current_files_to_lint +
[config_pylint],
                exit=False).linter
            # These lines invoke Pycodestyle and print
its output
            # to the target stdout.
            style_guide =
pycodestyle.StyleGuide(config_file=config_pycodestyle)
            pycodestyle_report = style_guide.check_files(
                paths=current_files_to_lint)
```

  Also to help with staying compatible, any new files that will be created should have at least the following block of code at the top of it and there would be a lint check to enforce this:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals
```

- After creating the **python_utils.py** file we would want other developers to strictly use the functions defined there. For example, we would want to enforce people to not use StringIO and instead use the custom function defined in the above file. I will create a lint check to enforce this. This lint check would detect patterns when a certain code is used which should rather be called from the **python_utils.py** file. For example:

```
PYTHON_UTILS_PATTERNS = {
    'Import StringIO': 'Use import
python_utils.import_string_io()'
    'get_args_list_from_function_definition()': 'use
python.utils.get_args_of_function()'
}
all_filepaths = [
            filepath for filepath in self.all_filepaths if
not (
                filepath.endswith('pre_commit_linter.py')
or
                any(
                    fnmatch.fnmatch(filepath, pattern)
                    for pattern in EXCLUDED_PATHS)
                )]
        with _redirect_stdout(_TARGET_STDOUT):
            for filepath in all_filepaths:
                    file_content =
FileCache.read(filepath)
        for pattern in PYTHON_UTILS_PATTERNS:
                    if pattern in file_content:
                        failed = True
                        print('%s --> %s' % (

filepath,PYTHON_UTILS_PATTERNS[pattern]))
```

These lint checks together with the file **python_utils** created in milestone 2 would be sufficient for checking compatibility with Python 2 and 3.

3.2. Convert all bash scripts to python

The bash scripts would be converted to python so that Oppia is just using one scripting language and so that the scripts are less platform-dependent. This part would include heavy use of the shutil and the os library. Some scripts like the **clean.sh, install_chrome_on_travis.sh** and **install_prerequisites.sh** are fairly easy to convert. These include basic file operations like **rm, mkdir,** installing libraries, etc.

Following is the table for comparing the basic syntax between bash and python:-

BASH	PYTHON
rm -rf	shutil.rmtree(path)
echo	print
if [! -f dir_path)	os.path.isdir(dir_path)
mkdir	os.makedirs()
dpkg	DebPackage()

- For the networking and the server stuff, socket is a great choice. Its a low level networking interface. For example, to see if a server is running in a particular port we can use the following function:

```
def is_port_in_use(port):
        import socket
        with socket.socket(socket.AF_INET,
socket.SOCK_STREAM) as s:
            return s.connect_ex(('localhost', port)) == 0
```

I tested in my local machine and it works.

For other commands that are specific to bash like starting Oppia requires **subprocess** to execute it.

```
cmd = "python GOOGLE_APP_ENGINE_HOME/dev_appserver.py
CLEAR_DATASTORE_ARG ENABLE_CONSOLE_ARG --admin_host
0.0.0.0 --admin_port 8000 --host 0.0.0.0 --port 8181
--skip_sdk_update_check true app.yaml)&"
task = subprocess.Popen(cmd.split(),
stdout=subprocess.PIPE,
                                stderr=subprocess.PIPE)
out, err = task.communicate()
```

- I will also integrate the test scripts with their respective Python files. For example the **run_backend_tests.sh** code can be included in the **backend_tests.py** file by simple syntax conversion and running the **coverage reports** by **subprocess.** Only the **TestingTaskSpec** class needs to be modified as below to report the coverage.

```
import setup
import setup_gae

TOOLS_DIR = os.path.join(
    os.getcwd(), '..', 'oppia_tools')
THIRD_PARTY_DIR = os.path.join(
    os.getcwd(), 'third_party')
class TestingTaskSpec(object):
    """Executes a set of tests given a test class name."""

    def __init__(self, test_target,
generate_coverage_report):
        self.test_target = test_target
        self.generate_coverage_report =
generate_coverage_report

    def run(self):
        """Runs all tests corresponding to the given test
target."""
        test_target_flag = '--test_target=%s' %
self.test_target
```

```
        if self.generate_coverage_report:
            exc_list = [
                'python', COVERAGE_PATH, 'run', '-p',
TEST_RUNNER_PATH,
                test_target_flag]
            e = run_shell_cmd(exc_list)
            exc_list1 = 'python %s report
--omit=%s/*,%s/*,/usr/share/ --show-missing' %
(COVERAGE_PATH, TOOLS_DIR, THIRD_PARTY_DIR)
            e1 = run_shell_cmd(exc_list1.split())
            print e1
        else:
            exc_list = ['python', TEST_RUNNER_PATH,
test_target_flag]
            e = run_shell_cmd(exc_list)

        return e
```

- After the bash scripts are converted to Python, it needs to have tests of its own so it does not decrease coverage. For testing if the scripts are downloading/removing libraries, I would create a mock class that will inherit the actual class and ensure that the calls to the download/remove in the mock class occur. For example:

  install_third_party.py:

```
class Install_libraries():
    def install_via_pip(self):
        pip_install(pylint==1.9.3)
        pip_install(browsermob-proxy==0.7.1)
```

  install_third_party_test.py:

```
import install_third_party
class Mock_install_libraries(Install_libraries):
    def install_via_pip(self):
        with self.assertRaises(Exception):
            pip_install(test_package)
```

  The socket connection can be tested as:

```
import start
is_port_in_use = start.is_port_in_use(8181)
self.assertTrue(is_port_in_use)
start.close_port()
```

We can check if a library is installed or not as follows:

```
response = requests.get(url_to_library)
self.assertTrue(response.ok)
```

To check if a library is installed properly we can use the **mock** library. It mocks out HTTP code.

The **mock** library is a part of the unittest module in Python 3 while it needs to be separately installed as a pip package in Python 2. This would be taken care by the **python_utils.py** file.

For example mocking a function that deletes a folder would be:

```
import shutil
import mock
def del_dir(test_path):
   shutil.rmtree(test_path)

response_mock = mock.Mock()
response_mock.del_dir('test_path/')
response_mock.del_dir.assert_called_once_with()
```

- The config files of CircleCI and Travis would be edited to run the new Python scripts instead of the old bash ones. For example to run the front-end tests on CircleCI, the following code snippet would be used:

```
frontend_tests:
    <<: *job_defaults
    steps:
      - checkout
      - run: date +%F > date
      - restore_cache:
          <<: *restore_cache
      - run:
          name: Run frontend tests
```

```
          command: |
            python scripts/run_frontend_tests.py
--run-minified-tests=true
      - run:
          name: Generate frontend coverage report
          command: |
            sudo pip install codecov
            codecov --file
../karma_coverage_reports/coverage-final.json
          when: on_success
```

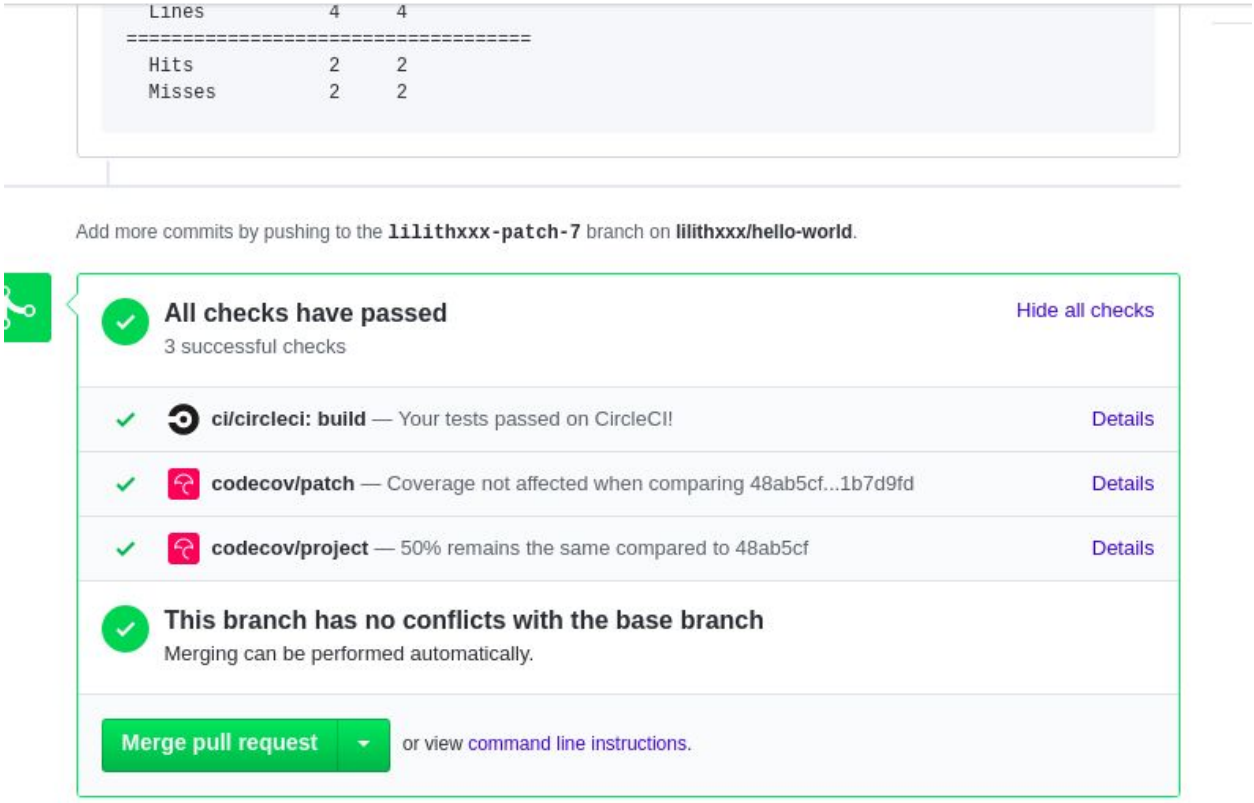
Timeline

Pre-GSoC Period

1. Check to ensure that untested code does not get into develop branch

There should be checks to ensure that code which would decrease coverage should not make into the develop branch. For that I suggest we use Codecov's commit status. It measures overall project coverage and compares it against the base branch. It can be used to automatically detect if the coverage is reduced.

I already implemented the above commit-status in my local repo. Below is the screenshot:



Oppia already uses Codecov and hence needs just a yaml configuration file to enable the commit-status. The yaml file roughly consists of:

```
coverage:
  status:
    project:
      default:
        # basic
        target: auto
        threshold: null
        base: auto
```

- The target field would be updated to **100%** after milestone 2. This would ensure that our backend coverage is 100%.
- The codecov/patch status(as shown in the above screenshot) only measures lines adjusted in the pull request. This status provides an indication of how well

the pull request is tested. For example, in a PR we add the following lines denoted by '+':

```
def divide(x, y):
+    if y <= 0:
+        raise ValueError("y must be greater than 0")
    return x * y
```

The resulting codecov/patch status of this commit would be 0% covered because no tests are created for this method. Even though the project coverage is 90% (approximate coverage of the entire Oppia codebase), this patch status will only measure lines added.

Another commit adds the following lines:

```
+ def test_divide_by_1(self):
+     assert divide(10, 1) == 10
```

Running the tests will result in a patch coverage of 50% covered because we have not yet tested the behavior of dividing by zero. Let's add another test by adding another commit:

```
 def test_divide_by_1(self):
     assert divide(10, 1) == 10

+ def test_divide_by_zero(self):
+     with self.assertRaises(ValueError)
+         divide(1, 0)
```

Now Codecov will report a codecov/patch status of 100% covered for this full pull request. This indicates that the pull request adjusted code is properly executed by tests.

- The other commit status measures overall project coverage with the newly created PR and compares it against `develop`. This tests if the PR increases or decreases the overall project coverage.

  Hence two commit status will be added to check coverage -- one for the whole codebase and one for the PR created.

  That's it! The configuration file for the CircleCI would automatically call the Codecov's configuration file resulting in the display of commit-status. The

`codecov` command in the CircleCI yaml file would do the above task and thus there is no need to make changes in the CircleCI yaml file.

2. Write a few backend tests:

I am planning to write the backend tests for the following files before the community bonding period starts as these files are fairly small and only 5-10 lines of code remains in each file to be tested:

- core/controllers/creator_dashboard.py
- core/controllers/moderator.py
- core/controllers/practice_sessions.py
- core/controllers/question_editor.py
- core/controllers/recent_commits.py
- core/controllers/resources.py
- core/controllers/topics_and_skills_dashboard.py
- core/domain/activity_jobs_one_off.py
- core/domain/classifier_services.py
- core/domain/config_domain.py
- core/domain/config_services.py
- core/domain/feedback_jobs_continuous.py
- core/domain/fs_services.py
- core/domain/html_cleaner.py
- core/domain/obj_services.py
- core/domain/param_domain.py
- core/domain/question_domain.py
- core/domain/question_jobs_one_off.py
- core/domain/rating_services.py
- core/domain/recommendations_jobs_one_off.py
- core/domain/search_services.py
- core/domain/skill_jobs_one_off.py

- core/domain/stats_jobs_continuous.py
- core/domain/story_jobs_one_off.py
- core/domain/user_query_jobs_one_off.py
- core/domain/value_generators_domain.py
- core/domain/visualization_registry.py

The files are divided into Easy Medium and Hard on the basis of difficulty. The difficulty rating is based on the following table:

Easy	Medium	Hard
Less LOC(5 - 20)	More LOC than Easy ones (20 - 80)	More LOC than medium ones(80 - 180)
Mainly includes writing tests for catching exceptions	Includes: • Writing tests for catching exceptions • Writing tests for handlers which are small in size and does not have much branching statements and hence are less complicated	Includes: • Writing tests for catching exceptions • Writing tests for handlers which are bigger in size and has more branching statements and hence are more complicated.

Expected date for submission of PR	Expected merge date	TASKS(Difficulty)
• May 8 • May 9 • May 11	• May 9 • May 11 • May 13	Write backend tests for files in **core/controllers**: • base.py(Medium) • profile.py(Medium) • editor.py(Medium)

• May 13 • May 14 • May 15 • May 16 • May 17 • May 18 • May 19 • May 21 • May 23 • May 25	• May 14 • May 15 • May 16 • May 17 • May 18 • May 19 • May 21 • May 23 • May 25 • May 27	• email_dashboard.py(Medium) • feedback.py(Medium) • learner_dashboard.py(Medium) • library.py(Medium) • skill_editor.py(Easy) and story_editor.py(Easy) • suggestion.py(Easy) and topic_editor.py(Easy) • collection_editor.py(Easy), collection_viewer.py(Easy), and translator.py(Easy) • acl_decorators.py(Easy) and pages.py(Easy) • Files in core/platform(Easy) • Files in core/storage(Easy) Also my college final exams would be going on.
• May 28 • May 29 • May 30 • May 31 • June 1 • June 2 • June 3 • June 4 • June 5 • June 6 • June 7 • June 8 • June 9 • June 10 • June 11	• May 29 • May 30 • May 31 • June 1 • June 2 • June 3 • June 4 • June 5 • June 6 • June 7 • June 8 • June 9 • June 10 • June 11 • June 12	Complete the remaining backend tests in **core/domain**: • collection_domain.py(Hard) • exp_domain.py(Hard) • topic_services.py(Hard) • collection_services.py and its test file(Medium) • learner_progress_services.py(Medium) • user_jobs_one_off.py(Medium) • exp_services.py(Medium) • feedback_services.py(Medium) • html_validation_service.py(Medium) • question_services.py(Medium) • topic_domain.py(Medium) • skill_domain.py(Medium) • skill_services.py(Medium) • state_domain.py(Medium) • stats_domain.py(Medium)

• June 12 • June 13 • June 14 • June 15 • June 16 • June 17 • June 19 • June 20 • June 21 • June 22	• June 13 • June 14 • June 15 • June 16 • June 17 • June 19 • June 20 • June 21 • June 22 • June 23	• exp_jobs_one_off.py(Medium) • stats_services.py(Medium) • story_domain.py(Medium) • story_services.py(Medium) • suggestion_services.py(Medium) • event_services.py(Easy) and classifier_domain.py(Easy) • subtopic_page_domain.py(Easy), subtopic_page_services.py(Easy) • rights_manager.py(Easy), suggestion_registry.py(Easy) • summary_services.py(Easy) • topic_jobs_one_off.py(Easy), user_jobs_continuous.py(Easy)
June 24	June 28	Buffer period for first milestone
• June 29 • July 1 • July 3 • July 5 • July 6 • July 7 • July 8 • July 10 • July 11	• July 1 • July 3 • July 5 • July 6 • July 7 • July 8 • July 9 • July 11 • July 12	Write the remaining backend tests in: • scripts/build.py(Hard) • scripts/pylint_extensions.py(Hard) • core/domain/stats_jobs_one_off.py(Hard) • core/controllers/reader.py(Hard) • core/controllers/admin.py(Hard) • scripts/docstrings_checker.py(Medium) • core/tests/test_utils.py(Medium) • core/jobs.py(Medium) • core/domain/user_services.py(Easy)
July 13	July 21	All of the following would be done in a single PR to make all the backend tests pass:- • Port the code to Python3 and convert the binary data to the unicode and change the docstrings to use the correct datatype. • Create **python_utils.py** to use features different in Python2 and Python3 • Make further code changes that are necessary • Migrate libraries that would create dependency problems.

July 22	July 26	Buffer period for second milestone.
July 27	July 29	Put lint checks to ensure that the backend code always remains compatible with both Python2 and Python3.
● July 30 ● Aug 2	● Aug 1 ● Aug 4	Convert the following bash scripts to Python(The following order is required so that we don't have importing/sourcing problems) and write its test file:- ● setup.sh ● clean.sh, create_expression_parser.sh,install_chrome_on_travis.sh, install_frontend_tests_dependencies.sh
● Aug 5	● Aug 7	● vagrant_lock.sh, install_prerequisites.sh
● Aug 8 ● Aug 11 ● Aug 14	● Aug 10 ● Aug 13 ● Aug 16	● Install_third_party.sh ● Setup_gae.sh, start.sh ● run_backend_tests.sh, run_frontend_tests.sh
● Aug 17	● Aug 19	● run_e2e_tests.sh, run_performance_tests.sh
● Aug 20	● Aug 21	● run_presubmit_checks.sh, run_tests.sh
August 22	August 26	Buffer period for third milestone. Create a list of remaining steps that need to be taken for a final migration to python3

Which timezone(s) will I primarily be in during the summer?

I will be in India throughout the summer (Timezone: UTC+05:30)

How much time will you be able to commit to this project?

Throughout May I will be having my college final exams and hence I can spend around 20-25 hours per week.

In June and July, I will be having summer holidays and can spend around 55-60 hours per week.

In August, my classes begin and hence the time spent during weekdays would be lesser (around 4-5 hours) while in the weekends I can spend more time (about 8-10 hours, to make up for any pending tasks for the week).

What jobs, summer classes, and other obligations might you need to work around?

I have no other commitments during the summer. I have no other obligations till July end. In August I will have to get back to college, so the number of hours per week may be slightly lower. I will try to make up during weekends. There might also be two days of travelling, one in the first week of May and the other in the third week of July.

Communication

- Email: annonymousxyz@outlook.com
- Other mail: reshav01@gmail.com
- Github: @lilithxxx

I am comfortable with all modes of communication, be it Gitter or Hangouts and am willing to choose any mode used by the mentors.

I will be in continuous touch with the mentors via email, Gitter or Hangouts. There could be biweekly (or as preferred by the mentors) meetings on Hangouts to discuss about the workflow to be followed ahead.