Otter-Grader Documentation

UCBDS Infrastructure Team

Apr 20, 2020

Contents:

CHAPTER 1

Installation

Otter is a Python package that can be installed using pip:

```
pip install otter-grader
```

1.1 Docker

Otter uses Docker to create containers in which to run the students' submissions. Please make sure that you install Docker and pull our Docker image, which is used to grade the notebooks.

1.1.1 Pull from DockerHub

To pull the image from DockerHub, run `docker pull ucbdsinfra/otter-grader`. If you choose this method, otter will automatically use this image for you.

1.1.2 Download the Dockerfile from GitHub

To install from the GitHub repo, follow the steps below:

1. Clone the GitHub repo
2. `cd` into the `otter-grader/docker` directory
3. Build the Docker image with this command: `docker build . -t YOUR_DESIRED_IMAGE_NAME`

Note: With this setup, you will need to pass in a custom docker image name when using the CLI with the `--image` flag.

CHAPTER 2

Tutorial

This tutorial can help you to verify that you have installed Otter correctly and introduce you to the general Otter workflow. Once you have *installed* Otter, download this zipfile and unzip it into some directory on your machine; I'll unzip it into my home directory, so that I have the following structure:

```
| ~
  | tutorial
    | - demo-fails1.ipynb
    | - demo-fails2.ipynb
    | - demo-fails2Hidden.ipynb
    | - demo-fails3.ipynb
    | - demo-fails3Hidden.ipynb
    | - demo-passesAll.ipynb
    | - meta.json
    | hidden-tests
      | - q1.py
      | - q1H.py
      | - q2.py
      | - q2H.py
      | - q3.py
      | - q3H.py
    | tests
      | - q1.py
      | - q2.py
      | - q3.py
```

`cd` into `tutorial` and let's get started.

The first thing to note is that we have provided a metadata file that maps student identifiers to filenames in `tutorial/meta.json`:

```
[
    {
        "identifier": "passesAll",
        "filename": "demo-passesAll.ipynb"
```

(continues on next page)

(continued from previous page)

```
    },
    {
        "identifier": "fails1",
        "filename": "demo-fails1.ipynb"
    },
    {
        "identifier": "fails2",
        "filename": "demo-fails2.ipynb"
    },
    {
        "identifier": "fails2Hidden",
        "filename": "demo-fails2Hidden.ipynb"
    },
    {
        "identifier": "fails3",
        "filename": "demo-fails3.ipynb"
    },
    {
        "identifier": "fails3Hidden",
        "filename": "demo-fails3Hidden.ipynb"
    }
]
```

The filename and identifier of each notebook indicate which tests should be failing; for exampl, `demo-fails2.ipynb` fails `q2` and `q2H`, and `demo-fails2Hidden.ipynb` fails `q2H`.

Let's now construct a call to `otter` that will grade these notebooks. We know that we have JSON-formatted metadata, so we'll be use the `-j` metadata flag. Our notebooks are in the current working directory, so we won't need to use the `-p` flag. However, we have two test directories: `tests`, which contains public tests, and `hidden-tests`, which contains *all* tests. We want to use the latter, so we'll need to specify `-t hidden-tests` in our call. The notebooks also contain a couple of written questions, and the *filtering* is implemented using HTML comments, so we'll specify the `--html-filter` flag.

Let's run Otter:

```
$ otter -j meta.json -t hidden-tests --html-filter -v
```

(I've added the `-v` flag so that we get verbose output.) After this finishes running, there should be a new file and a new folder in the working directory: `final_grades.csv` should contain the grades for each file, and should look something like this:

```
identifier,file,manual,q1,q1H,q2,q2H,q3,q3H,total,possible
fails2Hidden,demo-fails2Hidden.ipynb,manual_submissions/demo-fails2Hidden.pdf,1.0,2.0,
↪1.0,0.0,1.0,0.0,5.0,8
fails1,demo-fails1.ipynb,manual_submissions/demo-fails1.pdf,1.0,0.0,1.0,1.0,1.0,0.0,4.
↪0,8
fails2,demo-fails2.ipynb,manual_submissions/demo-fails2.pdf,1.0,2.0,0.0,0.0,1.0,0.0,4.
↪0,8
fails3,demo-fails3.ipynb,manual_submissions/demo-fails3.pdf,1.0,2.0,1.0,1.0,0.0,0.0,5.
↪0,8
passesAll,demo-passesAll.ipynb,manual_submissions/demo-passesAll.pdf,1.0,2.0,1.0,1.0,
↪1.0,0.0,6.0,8
fails3Hidden,demo-fails3Hidden.ipynb,manual_submissions/demo-fails3Hidden.pdf,1.0,2.0,
↪1.0,1.0,1.0,0.0,6.0,8
```

Let's make that a bit prettier:

Note that public tests are worth 1 point in the above example and `q1H`, `q2H`, and `q3H` are worth 2, 1, and 2 points,

respectively, for a total of 8 points (the `possible` column). In practice, you would probably have 0-point public tests, as hidden tests are meant to determine correctness. You should not that `fails2Hidden` failed `q2H` but not `q2`, and similarly for all other notebooks.

Congratulations, that's how you use Otter! If you've reached the end of this tutorial, you've correctly installed Otter and are ready to get grading. For more information about using Otter, see *Using Otter*.

CHAPTER 3

Using Otter

Otter has two major use cases: grading on the instructor's machine (local grading), and generating files to use Gradescope's autograding infrastructure.

3.1 Local Grading

Once you've *installed otter*, get started by creating some *test cases* and creating a requirements.txt file (if necessary).

3.1.1 Collecting Student Submissions

The first major decision is how you'll collect student submissions. You can collect these however you want, although otter has builtin compatibility with Gradescope and Canvas. If you choose either Gradescope or Canvas, just export the submissions and unzip that into some directory. If you are collecting another way, you'll need to create a metadata file. You can use either JSON or YAML format, and the structure is pretty simple: each element needs to have a filename and a student identifier. A sample YAML metadata file would be:

```
- identifier: 0
  filename: test-nb-0.ipynb
- identifier: 1
  filename: test-nb-1.ipynb
- identifier: 2
  filename: test-nb-2.ipynb
- identifier: 3
  filename: test-nb-3.ipynb
- identifier: 4
  filename: test-nb-4.ipynb
- identifier: 5
  filename: test-nb-5.ipynb
...
```

3.1.2 Support Files

If you have any files that are needed by the notebooks (e.g. data files), put these into the directory that contains the notebooks. You should also have your directory of tests nearby. At this stage, your directory should probably look something like this:

```
| grading
  | - requirements.txt
  | submissions
    | - meta.yml
    | - nb0.ipynb
    | - nb1.ipynb
    | - nb2.ipynb
    | - nb3.ipynb
    | - nb4.ipynb
    | - nb5.ipynb
    | - data.csv
  | tests
    | - q1.py
    | - q2.py
    | - q3.py
    | - q4.py
```

3.1.3 Grading

Now that you've set up the directory, let's get down to grading. Go to the terminal, `cd` into your grading directory (`grading` in the example above), and let's build the `otter` command. The first thing we need is our notebooks path or `-p` argument. Otter assumes this is `./`, but our notebooks are located in `./submissions`, so we'll need `-p submissions` in our command. We also need a tests directory, which otter assumes is at `./tests`; because this is where our tests are, we're alright on this front, and don't need a `-t` argument.

Now we need to tell otter how we've structured our metadata. If you're using a Gradescope or Canvas export, just pass the `-g` or `-c` flag, respectively, with no arguments. If you're using a custom metadata file, as in our example, pass the `-j` or `-y` flag with the path to the metadata file as its argument; in our case, we will pass `-y submissions/meta.yml`.

At this point, we need to make a decision: do we want PDFs? If there are questions that need to be manually graded, it might be nice to generate a PDF of each submission so that it can be easily read; if you want to generate PDFs, pass one of the `--pdf`, `--tag-filter`, or `--html-filter` flags (cf. *PDFs*). For our example, let's say that we *do* want PDFs.

Now that we've made all of these decisions, let's put our command together. Our command is:

```
otter -p submissions -y meta.yml --pdf -v
```

Note that Otter automatically found our requirements file at `./requirements.txt`. If it had been in a different location, we would have needed to pass the path to it to the `-r` flag. Note also that we pass the `-v` flag so that it prints verbose output. Once this command finishes running, you will end up with a new file and a new folder in your working directory:

```
| grading
  | - final_grades.csv
  | - requirements.txt
  | manual_submissions
    | - nb0.pdf
    | - nb1.pdf
```

(continues on next page)

(continued from previous page)

```
    | - nb2.pdf
    | - nb3.pdf
    | - nb4.pdf
    | - nb5.pdf
  | submissions
    ...
  | tests
    ...
```

Otter created the `final_grades.csv` file with the grades for each student, broken down by test, and the `manual_submissions` directory to house the PDF that was generated of each notebook.

Congrats, you're done! You can use the grades in the CSV file and the PDFs to complete grading however you want.

You can find more information about the command line utility *here*

3.2 Gradescope

To get started using otter with Gradescope, create some *test cases* and a requirements.txt file (if necessary). Once you have these pieces in place, put them into a directory along with any additional files that your notebook requires (e.g. data files), for example:

```
| gradescope
  | - data.csv
  | - requirements.txt
  | - utils.py
  | tests
    | - q1.py
    | - q2.py
    ...
```

To create the zipfile for Gradescope, use the `otter gen` command after `cd`ing into the directory you created. For the directory above, once I've `cd`ed into `gradescope`, I would run the following to generate the zipfile:

```
otter gen data.csv utils.py
```

As above, Otter automatically found our requirements file at `./requirements.txt`. Notice also that we didn't indicate the path to the tests directory; this is because the default argument of the `-t` flag is `./tests`, so otter found them automatically.

After this command finishes running, you should have a file called `autograder.zip` in the current working directory:

```
| gradescope
  | - autograder.zip
  | - data.csv
  | - requirements.txt
  | - utils.py
  | tests
    | - q1.py
    | - q2.py
    ...
```

To use this zipfile, create a Programming Assignment on Gradescope and upload this zipfile on the Configure Autograder page of the assignment. Gradescope will then build a Docker image on which it will grade each student's

submission.

You can find more information about Gradescope usage *here*.

CHAPTER 4

Test Files

Otter requires OK-formatted tests to check students' work against. These have a very specific format, described in detail in the OK documentation.

4.1 OK Format Caveats

While otter uses OK format, there are a few caveats to the tests when using them with otter.

- Otter only allows a single suite in each test, although the suite can have any number of cases. This means that `test["suite"]` should be a list of length 1, whose only element is a `dict`.
- Otter has an additional key in the `test` dict, called `hidden`. `test["hidden"]` should evaluate to a boolean. This is used to indicate whether or not the test should be shown on Gradescope when students submit their work. If `test["hidden"]` is `True`, then all cases will be shown to students on Gradescope. **This is not to be confused with the `hidden` key of each case, which are ignored by otter.**

4.2 Sample Test

Here is an annotated sample OK test:

```
test = {
    "name": "q1",         # name of the test
    "points": 1,          # number of points for the entire suite
    "hidden": False,      # whether the test is hidden on Gradescope
    "suites": [           # list of suites, only 1 suite allowed!
        {
            "cases": [                  # list of test cases
                {                       # each case is a dict
                    "code": r"""        # test, formatted for Python interpreter
                    >>> 1 == 1          # note that in any subsequence line of a␣
↪multiline
```

(continues on next page)

(continued from previous page)

```
                    True                # statement, the prompt becomes ... (see
↪below)
                    """,
                    "hidden": False,    # ignored by otter
                    "locked": False,    # ignored by otter
                },
                {
                    "code": r"""
                    >>> for i in range(4):
                    ...         print(i == 1)
                    False
                    True
                    False
                    False
                    """,
                    "hidden": False,
                    "locked": False,
                },
            ],
            "scored": False,            # ignored by otter
            "setup": "",                # ignored by otter
            "teardown": "",             # ignored by otter
            "type": "doctest"           # the type of test; only "doctest" allowed
        },
    ]
}
```

4.3 Writing OK Tests

You can find an online OK test generator that will assist you in generating these test files.

CHAPTER 5

Metadata

When organizing submissions for grading, Otter supports two major categories of exports:

- 3rd party collection exports, or
- instructor-made metadata files

These two categories are broken down and described below.

5.1 3rd Party Exports

If you are using a grading service like Gradescope or an LMS like Canvas to collect submissions, Otter can interpret the export format of these 3rd party services in place of a metadata file. To use a service like Gradescope or Canvas, download the submissions for an assignment, unzip the provided file, and then proceed as described in *the command line usage* using the metadata flag corresponding to your chosen service.

For example, if I had a Canvas export, I would unzip it, `cd` into the unzipped directory, copy my tests to `./tests`, and run

```
otter -c
```

to grade the submissions (assuming no extra requirements, no PDF generation, and no support files required).

5.2 Metadata Files

If you are not using a service like Gradescope or Canvas to collect submissions, instructors can also create a simple JSON- or YAML-formatted metadata file for their students' submissions and provide this to Otter.

The strucure of the metadata is quite simple: it is a list of 2-item dictionaries, where each dictionary has a student identifier stored as the `identifier` key and the filename of the student's submission as `filename`. An example of each is provided below.

YAML format:

```yaml
- identifier: 0
  filename: test-nb-0.ipynb
- identifier: 1
  filename: test-nb-1.ipynb
- identifier: 2
  filename: test-nb-2.ipynb
- identifier: 3
  filename: test-nb-3.ipynb
- identifier: 4
  filename: test-nb-4.ipynb
```

JSON format:

```json
[
  {
    "identifier": 0,
    "filename": "test-nb-0.ipynb"
  },
  {
    "identifier": 1,
    "filename": "test-nb-1.ipynb"
  },
  {
    "identifier": 2,
    "filename": "test-nb-2.ipynb"
  },
  {
    "identifier": 3,
    "filename": "test-nb-3.ipynb"
  },
  {
    "identifier": 4,
    "filename": "test-nb-4.ipynb"
  }
]
```

A JSON- or YAML-formatted metadata file is specified to Otter using the `-j` or `-y` flag, respectively. Each flag requires a single argument that corresponds to the path to the metadata file. See the "Basic Usage" section of the *CLI documentation* for more information.

CHAPTER 6

Student Usage

Otter provides an IPython API and a command line tool that allow students to run checks and export notebooks within the assignment environment.

6.1 The `Notebook` API

Otter supports in-notebook checks so that students can check their progress when working through assignments via the `otter.Notebook` class. The `Notebook` takes one optional parameter that corresponds to the path from the current working directory to the directory of tests; the default for this path is `./tests`.

```
import otter
grader = otter.Notebook()
```

If my tests were in `./hw00-tests`, then I would instantiate with

```
grader = otter.Notebook("hw00-tests")
```

Students can run tests in the test directory using `Notebook.check` which takes in a questio identifier (the file name without the `.py` extension) For example,

```
grader.check("q1")
```

will run the test `q1.py` in the tests directory. If a test passes, then the cell displays "All tests passed!" If the test fails, then the details of the first failing test are printed out, including the test code, expected output, and actual output:

In [6]:
```
1 square = lambda x: x**3
2 grader.check("q4")
```

Out[6]: 0 of 1 tests passed

Tests failed:

- **./tests/q4.py**

  Test code:

```
>>> square(5)
25
>>> square(2.5)
6.25
```

  Test result:

```
Trying:
    square(5)
Expecting:
    25
**********************************************************************
*
Line 2, in ./tests/q4.py 0
Failed example:
    square(5)
Expected:
    25
Got:
    125
Trying:
    square(2.5)
Expecting:
    6.25
**********************************************************************
*
Line 4, in ./tests/q4.py 0
Failed example:
    square(2.5)
Expected:
    6.25
Got:
    15.625
```

Students can also run all tests in the tests directory at once using `Notebook.check_all`:

```
grader.check_all()
```

This will rerun all tests against the current global environment and display the results for each tests concatenated into a single HTML output. It is recommended that this cell is put at the end of a notebook for students to run before they submit so that students can ensure that there are no variable name collisions, propagating errors, or other things that would cause the autograder to fail a test they should be passing.

Students can also use the `Notebook` class to generate their own PDFs for manual grading using the static method `Notebook.export`. `Notebook.export` has a required positional argument of the path to the notebook to be exported (usually the notebook that students are working through). There are also two optional arguments related to filtering cells: `filtering` indicates whether or not to filter notebooks and defaults to `True`, and `filter_type` indicates the filter type (`"tags"` or `"html"`) to use and defaults to `"html"`. You can find more information about PDF generation *here*.

Because `Notebook.export` is a static method, it can be called either from the class as `otter.Notebook.export()` or from the grader instance as `grader.export()`. We use the latter convetion in the examples below.

As an example, if I wanted to export `hw01.ipynb` with HTML comment filtering, my call would be

```
grader.export("hw01.ipynb")
```

as filtering is by defult on and the default filtering behavior is HTML comments. If I instead wanted to filter with cell tags, I would call

```
grader.export("hw01.ipynb", filter_type="tags")
```

Lastly, if I wanted to generate a PDF with no filtering, I would use

```
grader.export("hw01.ipynb", filtering=False)
```

We don't need to specify a `filter_type` argument here because it would be ignored by the fact that `filtering=False`.

6.2 Command Line Script Checker

Otter also features a command line tool that allows students to run checks on Python files from the command line. `otter check` takes one required argument, the path to the file that is being checked, and two optional flags:

- `-t` is the path to the directory of tests. If left unspecified, it is assumed to be `./tests`
- `-q` is the identifier of a specific question to check (the file name without the `.py` extension). If left unspecified, all tests in the tests directory are run.

The recommended file structure for using the checker is something like the one below:

```
| hw00
  | - hw00.py
  | tests
    | - q1.py
    | - q2.py
    ...
```

After `cd`ing into `hw00`, if I wanted to run the test q2.py, I would run

```
$ otter check hw00.py -q q2
All tests passed!
```

In the example above, I passed all of the tests. If I had failed any of them, I would get an output like that below:

```
$ otter check hw00.py -q q2
1 of 2 tests passed

Tests passed:
```

(continues on next page)

(continued from previous page)

```
 possible

Tests failed:
**********************************************************************
Line 2, in tests/q2.py 0
Failed example:
    1 == 1
Expected:
    False
Got:
    True
```

To run all tests at once, I would run

```
$ otter check hw00.py
Tests passed:
 q1  q3  q4  q5

Tests failed:
**********************************************************************
Line 2, in tests/q2.py 0
Failed example:
    1 == 1
Expected:
    False
Got:
    True
```

As you can see, I passed for of the five tests above, and filed q2.py.

If I instead had the directory structure below (note the new tests directory name)

```
| hw00
  | - hw00.py
  | hw00-tests
    | - q1.py
    | - q2.py
    ...
```

then all of my commands would be changed by adding `-t hw00-tests` to each call. As an example, let's rerun all of the tests again:

```
$ otter check hw00.py -t hw00-tests
Tests passed:
 q1  q3  q4  q5

Tests failed:
**********************************************************************
Line 2, in hw00-tests/q2.py 0
Failed example:
    1 == 1
Expected:
    False
Got:
    True
```

CHAPTER 7

Command Line Utility

The command line interface allows instructors to grade notebooks locally by launching Docker containers on the instructor's machine that grade notebooks and return a CSV of grades and (optionally) PDF versions of student submissions for manually graded questions.

7.1 Prerequisites

Before using the command line utility, you should have

- written tests for the assignment
- downloaded submissions into a directory
- written a *metadata file* if not using a Gradescope or Canvas export

7.2 Using the CLI

The CLI, encapsulated in the `otter` command, runs the local grading process and defines the options that instructors can set when grading. A comprehensive list of flags is provided below.

7.2.1 Basic Usage

The simplest usage of the `otter` command is when we have a directory structure as below (and we have `cd`ed into `grading` in the command line) and we don't require PDFs or additional requirements.

```
| grading
  | - meta.yml
  | - nb0.ipynb
  | - nb1.ipynb
  | - nb2.ipynb
```

(continues on next page)

(continued from previous page)

```
  ...
  | tests
    | - q1.py
    | - q2.py
    | - q3.py
    ...
```

In the case above, our otter command would be, very simply,

```
otter -y meta.yml
```

Because the submissions are on the current working directory (`grading`), our tests are at `./tests`, and we don't mind output to `./`, we can use the defualt values of the `-p`, `-t`, and `-o` flags, leaving the only necessary flag the metadata flag. Since we have a YAML metadata file, we specify `-y` and pass the path to the metadata file, `./meta.yml`.

After grader, our directory will look like this:

```
| grading
  | - final_grades.csv
  | - meta.yml
  | - nb0.ipynb
  | - nb1.ipynb
  | - nb2.ipynb
  ...
  | tests
    | - q1.py
    | - q2.py
    | - q3.py
    ...
```

and the grades for each submission will be in `final_grades.csv`.

If we wanted to generate PDFs for manual grading, we would specify one of the three PDF flags:

```
otter -y meta.yml --pdf
```

and at the end of grading we would have

```
| grading
  | - final_grades.csv
  | - meta.yml
  | - nb0.ipynb
  | - nb1.ipynb
  | - nb2.ipynb
  ...
  | manual_submissions
    | - nb0.pdf
    | - nb1.pdf
    | - nb2.pdf
    ...
  | tests
    | - q1.py
    | - q2.py
    | - q3.py
    ...
```

7.2.2 Metadata Flags

The four metadata flags, `-g`, `-c`, `-j`, and `-y`, correspond to different export/metadata file formats. Also note that the latter two require you to specify a path to the metadata file. You must specify a metadata flag every time you run `otter`, and you may not specify more than one. For more information about metadata and export formats, see *this page*.

7.2.3 Requirements

The `ucbdsinfra/otter-grader` Docker image comes preinstalled with the following Python packages and their dependencies:

- datascience
- jupyter_client
- ipykernel
- matplotlib
- pandas
- ipywidgets
- scipy
- tornado
- nb2pdf
- otter-grader

If you require any packages not listed above, or among the dependencies of any packages above, you should create a requirements.txt file *containing only those packages*. If this file is created in the working directory (i.e. `./requirements.txt`), then Otter will automatically find this file and include it. If this file is not at `./requirements.txt`, pass its path to the `-r` flag.

For example, continuining the example above with the package SymPy, I would create a requirements.txt

```
| grading
  | - meta.yml
  | - nb0.ipynb
  | - nb1.ipynb
  | - nb2.ipynb
  ...
  | - requirements.txt
  | tests
    | - q1.py
    | - q2.py
    | - q3.py
    ...
```

that lists only SymPy

```
$ cat requirements.txt
sympy
```

Now my `otter` call, using HTML comment filtered PDF generation this time, would become

```
otter -y meta.yml --html-filter
```

Note the lack of the `-r` flag; since I created my requirements file in the working directory, Otter found it automatically.

7.2.4 Grading Python Scripts

If I wanted to grade Python scripts instead of IPython notebooks, my call to `otter` would only add the `-s` flag. Consider the directory structure below:

```
| grading
  | - meta.yml
  | - sub0.py
  | - sub1.py
  | - sub2.py
  ...
  | - requirements.txt
  | tests
    | - q1.py
    | - q2.py
    | - q3.py
    ...
```

My call to grade these submissions would be

```
otter -sy meta.yml
```

Note the lack of a PDF flag, as it doesn't make sense to convert Python files to PDFs. PDF flags only work when grading IPython Notebooks.

7.2.5 Support Files

Some notebooks require support files to run (e.g. data files). If your notebooks require any such files, there are two ways to get them into the container so that they are available to notebooks:

- specifying paths to the files with the `-f` flag
- putting them into the notebook path

Suppose that my notebooks in `grading` required `data.csv` in my `../data` directory:

```
| data
  | - data.csv
| grading
  | - meta.yml
  | - nb0.ipynb
  | - nb1.ipynb
  | - nb2.ipynb
  ...
  | - requirements.txt
  | tests
    | - q1.py
    | - q2.py
    | - q3.py
    ...
```

I could pass this data into the container using the `otter` call

```
otter -y meta.yml -f ../data/data.csv
```

Or I could move (or copy) `data.csv` into `grading`:

```
$ mv ../data/data.csv ./
```

```
| data
| grading
  | - data.csv
  | - meta.yml
  | - nb0.ipynb
  | - nb1.ipynb
  | - nb2.ipynb
  ...
  | - requirements.txt
  | tests
    | - q1.py
    | - q2.py
    | - q3.py
    ...
```

and then just run `otter` as normal:

```
otter -y meta.yml
```

All non-notebook files in the notebooks path are copied into all of the containers, so `data.csv` will be made available to all notebooks.

CHAPTER 8

Gradescope

Otter-Grader also allows instructors to use Gradescope's autograding system to collect and grade students' submissions. This section assumes that instructors are familiar with Gradescope's interface and know how to set up assignments on Gradescope; for more information on using Gradescope, see their help pages.

8.1 Writing Tests for Gradescope

The tests that are used by the Gradescope autograder are the same as those used in other uses of Otter, but there is one important field that is relevant to Gradescope that is not pertinent to any other uses.

As noted in the second bullet here, the `test` variable has a `"hidden"` key which maps to a boolean value that indicates the visibility of the test on Gradescope. For any test, if `test["hidden"]` evaluates to `False`, then the test result will be shown to students when they submit to Gradescope. If they pass, the test name will show in a green box; if they fail, then the usual failed test output will show.

For more information on how tests are displayed to students, see *below*.

8.2 Using the Command Line Generator

To use Otter with Gradescope's autograder, you must first generate a zipfile that you will upload to Gradescope so that they can create a Docker image with which to grade submissions. Otter's command line utility `otter gen` allows instructors to create this zipfile from their machines.

8.2.1 Before Using `otter gen`

Before using `otter gen`, you should already have

- written *tests* for the assignment
- created a Gradescope autograder assignment

- have collected extra requirements into a requirements.txt file (see here).

8.2.2 Directory Structure

For the rest of this page, assume that we have the following directory structure:

```
| hw00-dev
  | - data.csv
  | - hw00-sol.ipynb
  | - hw00.ipynb
  | - requirements.txt
  | - utils.py
  | tests
    | - q1.py
    | - q2.py
    ...
  | hidden-tests
    | - q1.py
    | - q2.py
    ...
```

Also assume that we have `cd`ed into `hw00-dev`.

8.2.3 Usage

The general usage of `otter gen` is to create a zipfile at some output path (`-o` flag, default `./`) which you will then upload to Gradescope. `otter gen` has three optional flags:

If you do not specify `-t` or `-o`, then the defaults will be used. If you do not specify `-r`, Otter looks in the working directory for `requirements.txt` and automatically adds it if found; if it is not found, then it is assumed there are no additional requirements. There is also an optional positional argument that goes at the end of the command, `files`, that is a list of any files that are required for the notebook to execute (e.g. data files, Python scripts).

The simplest usage in our example would be

```
otter gen
```

This would create a zipfile with the tests in `./tests` and no extra requirements or files. If we needed `data.csv` in the notebook, our call would instead become

```
otter gen data.csv
```

Note that if we needed the requirements in `requirements.txt`, our call wouldn't change, since Otter automatically found `./requirements.txt`.

Now let's say that we maintained to different directories of tests: `tests` with public versions of tests and `hidden-tests` with hidden versions. Because I want to grade with the hidden tests, my call then becomes

```
otter gen -t hidden-tests data.csv
```

Now let's say that I need some functions defined in `utils.py`; then I would add this to the last part of my `otter gen` call:

```
otter gen -t hidden-tests data.csv utils.py
```

Pass/Fail Thresholds

The Gradescope generator supports providing a pass/fail threshold. A threshold is passed as a float between 0 and 1 such that if a student receives at least that percentage of points, they will receive full points as their grade and 0 points otherwise.

The threshold is specified with the `--threshold` flag:

```
otter gen -t hidden-tests data.csv --threshold 0.75
```

For example, if a student passes a 2- and 1- point test but fails a 4-point test (a 43%) on a 25% threshold, they will get all 7 points. If they only pass the 1-point test (a 14%), they will get 0 points.

Overriding Points Possible

By default, the number of points possible on Gradescope is the sum of the point values of each test. This value can be overrided, however, to some other value using the `--points` flag, which accepts an integer. Then the number of points awarded will be the provided points value scaled by the percentage of points awarded by the autograder.

For example, if a student passes a 2- and 1- point test but fails a 4-point test, they will receive (2 + 1) / (2 + 1 + 4) * 2 = 0.8571 points out of a possible 2.

As an example, the command below scales the number of points to 3:

```
otter gen -t hidden-tests data.csv --points 3
```

Showing Autograder Results

The generator lastly allows intructors to specify whether or not the stdout of the grading process (anything printed to the console by the grader or the notebook) is shown to students. **The stdout includes a summary of the student's test results, including the points earned and possible of public *and* hidden tests, as well as the visibility of tests as indicated by `test["hidden"]`.**

This behavior is turned off by default and can be turned on by passing the `--show-results` flag to `otter gen`.

```
otter gen -t hidden-tests data.csv --show-results
```

If `--show-results` is passed, the stdout will be made available to students *only after grades are published on Gradescope*. The *next section* details more about what is included in the stdout.

8.3 Gradescope Results

This section details how results are displayed to students and instructors on Gradescope.

8.3.1 Instructor View

Once a student's submission has been autograder, the Autograder Results page will show the stdout of the grading process in the "Autograder Output" box and the student's score in the side bar to the right of the output. The stdout includes a DataFrame that contains the student's score breakdown by question:

Autograder Output

```
[W:pyppeteer.chromium_downloader] start chromium download.
Download may take a few minutes.
[W:pyppeteer.chromium_downloader]
chromium download done.
[W:pyppeteer.chromium_downloader] chromium extracted to: /root/.
```

	name	score	possible	visibility
0	q1_1	1.0	1	visible
1	q1_2	1.0	1	visible
2	q1_3	1.0	1	visible
3	q1_4	1.0	1	visible
4	q2_1	1.0	1	visible
5	q2_2	0.0	0	visible
6	q2_2H	0.0	1	hidden
7	q2_3	0.0	0	visible
8	q2_3H	0.0	1	hidden
9	q2_4	1.0	1	visible
10	q2_5	0.0	0	visible
11	q2_5H	0.0	1	hidden
12	q3_1	1.0	1	visible
13	q3_2	1.0	1	visible
14	q3_3	0.0	0	visible
15	q3_3H	1.0	1	hidden
16	q3_5	0.0	0	visible
17	q3_5H	1.0	1	hidden
18	q3_6	1.0	1	visible
19	q3_7	0.0	0	visible
20	q3_7H	0.0	1	hidden
21	q3_8	0.0	0	visible
22	q3_8H	0.0	1	hidden

Below the autograder output, each test case is broken down into boxes. If there is no output for the box, then that test was passed. If a test is failed, then the usual test failure output is displayed.

q1_1

q1_2

q1_3

q1_4

q2_1

q2_2

q2_2H

```
**********************************************************************
Line 2, in /autograder/source/tests/q2_2H.py 0
Failed example:
    max(nondelta_below_2000["Quantity"]) == 1990.21
Expected:
    True
Got:
    False
```

Instructors will be able to see *all* tests. The visibility of a test to students is indicated to instructors by the  icon (all tests with this icon are hidden to students).

8.3.2 Student View

On submission, students will only be able to see the results of those tests for which `test["hidden"]` evaluates to `True` (see *Test Files* for more info). If `test["hidden"]` is `False` or not specified, then `test` is hidden.

If `--show-results` was specified when constructing the autograder zipfile, then the autograder output from above will be shown to students *after grades are published on Gradescope*. Students will **not** be able to see the results of hidden tests nor the tests themselves, but they will see that they failed some hidden test in the printed DataFrame from the stdout.

Note that, because some tests are hidden, students will never see the autograder score in the right sidebar; instead,

their score will only show as a dash – out of the points possible. Therefore, the only way for students to calculate their autograder score is to use the DataFrame printed to the stdout if `--show-results` is passed.

CHAPTER 9

PDF Generation and Filtering

When exporting IPython notebooks as PDFs, Otter uses the library nb2pdf that relies on nbpdfexport and chromium to export notebooks without pandoc or LaTeX. This requires that chromium be installed both in the Docker container being used for grading and on the JupyterHub distribution on which students export their notebooks from the `Notebook` API.

The `Notebook.export` function encapsulates PDF generation on the student end, and by default filtering is turned on. To generate unfiltered PDFs with `Notebook.export`, set `filtering=False` in your call. To generate unfiltered PDFs from the command line, use the `--pdf` flag when calling `otter`.

9.1 Cell Filtering

nb2pdf supports two different formats for filtering cells when exporting notebooks: using cell tags and using HTML comments.

9.1.1 Cell Tag Filtering

When generating the PDF (if filtering is indicated by the `--tag-filter` flag or by setting `filter_type="tags"` in `Notebook.export`), the following cells will be **included** in the export:

- All Markdown cells
- Code cells that have an image in their output
- All cells tagged with `include`

If you would like to override the behavior above, tag a cell with `ignore` and it will not be included.

9.1.2 HTML Comment Filtering

Alternatively, you can place HTML comments in Markdown cells to capture everything in between them in the output. To start a filtering group, place the comment `<!-- BEGIN QUESTION -->` whereever you want to start exporting

and place `<!-- END QUESTION -->` at the end of the filtering group. Everything capture between these comments will be exported, and everything outside them removed. You can have multiple filtering groups in a notebook.

This filtering behavior is triggered with the `--html-filter` flag or in the default behavior of `Notebook.export` (as the `filter_type` argument defaults to `"html"`).

CHAPTER 10

otter package

Otter-Grader is an open-source local grader from the Division of Computing, Data Science, and Society at the University of California, Berkeley. It is designed to be a scalable grader that utilizes parallel Docker containers on the instructor's machine in order to remove the traditional overhead requirement of a live server. It also supports student-run tests in Jupyter Notebooks and from the command line, and is compatible with Gradescope's proprietary autograding service.

CHAPTER 11

Changelog

v0.4.7:

- fix relative import issue on Gradescope (again, *sigh*)

v0.4.6: re-release of v0.4.5

v0.4.5:

- added missing patch of `otter.Notebook.export` in otter/grade.py
- added `__version__` global in otter/**init**.py
- fixed relative import issue when running on Gradescope
- fixed not finding/rerunning tests on Gradescope with `otter.Notebook.check`

v0.4.4:

- fixed template escape bug in otter/gs_generator.py

v0.4.3:

- fixed dead link in docs/gradescope.md
- updated to Python 3.7 in setup.sh for Gradescope
- made `otter` and `otter gen` CLIs find `./requirements.txt` automatically if it exists
- fix bug where GS generator fails if no `-r` flag specified