Otter-Grader Documentation

UCBDS Infrastructure Team

May 03, 2024

CONTENTS

CHAPTER

ONE

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 zip file and unzip it into some directory on your machine; you should have the following directory structure:

```
tutorial
├── demo.ipynb
├── requirements.txt
└── submissions
    ├── ipynbs
    │   ├── demo-fails1.ipynb
    │   ├── demo-fails2.ipynb
    │   ├── demo-fails2Hidden.ipynb
    │   ├── demo-fails3.ipynb
    │   ├── demo-fails3Hidden.ipynb
    │   └── demo-passesAll.ipynb
    └── zips
        ├── demo-fails1.zip
        ├── demo-fails2.zip
        ├── demo-fails2Hidden.zip
        ├── demo-fails3.zip
        ├── demo-fails3Hidden.zip
        └── demo-passesAll.zip
```

This section describes the basic execution of Otter's tools using the provided zip file. It is meant to verify your installation and to *loosely* describe how a few Otter tools are used. This tutorial covers Otter Assign, Otter Generate, and Otter Grade.

1.1 Otter Assign

Start by moving into the `tutorial` directrory. This directory includes the master notebook `demo.ipynb`. Look over this notebook to get an idea of its structure. It contains five questions, four code and one Markdown (two of which are manually-graded). Also note that the assignment configuration in the first cell tells Otter Assign to generate a solutions PDF and an autograder zip file and to include special submission instructions before the export cell. To run Otter Assign on this notebook, run

```
otter assign demo.ipynb dist
```

Otter Assign should create a `dist` directory which contains two further subdirectories: `autograder` and `student`. The `autograder` directory contains the Gradescope autograder, solutions PDF, and the notebook with solutions. The `student` directory contains just the sanitized student notebook.

```
tutorial/dist
├── autograder
│   ├── autograder.zip
│   ├── demo-sol.pdf
│   ├── demo.ipynb
│   ├── otter_config.json
│   └── requirements.txt
└── student
    └── demo.ipynb
```

For more information about the configurations for Otter Assign and its output format, see *Creating Assignments*.

1.2 Otter Generate

In the `dist/autograder` directory created by Otter Assign, there should be a file called `autograder.zip`. This file is the result of using Otter Generate to generate a zip file with all of your tests and requirements, which is done invisibly by Otter Assign when it is used (which it is configured to do in the assignment metadata). Alternatively, you could generate this zip file yourself from the contents of `dist/autograder` by running

```
otter generate
```

in that directory (but this is not recommended).

1.3 Otter Grade

Note: You should complete the Otter Assign tutorial above before running this tutorial, as you will need some of its output files.

At this step of grading, the instructor faces a choice: where to grade assignments. The rest of this tutorial details how to grade assignments locally using Docker containers on the instructor's machine. You can also grade on Gradescope or without containerization, as described in the *Executing Submissions* section.

Let's now construct a call to Otter that will grade these notebooks. We will use `dist/autograder/autograder.zip` from running Otter Assign to configure our grading image. Our notebooks are in the `ipynbs` subdirectory and contain a couple of written questions, so we'll specify the `--pdfs` flag to indicate that Otter should grab the PDFs out of the Docker containers. Lastly, we'll name the assignment `demo` with the `-n` flag.

Let's run Otter on the notebooks:

```
otter grade -n demo -a dist/autograder/demo-autograder_*.zip --pdfs -v submissions/ipynbs
```

(The `-v` flag is 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` and `submission_pdfs`. The former should contain the grades for each file, and should look something like this:

```
file,q1,q2,q3
fails3Hidden.ipynb,1.0,1.0,0.5
passesAll.ipynb,1.0,1.0,1.0
fails1.ipynb,0.6666666666666666,1.0,1.0
fails2Hidden.ipynb,1.0,0.5,1.0
fails3.ipynb,1.0,1.0,0.375
fails2.ipynb,1.0,0.0,1.0
```

Let's make that a bit prettier:

file	q1	q2	q3
fails3Hidden.ipynb	1.0	1.0	0.5
passesAll.ipynb	1.0	1.0	1.0
fails1.ipynb	0.6666666666666666	1.0	1.0
fails2Hidden.ipynb	1.0	0.5	1.0
fails3.ipynb	1.0	1.0	0.375
fails2.ipynb	1.0	0.0	1.0

The latter, the `submission_pdfs` directory, should contain the filtered PDFs of each notebook (which should be relatively similar).

Otter Grade can also grade the zip file exports provided by the `Notebook.export` method. All we need to do is add the `--ext` flag to indicate that the submissions are zip files. We have provided some example submissions, with the same notebooks as above, in the `zips` directory, so let's grade those:

```
otter grade -n demo -a dist/autograder/demo-autograder_*.zip -v --ext zip submissions/
↪zips
```

This should have the same CSV output as above but no `submission_pdfs` directory since we didn't tell Otter to generate PDFs.

You can learn more about the grading workflow for Otter in *this section*.

CHAPTER

TWO

TEST FILES

2.1 Python Format

Python test files can follow one of two formats: exception-based or the OK format.

2.1.1 Exception-Based Format

The exception-based test file format relies on an idea similar to unit tests in pytest: an instructor writes a series of test case functions that raise errors if the test case fails; if no errors are raised, the test case passes.

Test case functions should be decorated with the `otter.test_files.test_case` decorator, which holds metadata about the test case. The `test_case` decorator takes (optional) the arguments:

- `name`: the name of the test case
- `points`: the point value of the test case (default `None`)
- `hidden`: whether the test case is hidden (default `False`)
- `success_message`: a message to display to the student if the test case passes
- `failure_message`: a message to display to the student if the test case fails

The test file should also declare the global variable `name`, which should be a string containing the name of the test case, and (optionally) `points`, which should be the total point value of the question. If this is absent (or set to `None`), it will be inferred from the point values of each test case as described *below*. Because Otter also supports OK-formatted test files, the global variable `OK_FORMAT` must be set to `False` in exception-based test files.

When a test case fails and an error is raised, the full stack trace and error message will be shown to the student. This means that you can use the error message to provide the students with information about why the test failed:

```
assert fib(1) == 0, "Your fib function didn't handle the base case correctly."
```

Calling Test Case Functions

Because test files are evaluated before the student's global environment is provided, test files will not have access to the global environment during execution. However, Otter uses the test case function arguments to pass elements from the global environment, or the global environment itself.

If the test case function has an argument name `env`, the student's global environment will be passed in as the value for that argument. For any other argument name, the value of that variable in the student's global environment will be passed in; if that variable is not present, it will default to `None`.

For example, the test function with the signature

```
test_square(square, env)
```

would be called like

```
test_square(square=globals().get("square"), env=globals())
```

in the student's environment.

Sample Test

Here is a sample exception-based test file. The example below tests a student's `sieve` function, which uses the Sieve of Eratosthenes to return a set of the `n` first prime numbers.

```
from otter.test_files import test_case

OK_FORMAT = False

name = "q1"
points = 2

@test_case()
def test_low_primes(sieve):
    assert sieve(1) == set()
    assert sieve(2) == {2}
    assert sieve(3) == {3}

@test_case(points=2, hidden=True)
def test_higher_primes(env):
    assert env["sieve"](20) == {2, 3, 5, 7, 11, 13, 17, 19}
    assert sieve(100) == {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
        61, 67, 71, 73, 79, 83, 89, 97}
```

Resolving Point Values

Point values for each test case and the question defined by the test file will be resolved as follows:

- If one or more test cases specify a point value and no point value is specified for the question, each test case with unspecified point values is assumed to be worth 0 points unless all test cases with specified points are worth 0 points; in this case, the question is assumed to be worth 1 point and the test cases with unspecified points are equally weighted.
- If one or more test cases specify a point value and a point value *is* specified for the test file, each test case with unspecified point values is assumed to be equally weighted and together are worth the test file point value less the sum of specified point values. For example, in a 6-point test file with 4 test cases where two test cases are each specified to be worth 2 points, each of the other test cases is worth $\frac{6-(2+2)}{2} = 1$ point.)
- If no test cases specify a point value and a point value *is* specified for the test file, each test case is assumed to be equally weighted and is assigned a point value of $\frac{p}{n}$ where p is the number of points for the test file and n is the number of test cases.
- If no test cases specify a point value and no point value is specified for the test file, the test file is assumed to be worth 1 point and each test case is equally weighted.

2.1.2 OK Format

You can also write OK-formatted tests to check students' work against. These have a very specific format, described in detail in the OkPy documentation. There is also a resource we developed on writing autograder tests that can be found here; this guide details things like the doctest format, the pitfalls of string comparison, and seeding tests.

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["suites"]` should be a `list` of length 1, whose only element is a `dict`.
- Otter uses the `"hidden"` key of each test case only on Gradescope. When displaying results on Gradescope, the `test["suites"][0]["cases"][<int>]["hidden"]` should evaluate to a boolean that indicates whether or not the test is hidden. The behavior of showing and hiding tests is described in *Grading on Gradescope*.

Writing OK Tests

We recommend that you develop assignments using *Otter Assign*, a tool which will generate these test files for you. If you already have assignments or would prefer to write them yourself, you can find an online OK test generator that will assist you in generating these test files without using Otter Assign.

Because Otter also supports exception-based test files, the global variable `OK_FORMAT` must be set to `True` in OK-formatted test files.

Sample Test

Here is an annotated sample OK test:

```
OK_FORMAT = True

test = {
    "name": "q1",       # name of the test
    "points": 1,        # number of points for the entire suite
    "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
                    True                # statement, the prompt becomes ... (see below)
                    """,
                    "hidden": False,    # used to determine case visibility on Gradescope
                    "locked": False,    # ignored by Otter
                },
                {
                    "code": r"""
                    >>> for i in range(4):
                    ...     print(i == 1)
                    False
                    True
```

(continues on next page)

(continued from previous page)

```
                    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
        },
    ]
}
```

2.2 R Format

Ottr tests are constructed by creating a JSON-like object (a `list`) that has a list of instances of the R6 class `ottr::TestCase`. The body of each test case is a block of code that should raise an error if the test fails, and no error if it passes. The list of test cases should be declared in a global `test` variable. The structure of the test file looks something like:

```
test = list(
    name = "q1",
    cases = list(
        ottr::TestCase$new(
            name = "q1a",
            code = {
                testthat::expect_true(ans.1 > 1)
                testthat::expect_true(ans.1 < 2)
            }
        ),
        ottr::TestCase$new(
            name = "q1b",
            hidden = TRUE,
            code = {
                tol = 1e-5
                actual_answer = 1.27324
                testthat::expect_true(ans.1 > actual_answer - tol)
                testthat::expect_true(ans.1 < actual_answer + tol)
            }
        )
    )
)
```

Note that the example above uses the `expect_*` functions exported by `testthat` to assert conditions that raise errors. The constructor for `ottr::TestCase` accepts the following arguments:

- `name`: the name of the test case
- `points`: the point value of the test case

- `hidden`: whether this is a hidden test
- `code`: the body of the test

Otter has different test file formats depending on which language you are grading. Python test files can follow an exception-based unit-test-esque format like pytest, or can follow the OK format, a legacy of the OkPy autograder that Otter inherits from. R test files are like unit tests and rely on `TestCase` classes exported by the `ottr` package.

CHAPTER

THREE

CREATING ASSIGNMENTS

3.1 Notebook Format

Otter's notebook format groups prompts, solutions, and tests together into questions. Autograder tests are specified as cells in the notebook and their output is used as the expected output of the autograder when generating tests. Each question has metadata, expressed in raw YAML config cell when the question is declared.

The Otter Assign format uses raw notebook cells as boundary cells. Each boundary cell denotes the start or end of a block and contains *valid YAML syntax*. First-line comments are used in these YAML raw cells to denote what type of block is being entered or ended. If you're authoring notebooks in an environment where raw cells are unavailable (such as Deepnote or Google Colab), see *Alternative to Raw Cells*.

In the this format, Python and R notebooks follow the same structure. There are some features available in Python that are not available in R, and these are noted below, but otherwise the formats are the same.

3.1.1 Assignment Config

In addition to various command line arguments discussed below, Otter Assign also allows you to specify various assignment generation arguments in an assignment config cell. These are very similar to the question config cells described in the next section. Assignment config, included by convention as the first cell of the notebook, places YAML-formatted configurations in a raw cell that begins with the comment `# ASSIGNMENT CONFIG`.

```
# ASSIGNMENT CONFIG
init_cell: false
export_cell: true
generate: true
# etc.
```

This cell is removed from both output notebooks. These configurations can be **overwritten** by their command line counterparts (if present). The options, their defaults, and descriptions are listed below. Any unspecified keys will keep their default values. For more information about many of these arguments, see *Usage and Output*. Any keys that map to sub-dictionaries (e.g. `export_cell`, `generate`) can have their behaviors turned off by changing their value to `false`. The only ones that default to true (with the specified sub-key defaults) are `export_cell` and `generate`.

```
name: null                                  # a name for the assignment (to validate␣
↪that students submit to the correct autograder)
config_file: null                           # path to a file containing assignment␣
↪configurations; any configurations in this file are overridden by the in-notebook␣
↪config
requirements: null                          # the path to a requirements.txt file or a␣
↪list of packages
```

(continues on next page)

(continued from previous page)

```
overwrite_requirements: false                # whether to overwrite Otter's default␣
↪requirement.txt in Otter Generate
environment: null                            # the path to a conda environment.yml file
run_tests: true                              # whether to run the assignment tests␣
↪against the autograder notebook
solutions_pdf: false                         # whether to generate a PDF of the␣
↪solutions notebook
template_pdf: false                          # whether to generate a filtered Gradescope␣
↪assignment template PDF
init_cell: true                              # whether to include an Otter␣
↪initialization cell in the output notebooks
check_all_cell: false                        # whether to include an Otter check-all␣
↪cell in the output notebooks
export_cell:                                 # whether to include an Otter export cell␣
↪in the output notebooks
  instructions: ''                           # additional submission instructions to␣
↪include in the export cell
  pdf: true                                  # whether to include a PDF of the notebook␣
↪in the generated zip file
  filtering: true                            # whether the generated PDF should be␣
↪filtered
  force_save: false                          # whether to force-save the notebook with␣
↪JavaScript (only works in classic notebook)
  run_tests: true                            # whether to run student submissions␣
↪against local tests during export
  files: []                                  # a list of other files to include in the␣
↪student submissions' zip file
  require_no_pdf_ack:                        # whether to require students to␣
↪acknowledge that a PDF could not be created if one is meant to be included in the␣
↪submission zip file
    # Default value: false
    message: null                            # a message to show to students if a PDF is␣
↪meant to be included in the submission but cannot be generated
seed:                                        # intercell seeding configurations
  variable: null                             # a variable name to override with the␣
↪autograder seed during grading
  autograder_value: null                     # the value of the autograder seed
  student_value: null                        # the value of the student seed
generate:                                    # grading configurations to be passed to␣
↪Otter Generate as an otter_config.json
  score_threshold: null                      # a score threshold for pass-fail␣
↪assignments
  points_possible: null                      # a custom total score for the assignment;␣
↪if unspecified the sum of question point values is used.
  show_stdout: false                         # whether to display the autograding␣
↪process stdout to students on Gradescope
  show_hidden: false                         # whether to display the results of hidden␣
↪tests to students on Gradescope
  show_all_public: false                     # whether to display all test results if␣
↪all tests are public tests
  seed: null                                 # a random seed for intercell seeding
  seed_variable: null                        # a variable name to override with the seed
```

(continues on next page)

(continued from previous page)

```
  grade_from_log: false                       # whether to re-assemble the student's␣
↪environment from the log rather than by re-executing their submission
  serialized_variables: null                  # a mapping of variable names to type␣
↪strings for validating a deserialized student environment
  pdf: false                                  # whether to generate a PDF of the notebook␣
↪when not using Gradescope auto-upload
  token: null                                 # a Gradescope token for uploading a PDF of␣
↪the notebook
  course_id: null                             # a Gradescope course ID for uploading a␣
↪PDF of the notebook
  assignment_id: null                         # a Gradescope assignment ID for uploading␣
↪a PDF of the notebook
  filtering: false                            # whether the generated PDF should have␣
↪cells filtered out
  pagebreaks: false                           # whether the generated PDF should have␣
↪pagebreaks between filtered sections
  debug: false                                # whether to run the autograder in debug␣
↪mode (without ignoring errors)
  autograder_dir: /autograder                 # the directory in which autograding is␣
↪taking place
  lang: python                                # the language of the assignment; one of {
↪'python', 'r'}
  miniconda_path: /root/mambaforge            # the path to the mamba install directory
  plugins: []                                 # a list of plugin names and configuration␣
↪details for grading
  logo: true                                  # whether to print the Otter logo to stdout
  print_summary: true                         # whether to print the grading summary
  print_score: true                           # whether to print out the submission score␣
↪in the grading summary
  zips: false                                 # whether zip files are being graded
  log_level: null                             # a log level for logging messages; any␣
↪value suitable for ``logging.Logger.setLevel``
  assignment_name: null                       # a name for the assignment to ensure that␣
↪students submit to the correct autograder
  warn_missing_pdf: false                     # whether to add a 0-point public test to␣
↪the Gradescope output to indicate to students whether a PDF was found/generated for␣
↪this assignment
  force_public_test_summary: true             # whether to show a summary of public test␣
↪case results when show_hidden is true
  submit_blank_pdf_on_export_failure: false  # whether to submit a blank PDF to the␣
↪manual-grading Gradescope assignment if a PDF cannot be generated from the submission
  use_submission_pdf: false                   # use the PDF in the submission zip file␣
↪instead of exporting a new one; if no PDF is present, a new one is generated anyway;␣
↪assumes there is only 1 PDF file in the submission
save_environment: false                       # whether to save the student's environment␣
↪in the log
variables: null                               # a mapping of variable names to type␣
↪strings for serializing environments
ignore_modules: []                            # a list of modules to ignore variables␣
↪from during environment serialization
files: []                                     # a list of other files to include in the␣
↪output directories and autograder
```

(continues on next page)

(continued from previous page)

```
autograder_files: []                        # a list of other files only to include in␣
↪the autograder
plugins: []                                 # a list of plugin names and configurations
tests:                                      # information about the structure and␣
↪storage of tests
  files: false                              # whether to store tests in separate files,␣
↪instead of the notebook metadata
  ok_format: true                           # whether the test cases are in OK-format␣
↪(instead of the exception-based format)
  url_prefix: null                          # a URL prefix for where test files can be␣
↪found for student use
show_question_points: false                 # whether to add the question point values␣
↪to the last cell of each question
runs_on: default                            # the interpreter this notebook will be run␣
↪on if different from the default interpreter (one of {'default', 'colab', 'jupyterlite'})
python_version: null                        # the version of Python to use in the␣
↪grading image (must be 3.6+)
channel_priority_strict: true               # whether to set conda's channel_priority␣
↪config to strict in the setup.sh file
exclude_conda_defaults: false               # whether to exclude conda's defaults␣
↪channel in the generated environment.yml file
```

For assignments that share several common configurations, these can be specified in a separate YAML file whose path is passed to the `config_file` key. When this key is encountered, Otter will read the file and load the configurations defined therein into the assignment config for the notebook it's running. Any keys specified in the notebook itself will override values in the file.

```
# ASSIGNMENT CONFIG
config_file: ../assignment_config.yml
files:
    - data.csv
```

All paths specified in the configuration should be **relative to the directory containing the master notebook**. If, for example, you were running Otter Assign on the `lab00.ipynb` notebook in the structure below:

```
dev
├── lab
│   └── lab00
│       ├── data
│       │   └── data.csv
│       ├── lab00.ipynb
│       └── utils.py
└── requirements.txt
```

and you wanted your requirements from `dev/requirements.txt` to be included, your configuration would look something like this:

```
requirements: ../../requirements.txt
files:
    - data/data.csv
    - utils.py
```

Requirements

The *requirements* key of the assignment config can also be formatted as a list of package names in lieu of a path to a *requirements.txt* file; for exmaple:

```
requirements:
    - pandas
    - numpy
    - scipy
```

This structure is also compatible with the *overwrite_requirements* key.

By default, Otter's grading images uses Python 3.9. If you need a different version, you can specify one using the `python_version` config:

```
# ASSIGNMENT CONFIG
python_version: 3.10
```

Otter Generate

A note about Otter Generate: the `generate` key of the assignment config has two forms. If you just want to generate and require no additional arguments, set `generate: true` in the YAML (the default) and Otter Assign will simply run `otter generate` from the autograder directory (this will also include any files passed to `files`, whose paths should be **relative to the directory containing the notebook**, not to the directory of execution). If you require additional arguments, e.g. `points` or `show_stdout`, then set `generate` to a nested dictionary of these parameters and their values:

```
generate:
    seed: 42
    show_stdout: true
    show_hidden: true
```

You can also set the autograder up to automatically upload PDFs to student submissions to another Gradescope assignment by setting the necessary keys under `generate`:

```
generate:
    token: YOUR_TOKEN       # optional
    course_id: 1234         # required
    assignment_id: 5678     # required
    filtering: true         # true is the default
```

You can run the following to retrieve your token:

```
from otter.generate.token import APIClient
print(APIClient.get_token())
```

If you don't specify a token, you will be prompted for your username and password when you run Otter Assign; optionally, you can specify these via the command line with the `--username` and `--password` flags.

Any configurations in your `generate` key will be put into an `otter_config.json` and used when running Otter Generate.

Log Grading

If you are grading from the log or would like to store students' environments in the log, use the `save_environment` key. If this key is set to `true`, Otter will serialize the stuednt's environment whenever a check is run, as described in *Logging*. To restrict the serialization of variables to specific names and types, use the `variables` key, which maps variable names to fully-qualified type strings. The `ignore_modules` key is used to ignore functions from specific modules. To turn on grading from the log on Gradescope, set `generate[grade_from_log]` to `true`. The configuration below turns on the serialization of environments, storing only variables of the name `df` that are pandas dataframes.

```
save_environment: true
variables:
    df: pandas.core.frame.DataFrame
```

Assignment Names

You can also configure assignments created with Otter Assign to ensure that students submit to the correct assignment by setting the `name` key in the assignment config. When this is set, Otter Assign adds the provided name to the notebook metadata and the autograder configuration zip file; this configures the autograder to fail if the student uploads a notebook with a different assignment name in the metadata.

```
# ASSIGNMENT CONFIG
name: hw01
```

You can find more information about how Otter performs assignment name verification *here*.

Intercell Seeding

Python assignments support *intercell seeding*, and there are two flavors of this. The first involves the use of a seed variable, and is configured in the assignment config; this allows you to use tools like `np.random.default_rng` instead of just `np.random.seed`. The second flavor involves comments in code cells, and is described *below*.

To use a seed variable, specify the name of the variable, the autograder seed value, and the student seed value in your assignment config.

```
# ASSIGNMENT CONFIG
seed:
    variable: rng_seed
    autograder_value: 42
    student_value: 713
```

With this type of seeding, you do not need to specify the seed inside the `generate` key; this automatically taken care of by Otter Assign.

Then, in a cell of your notebook, define the seed variable *with the autograder value*. This value needs to be defined in a separate cell from any of its uses and the variable name cannot be used for anything other than seeding RNGs. This is because it the variable will be redefined in the student's submission at the top of every cell. We recommend defining it in, for example, your imports cell.

```
import numpy as np
rng_seed = 42
```

To use the seed, just use the variable as normal:

```
rng = np.random.default_rng(rng_seed)
rvs = [rng.random() for _ in range(1000)] # SOLUTION
```

Or, in R:

```
set.seed(rng_seed)
runif(1000)
```

If you use this method of intercell seeding, the solutions notebook will contain the original value of the seed, but the student notebook will contain the student value:

```
# from the student notebook
import numpy as np
rng_seed = 713
```

When you do this, Otter Generate will be configured to overwrite the seed variable in each submission, allowing intercell seeding to function as normal.

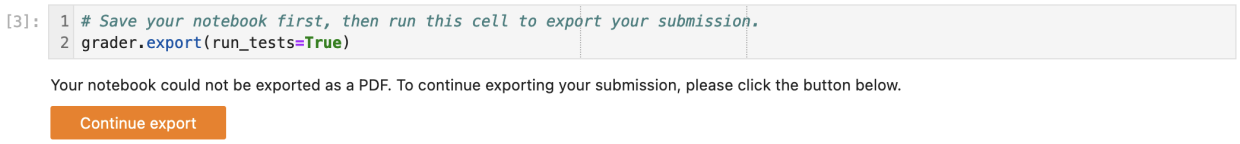
Remember that the student seed is different from the autograder seed, so any public tests cannot be deterministic otherwise they will fail on the student's machine. Also note that only one seed is available, so each RNG must use the same seed.

You can find more information about intercell seeding *here*.

Submission Export Cells

By default, Otter Assign includes cells to help students export their submission as a zip file that can be easily submitted and graded. To disable this feature, set `export_cell: false` in the assignment config.

These submission zip files include a PDF export of the notebook by default (this can be disabled with `export_cell: pdf: false`). In some cases, it may not be possible to export a PDF of the notebook (usually due to LaTeX errors), but the zip file may still be generated. Since this can occur without students realizing it, it is possible to have students acknowledge that their submission zip won't include a PDF before the zip file is generated. To require this acknowledgement, set `export_cell: require_no_pdf_ack: true` in the assignment config. If this is configured and the export cell fails to generate a PDF without raising an exception, the student will be presented with this acknowledgement built with ipywidgets:



To customize the message in the acknowledgement, set the `message` key of `require_no_pdf_ack`:

```
# ASSIGNMENT CONFIG
export_cell:
    require_no_pdf_ack:
        message: 'A PDF of your notebook could not be generated. Please acknowledge to␣
↪contiue submissione export.'
```

3.1.2 Autograded Questions

Here is an example question in an Otter Assign-formatted question:

Note the use of the delimiting raw cells and the placement of question config in the `# BEGIN QUESTION` cell. The question config can contain the following fields (in any order):

```
name: null        # (required) the path to a requirements.txt file
manual: false     # whether this is a manually-graded question
points: null      # how many points this question is worth; defaults to 1 internally
check_cell: true  # whether to include a check cell after this question (for autograded␣
↪questions only)
export: false     # whether to force-include this question in the exported PDF
```

As an example, the question config below indicates an autograded question `q1` that should be included in the filtered PDF.

```
# BEGIN QUESTION
name: q1
export: true
```

Solution Removal

Solution cells contain code formatted in such a way that the assign parser replaces lines or portions of lines with prespecified prompts. Otter uses the same solution replacement rules as jAssign. From the jAssign docs:

- A line ending in `# SOLUTION` will be replaced by `...` (or `NULL # YOUR CODE HERE` in R), properly indented. If that line is an assignment statement, then only the expression(s) after the = symbol (or the <- symbol in R) will be replaced.
- A line ending in `# SOLUTION NO PROMPT` or `# SEED` will be removed.
- A line `# BEGIN SOLUTION` or `# BEGIN SOLUTION NO PROMPT` must be paired with a later line `# END SOLUTION`. All lines in between are replaced with `...` (or `# YOUR CODE HERE` in R) or removed completely in the case of `NO PROMPT`.
- A line `""" # BEGIN PROMPT` must be paired with a later line `""" # END PROMPT`. The contents of this multiline string (excluding the `# BEGIN PROMPT`) appears in the student cell. Single or double quotes are allowed. Optionally, a semicolon can be used to suppress output: `"""; # END PROMPT`

```
def square(x):
    y = x * x # SOLUTION NO PROMPT
    return y # SOLUTION

nine = square(3) # SOLUTION
```

would be presented to students as

```
def square(x):
    ...

nine = ...
```

And

```
pi = 3.14
if True:
    # BEGIN SOLUTION
    radius = 3
    area = radius * pi * pi
    # END SOLUTION
    print('A circle with radius', radius, 'has area', area)

def circumference(r):
    # BEGIN SOLUTION NO PROMPT
    return 2 * pi * r
    # END SOLUTION
    """ # BEGIN PROMPT
    # Next, define a circumference function.
    pass
    """; # END PROMPT
```

would be presented to students as

```
pi = 3.14
if True:
    ...
    print('A circle with radius', radius, 'has area', area)

def circumference(r):
    # Next, define a circumference function.
    pass
```

For R,

```
# BEGIN SOLUTION
square <- function(x) {
    return(x ^ 2)
}
# END SOLUTION
x2 <- square(25)
```

would be presented to students as

```
...
x2 <- square(25)
```

Test Cells

Any cells within the `# BEGIN TESTS` and `# END TESTS` boundary cells are considered test cells. Each test cell corresponds to a single test case. There are two types of tests: public and hidden tests. Tests are public by default but can be hidden by adding the `# HIDDEN` comment as the first line of the cell. A hidden test is not distributed to students, but is used for scoring their work.

Test cells also support test case-level metadata. If your test requires metadata beyond whether the test is hidden or not, specify the test by including a mutliline string at the top of the cell that includes YAML-formatted test config. For example,

```
""" # BEGIN TEST CONFIG
points: 1
success_message: Good job!
""" # END TEST CONFIG
...  # your test goes here
```

The test config supports the following keys with the defaults specified below:

```
hidden: false          # whether the test is hidden
points: null           # the point value of the test
success_message: null  # a messsge to show to the student when the test case passes
failure_message: null  # a messsge to show to the student when the test case fails
```

Because points can be specified at the question level and at the test case level, Otter will resolve the point value of each test case as described *here*.

If a question has no solution cell provided, the question will either be removed from the output notebook entirely if it has only hidden tests or will be replaced with an unprompted `Notebook.check` cell that runs those tests. In either case, the test files are written, but this provides a way of defining additional test cases that do not have public versions. Note, however, that the lack of a `Notebook.check` cell for questions with only hidden tests means that the tests are run *at the end of execution*, and therefore are not robust to variable name collisions.

Because Otter supports two different types of test files, test cells can be written in two different ways.

OK-Formatted Test Cells

To use OK-formatted tests, which are the default for Otter Assign, you can write the test code in a test cell; Otter Assign will parse the output of the cell to write a doctest for the question, which will be used for the test case. **Make sure that only the last line of the cell produces any output, otherwise the test will fail.**

Exception-Based Test Cells

To use Otter's exception-based tests, you must set `tests: ok_format: false` in your assignment config. Your test cells should define a test case function as described *here*. You can run the test in the master notebook by calling the function, but you should make sure that this call is "ignored" by Otter Assign so that it's not included in the test file by appending `# IGNORE` to the end of line. You should *not* add the `test_case` decorator; Otter Assign will do this for you.

For example,

```
""" # BEGIN TEST CONFIG
points: 0.5
""" # END TEST CONFIG
def test_validity(arr):
    assert len(arr) == 10
    assert (0 <= arr <= 1).all()

test_validity(arr)  # IGNORE
```

It is important to note that the exception-based test files are executed before the student's global environment is provided, so no work should be performed outside the test case function that relies on student code, and any libraries or other variables declared in the student's environment must be passed in as arguments, otherwise the test will fail.

For example,

```
def test_values(arr):
    assert np.allclose(arr, [1.2, 3.4, 5.6])  # this will fail, because np is not in the␣
↪test file

def test_values(np, arr):
    assert np.allclose(arr, [1.2, 3.4, 5.6])  # this works

def test_values(env):
    assert env["np"].allclose(env["arr"], [1.2, 3.4, 5.6])  # this also works
```

R Test Cells

Test cells in R notebooks are like a cross between exception-based test cells and OK-formatted test cells: the checks in the cell do not need to be wrapped in a function, but the passing or failing of the test is determined by whether it raises an error, not by checking the output. For example,

```
. = " # BEGIN TEST CONFIG
hidden: true
points: 1
" # END TEST CONFIG
testthat::expect_equal(sieve(3), c(2, 3))
```

Intercell Seeding

The second flavor of intercell seeding involves writing a line that ends with `# SEED`; when Otter Assign runs, this line will be removed from the student version of the notebook. This allows instructors to write code with deterministic output, with which hidden tests can be generated.

For example, the first line of the cell below would be removed in the student version of the notebook.

```
np.random.seed(42) # SEED
rvs = [np.random.random() for _ in range(1000)] # SOLUTION
```

The same caveats apply for this type of seeding as *above*.

R Example

Here is an example autograded question for R:

3.1.3 Manually-Graded Questions

Otter Assign also supports manually-graded questions using a similar specification to the one described above. To indicate a manually-graded question, set `manual: true` in the question config.

A manually-graded question can have an optional prompt block and a required solution block. If the solution has any code cells, they will have their syntax transformed by the solution removal rules listed above.

If there is a prompt for manually-graded questions, then this prompt is included unchanged in the output. If none is present, Otter Assign automatically adds a Markdown cell with the contents `_Type your answer here, replacing this text._` if the solution block has any Markdown cells in it.

Here is an example of a manually-graded code question:

Manually graded questions are automatically enclosed in `<!-- BEGIN QUESTION -->` and `<!-- END QUESTION -->` tags by Otter Assign so that only these questions are exported to the PDF when filtering is turned on (the default). In the autograder notebook, this includes the question cell, prompt cell, and solution cell. In the student notebook, this includes only the question and prompt cells. The `<!-- END QUESTION -->` tag is automatically inserted at the top of the next cell if it is a Markdown cell or in a new Markdown cell before the next cell if it is not.

3.1.4 Ignoring Cells

For any cells that you don't want to be included in *either* of the output notebooks that are present in the master notebook, include a line at the top of the cell with the `## Ignore ##` comment (case insensitive) just like with test cells. Note that this also works for Markdown cells with the same syntax.

```
## Ignore ##
print("This cell won't appear in the output.")
```

3.1.5 Student-Facing Plugins

Otter supports student-facing plugin events via the `otter.Notebook.run_plugin` method. To include a student-facing plugin call in the resulting versions of your master notebook, add a multiline plugin config string to a code cell of your choosing. The plugin config should be YAML-formatted as a mutliline comment-delimited string, similar to the solution and prompt blocks above. The comments `# BEGIN PLUGIN` and `# END PLUGIN` should be used on the lines with the triple-quotes to delimit the YAML's boundaries. There is one required configuration: the plugin name, which should be a fully-qualified importable string that evaluates to a plugin that inherits from `otter.plugins.AbstractOtterPlugin`.

There are two optional configurations: `args` and `kwargs`. `args` should be a list of additional arguments to pass to the plugin. These will be left unquoted as-is, so you can pass variables in the notebook to the plugin just by listing them. `kwargs` should be a dictionary that mappins keyword argument names to values; thse will also be added to the call in `key=value` format.

Here is an example of plugin replacement in Otter Assign:

Note that student-facing plugins are not supported with R assignments.

3.1.6 Running on Non-standard Python Environments

For non-standard Python notebook environments (which use their own interpreters, such as Colab or Jupyterlite), some Otter features are disabled and the the notebooks that are produced for running on those environments are slightly different. To indicate that the notebook produce by Otter Assign is going to be run in such an environment, use the `runs_on` assignment configuration. It currently supports these values:

- `default`, indicating a normal IPython environment (the default value)
- `colab`, indicating that the notebook will be used on Google Colab
- `jupyterlite`, indicating that the notebook will be used on Jupyterlite (or any environment using the Pyolite kernel)

3.1.7 Alternative to Raw Cells

If you're authoring your notebooks in an environment where raw cells are not supported (such as Deepnote or Google Colab), all of the places where Otter requires raw cells can be exchanged for normal Markdown cells by wrapping the cell's contents in a code block with the language set to `otter`. For example, an assignment configuration cell would look like

````
```otter
# ASSIGNMENT CONFIG
...
```
````

There should be nothing else in the Markdown cell.

3.1.8 Sample Notebook

You can find a sample Python notebook here.

3.2 R Markdown Format

Otter Assign is compatible with Otter's R autograding system and currently supports Jupyter notebook and R Markdown master documents. The format for using Otter Assign with R is very similar to the Python format with a few important differences. The main difference is that, where in the notebook format you would use raw cells, in R Markdown files you wrap what would normally be in a raw cell in an HTML comment.

For example, a `# BEGIN TESTS` cell in R Markdown looks like:

```
<!-- # BEGIN TESTS -->
```

For cells that contain YAML configurations, you can use a multiline comment:

```
<!--
# BEGIN QUESTION
name: q1
-->
```

3.2.1 Assignment Config

As with Python, Otter Assign for R Markdown also allows you to specify various assignment generation arguments in an assignment config comment:

```
<!--
# ASSIGNMENT CONFIG
init_cell: false
export_cell: true
generate: true
# etc.
-->
```

You can find a list of available metadata keys and their defaults in the *notebook format section*.

3.2.2 Autograded Questions

Here is an example question in an Otter Assign-formatted R Markdown question:

````
<!--
# BEGIN QUESTION
name: q1
manual: false
points:
    - 1
    - 1
-->

**Question 1:** Find the radius of a circle that has a 90 deg. arc of length 2. Assign␣
↪this
value to `ans.1`

<!-- # BEGIN SOLUTION -->

```{r}
ans.1 <- 2 * 2 * pi * 2 / pi / pi / 2 # SOLUTION
```

<!-- # END SOLUTION -->

<!-- # BEGIN TESTS -->

```{r}
expect_true(ans.1 > 1)
expect_true(ans.1 < 2)
```

```{r}
# HIDDEN
tol = 1e-5
actual_answer = 1.27324
expect_true(ans.1 > actual_answer - tol)
expect_true(ans.1 < actual_answer + tol)
```

<!-- # END TESTS -->

<!-- # END QUESTION -->
````

For code questions, a question is a some description markup, followed by a solution code blocks and zero or more test code blocks. The blocks should be wrapped in HTML comments following the same structure as the notebook autograded question. The question config has the same keys as the notebook question config.

As an example, the question config below indicates an autograded question `q1` with 3 subparts worth 1, 2, and 1 points, resp.

```
<!--
# BEGIN QUESTION
name: q1
```

(continues on next page)

(continued from previous page)

```
points:
    - 1
    - 2
    - 1
-->
```

Solution Removal

Solution cells contain code formatted in such a way that the assign parser replaces lines or portions of lines with pre-specified prompts. The format for solution cells in Rmd files is the same as in Python and R Jupyter notebooks, described *here*. Otter Assign's solution removal for prompts is compatible with normal strings in R, including assigning these to a dummy variable so that there is no undesired output below the cell:

```
# this is OK:
. = " # BEGIN PROMPT
some.var <- ...
" # END PROMPT
```

Test Cells

Any cells within the `# BEGIN TESTS` and `# END TESTS` boundary cells are considered test cells. There are two types of tests: public and hidden tests. Tests are public by default but can be hidden by adding the `# HIDDEN` comment as the first line of the cell. A hidden test is not distributed to students, but is used for scoring their work.

When writing tests, each test cell maps to a single test case and should raise an error if the test fails. The removal behavior regarding questions with no solution provided holds for R Markdown files.

```
testthat::expect_true(some_bool)
```

```
testthat::expect_equal(some_value, 1.04)
```

As with notebooks, test cells also support test config blocks; for more information on these, see *R Test Cells*.

3.2.3 Manually-Graded Questions

Otter Assign also supports manually-graded questions using a similar specification to the one described above. To indicate a manually-graded question, set `manual: true` in the question config. A manually-graded question is defined by three parts:

- a question config
- (optionally) a prompt
- a solution

Manually-graded solution cells have two formats:

- If the response is code (e.g. making a plot), they can be delimited by solution removal syntax as above.
- If the response is markup, the the solution should be wrapped in special HTML comments (see below) to indicate removal in the sanitized version.

To delimit a markup solution to a manual question, wrap the solution in the HTML comments `<!-- # BEGIN SOLUTION -->` and `<!-- # END SOLUTION -->` on their own lines to indicate that the content in between should be removed.

```
<!-- # BEGIN SOLUTION -->

solution goes here

<!-- # END SOLUTION -->
```

To use a custom Markdown prompt, include a `<!-- # BEGIN/END PROMPT -->` block with a solution block:

```
<!-- # BEGIN PROMPT -->

prompt goes here

<!-- # END PROMPT -->

<!-- # BEGIN SOLUTION -->

solution goes here

<!-- # END SOLUTION -->
```

If no prompt is provided, Otter Assign automatically replaces the solution with a line containing `_Type your answer here, replacing this text._`.

An example of a manually-graded code question:

````
<!--
# BEGIN QUESTION
name: q7
manual: true
-->

**Question 7:** Plot $f(x) = \cos e^x$ on $[0,10]$.

<!-- # BEGIN SOLUTION -->

```{r}
# BEGIN SOLUTION
x = seq(0, 10, 0.01)
y = cos(exp(x))
ggplot(data.frame(x, y), aes(x=x, y=y)) +
    geom_line()
# END SOLUTION
```

<!-- # END SOLUTION -->

<!-- # END QUESTION -->
````

An example of a manually-graded written question (with no prompt):

```
<!--
# BEGIN QUESTION
name: q5
manual: true
-->

**Question 5:** Simplify $\sum_{i=1}^n n$.

<!-- # BEGIN SOLUTION -->

$\frac{n(n+1)}{2}$

<!-- # END SOLUTION -->

<!-- # END QUESTION -->
```

An example of a manually-graded written question with a custom prompt:

```
<!--
# BEGIN QUESTION
name: q6
manual: true
-->

**Question 6:** Fill in the blank.

<!-- # BEGIN PROMPT -->

The mitochondria is the _____________ of the cell.

<!-- # END PROMPT -->

<!-- # BEGIN SOLUTION-->

powerhouse

<!-- # END SOLUTION -->

<!-- # END QUESTION -->
```

3.3 Usage and Output

Otter Assign is called using the `otter assign` command. This command takes in two required arguments. The first is `master`, the path to the master notebook (the one formatted as described above), and the second is `result`, the path at which output shoud be written. **Otter Assign will automatically recognize the language of the notebook** by looking at the kernel metadata; similarly, if using an Rmd file, it will automatically choose the language as R. This behavior can be overridden using the `-l` flag which takes the name of the language as its argument.

The default behavior of Otter Assign is to do the following:

1. Filter test cells from the master notebook and create test objects from these
2. Add Otter initialization, export, and `Notebook.check_all` cells

3. Clear outputs and write questions (with metadata hidden), prompts, and solutions to a notebook in a new `autograder` directory
4. Write *all* tests to the notebook metadata of the `autograder` notebook
5. Copy autograder notebook with solutions removed into a new `student` directory
6. Remove all hidden tests from the notebook in the `student` directory``
7. Copy `files` into `autograder` and `student` directories
8. Generate a Gradescope autograder zipfile from the `autograder` directory
9. Run all tests in `autograder/tests` on the solutions notebook to ensure they pass

These behaviors can be customized using command line flags; see the *CLI Reference* for more information.

An important note: make sure that you *run all cells* in the master notebook and save it *with the outputs* so that Otter Assign can generate the test files based on these outputs. The outputs will be cleared in the copies generated by Otter Assign.

3.3.1 Export Formats and Flags

By default, Otter Assign adds an initialization cell at the *top* of the notebook with the contents

```
# Initialize Otter
import otter
grader = otter.Notebook()
```

To prevent this behavior, add the `init_cell: false` configuration in your assignment metadata.

Otter Assign also automatically adds a check-all cell and an export cell to the end of the notebook. The check-all cells consist of a Markdown cell:

```
To double-check your work, the cell below will rerun all of the autograder tests.
```

and a code cell that calls `otter.Notebook.check_all`:

```
grader.check_all()
```

The export cells consist of a Markdown cell:

```
## Submission

Make sure you have run all cells in your notebook in order before running the cell below,
↪ so
that all images/graphs appear in the output. **Please save before submitting!**
```

and a code cell that calls `otter.Notebook.export` with HTML comment filtering:

```
# Save your notebook first, then run this cell to export.
grader.export("/path/to/notebook.ipynb")
```

For R assignments, the export cell looks like:

```
# Save your notebook first, then run this cell to export.
ottr::export("/path/to/notebook.ipynb")
```

These behaviors can be changed with the corresponding assignment metadata configurations.

Note: Otter Assign currently only supports *HTML comment filtering*. This means that if you have other cells you want included in the export, you must delimit them using HTML comments, not using cell tags.

3.3.2 Otter Assign Example

Consider the directory stucture below, where `hw00/hw00.ipynb` is an Otter Assign-formatted notebook.

```
hw00
├── data.csv
└── hw00.ipynb
```

To generate the distribution versions of `hw00.ipynb` (after changing into the `hw00` directory), you would run

```
otter assign hw00.ipynb dist
```

If it was an Rmd file instead, you would run

```
otter assign hw00.Rmd dist
```

This will create a new folder called `dist` with `autograder` and `student` as subdirectories, as described above.

```
hw00
├── data.csv
├── dist
│   ├── autograder
│   │   ├── hw00.ipynb
│   │   └── tests
│   │       ├── q1.(py|R)
│   │       └── q2.(py|R)  # etc.
│   └── student
│       ├── hw00.ipynb
│       └── tests
│           ├── q1.(py|R)
│           └── q2.(py|R)  # etc.
└── hw00.ipynb
```

In generating the distribution versions, you can prevent Otter Assign from rerunning the tests using the `--no-run-tests` flag:

```
otter assign --no-run-tests hw00.ipynb dist
```

Otter ships with an assignment development and distribution tool called Otter Assign, an Otter-compliant fork of jAssign that was designed for OkPy. Otter Assign allows instructors to create assignments by writing questions, prompts, solutions, and public and private tests all in a single notebook, which is then parsed and broken down into student and autograder versions.

CHAPTER

FOUR

STUDENT USAGE

4.1 Otter Configuration Files

In many use cases, the use of the `otter.Notebook` class by students requires some configurations to be set. These configurations are stored in a simple JSON-formatted file ending in the `.otter` extension. When `otter.Notebook` is instantiated, it globs all `.otter` files in the working directory and, if any are present, asserts that there is only 1 and loads this as the configuration. If no `.otter` config is found, it is assumed that there is no configuration file and, therefore, only features that don't require a config file are available.

The available keys in a `.otter` file are listed below, along with their default values. The only required key is `notebook` (i.e. if you use `.otter` file it must specify a value for `notebook`).

```
{
    "notebook": "",             # the notebook filename
    "save_environment": false, # whether to serialize the environment in the log during␣
↪checks
    "ignore_modules": [],       # a list of modules whose functions to ignore during␣
↪serialization
    "variables": {}             # a mapping of variable names -> types to resitrct during␣
↪serialization
}
```

4.1.1 Configuring the Serialization of Environments

If you are using logs to grade assignemtns from serialized environments, the `save_environment` key must be set to `true`. Any module names in the `ignore_modules` list will have their functions ignored during serialization. If `variables` is specified, it should be a dictionary mapping variables names to fully-qualified types; variables will only be serialized if they are present as keys in this dictionary and their type matches the specified type. An example of this use would be:

```
{
    "notebook": "hw00.ipynb",
    "save_environment": true,
    "variables": {
        "df": "pandas.core.frame.DataFrame",
        "fn": "builtins.function",
        "arr": "numpy.ndarray"
    }
}
```

The function `otter.utils.get_variable_type` when called on an object will return this fully-qualified type string.

```
In [1]: from otter.utils import get_variable_type

In [2]: import numpy as np

In [3]: import pandas as pd

In [4]: get_variable_type(np.array([]))
Out[4]: 'numpy.ndarray'

In [5]: get_variable_type(pd.DataFrame())
Out[5]: 'pandas.core.frame.DataFrame'

In [6]: fn = lambda x: x

In [7]: get_variable_type(fn)
Out[7]: 'builtins.function'
```

More information about grading from serialized environments can be found in *Logging*.

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

4.2 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 question 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]:
```
square = lambda x: x**3
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.

4.2.1 Exporting Submissions

Students can also use the `Notebook` class to generate a zip file containing all of their work for submission with the method `Notebook.export`. This function takes an optional argument of the path to the notebook; if unspecified, it will infer the path by trying to read the config file (if present), using the path of the only notebook in the working directory if there is only one, or it will raise an error telling you to provide the path. This method creates a submission zip file that includes the notebook file, the log, and, optionally, a PDF of the notebook (set `pdf=False` to disable this last).

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

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

as filtering is by defult on. If I instead wanted no filtering, I would use

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

To generate just a PDF of the notebook, use `Notebook.to_pdf`.

Both of these methods support force-saving the notebook before exporting it, so that any unsaved changes a student has made to their notebook will be reflected in the exported version. In Python, this works by using *ipylab* to communicate with the JupyterLab frontend. To use it, set `force_save=True`:

```
grader.export("hw01.ipynb", force_save=True)
```

In R, you must install the `ottr_force_save_labextension` Python package. This JupyterLab extension exposes a hook that `ottr::export` uses by running JavaScript to save the notebook.

```
ottr::export("hw01.ipynb", force_save=TRUE)
```

Force saving is not supported for Rmd files, and the argument is ignored if used when not running on Jupyter.

4.2.2 Running on Non-standard Python Environments

When running on non-standard Python notebook environments (which use their own interpreters, such as Colab or Jupyterlite), some Otter features are disabled due differences in file system access, the unavailability of compatible versions of packages, etc. When you instantiate a `Notebook`, Otter automatically tries to determine if you're running on one of these environments, but you can manually indicate which you're running on by setting either the `colab` or `jupyterlite` argument to `True`.

4.3 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 three 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.
- `--seed` is an optional random seed for *execution seeding*

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

```
hw00
├── hw00.py
└── tests
    ├── q1.py
    └── q2.py  # etc.
```

After a `cd` 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:
    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
```

(continues on next page)

(continued from previous page)

```
    ├── q1.py
    └── q2.py  # etc.
```

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
```

4.4 `otter.Notebook` Reference

class otter.check.notebook.**Notebook**(*nb_path=None*, *tests_dir='./tests'*, *tests_url_prefix=None*, *colab=None*, *jupyterlite=None*)

Notebook class for in-notebook autograding

Parameters

- **nb_path** (str | None) – path to the notebook being run
- **tests_dir** (str) – path to tests directory
- **tests_url_prefix** (str | None) – a URL prefix to use to download test files
- **colab** (bool | None) – whether this notebook is being run on Google Colab; if None, this information is automatically parsed from IPython on creation
- **jupyterlite** (bool | None) – whether this notebook is being run on JupyterLite; if None, this information is automatically parsed from IPython on creation

add_plugin_files(*plugin_name*, **args*, *nb_path=None*, ***kwargs*)

Runs the `notebook_export` event of the plugin `plugin_name` and tracks the file paths it returns to be included when calling `Notebook.export`.

Parameters

- **plugin_name** (str) – importable name of an Otter plugin that implements the `from_notebook` hook
- ***args** – arguments to be passed to the plugin
- **nb_path** (str | None) – path to the notebook
- ****kwargs** – keyword arguments to be passed to the plugin

check(*question*, *global_env=None*)

Runs tests for a specific question against a global environment. If no global environment is provided, the test is run against the calling frame's environment.

Parameters

- **question** (str) – name of question being graded
- **global_env** (dict[str, object] | None) – a global environment in which to run the tests

Returns
the grade for the question

Return type
otter.test_files.abstract_test.TestFile

check_all()

Runs all tests on this notebook. Tests are run against the current global environment, so any tests with variable name collisions will fail.

export(*nb_path=None*, *export_path=None*, *pdf=True*, *filtering=True*, *pagebreaks=True*, *files=None*, *display_link=True*, *force_save=False*, *run_tests=False*)

Export a submission zip file.

Creates a submission zipfile from a notebook at nb_path, optionally including a PDF export of the notebook and any files in files.

If run_tests is true, the zip is validated by running it against local test cases in a separate Python process. The results of this run are printed to stdout.

Parameters

- **nb_path** (str | None) – path to the notebook we want to export; will attempt to infer if not provided
- **export_path** (str | None) – path at which to write zipfile; defaults to {notebook name}_{timestamp}.zip
- **pdf** (bool) – whether a PDF should be included
- **filtering** (bool) – whether the PDF should be filtered
- **pagebreaks** (bool) – whether pagebreaks should be included between questions in the PDF
- **files** (list[str] | None) – paths to other files to include in the zip file
- **display_link** (bool) – whether or not to display a download link
- **force_save** (bool) – whether or not to display JavaScript that force-saves the notebook (only works in Jupyter Notebook classic, not JupyterLab)
- **run_tests** (bool) – whether to validating the resulting submission zip against local test cases

run_plugin(*plugin_name*, **args*, *nb_path=None*, ***kwargs*)

Runs the plugin plugin_name with the specified arguments. Use nb_path if the path to the notebook is not configured.

Parameters

- **plugin_name** (str) – importable name of an Otter plugin that implements the from_notebook hook

- ***args** – arguments to be passed to the plugin
- **nb_path** (str, optional) – path to the notebook
- ****kwargs** – keyword arguments to be passed to the plugin

to_pdf(*nb_path=None*, *filtering=True*, *pagebreaks=True*, *display_link=True*, *force_save=False*)

Exports a notebook to a PDF using Otter Export

Parameters

- **nb_path** (str | None) – path to the notebook we want to export; will attempt to infer if not provided
- **filtering** (bool) – set true if only exporting a subset of notebook cells to PDF
- **pagebreaks** (bool) – if true, pagebreaks are included between questions
- **display_link** (bool) – whether or not to display a download link
- **force_save** (bool) – whether or not to display JavaScript that force-saves the notebook (only works in Jupyter Notebook classic, not JupyterLab)

CHAPTER

FIVE

GRADING WORKFLOW

5.1 Generating Configuration Files

5.1.1 Grading Container Image

When you use Otter Generate to create the configuration zip file for Gradescope, Otter includes the following software and packages for installation. The zip archive, when unzipped, has the following contents:

```
autograder
├── environment.yml
├── files/
├── otter_config.json
├── requirements.r
├── run_autograder
├── run_otter.py
├── setup.sh
└── tests/
```

Note that for pure-Python assignments, `requirements.r` is not included and all of the R-pertinent portions of `setup.sh` are removed. Below are descriptions of each of the items listed above and the Jinja2 templates used to create them (if applicable).

`setup.sh`

This file, required by Gradescope, performs the installation of necessary software for the autograder to run.

The template for Python assignments is:

```
#!/usr/bin/env bash

export DEBIAN_FRONTEND=noninteractive
apt-get clean
apt-get update
apt-get install -y wget texlive-xetex texlive-fonts-recommended texlive-plain-generic \
    build-essential libcurl4-gnutls-dev libxml2-dev libssl-dev libgit2-dev texlive-lang-
↪chinese

# install pandoc
wget -nv https://github.com/jgm/pandoc/releases/download/3.1.11.1/pandoc-3.1.11.1-1-
↪amd64.deb \
```

(continues on next page)

(continued from previous page)

```
    -O /tmp/pandoc.deb
dpkg -i /tmp/pandoc.deb

# install mamba
if [ $(uname -p) = "arm" ] || [ $(uname -p) = "aarch64" ] ; \
    then wget -nv https://github.com/conda-forge/miniforge/releases/latest/download/
↪Mambaforge-Linux-aarch64.sh \
        -O /autograder/source/mamba_install.sh ; \
    else wget -nv https://github.com/conda-forge/miniforge/releases/latest/download/
↪Mambaforge-Linux-x86_64.sh \
        -O /autograder/source/mamba_install.sh ; \
fi
chmod +x /autograder/source/mamba_install.sh
/autograder/source/mamba_install.sh -b
echo "export PATH=/root/mambaforge/bin:\$PATH" >> /root/.bashrc

export PATH=/root/mambaforge/bin:$PATH
export TAR="/bin/tar"

# install dependencies with mamba
mamba env create -f /autograder/source/environment.yml

# set mamba shell
mamba init --all
```

And the template for R assignments is:

```
#!/usr/bin/env bash

export DEBIAN_FRONTEND=noninteractive
apt-get clean
apt-get update
apt-get install -y wget texlive-xetex texlive-fonts-recommended texlive-plain-generic \
    build-essential libcurl4-gnutls-dev libxml2-dev libssl-dev libgit2-dev texlive-lang-
↪chinese
apt-get install -y libnlopt-dev cmake libfreetype6-dev libpng-dev libtiff5-dev libjpeg-
↪dev \
    apt-utils libpoppler-cpp-dev libavfilter-dev  libharfbuzz-dev libfribidi-dev␣
↪imagemagick \
    libmagick++-dev texlive-xetex texlive-fonts-recommended texlive-plain-generic \
    build-essential libcurl4-gnutls-dev libxml2-dev libssl-dev libgit2-dev texlive-lang-
↪chinese \
    libxft-dev

# install pandoc
wget -nv https://github.com/jgm/pandoc/releases/download/3.1.11.1/pandoc-3.1.11.1-1-
↪amd64.deb \
    -O /tmp/pandoc.deb
dpkg -i /tmp/pandoc.deb

# install mamba
if [ $(uname -p) = "arm" ] || [ $(uname -p) = "aarch64" ] ; \
```

(continues on next page)

(continued from previous page)

```
    then wget -nv https://github.com/conda-forge/miniforge/releases/latest/download/
↪Mambaforge-Linux-aarch64.sh \
        -O /autograder/source/mamba_install.sh ; \
    else wget -nv https://github.com/conda-forge/miniforge/releases/latest/download/
↪Mambaforge-Linux-x86_64.sh \
        -O /autograder/source/mamba_install.sh ; \
fi
chmod +x /autograder/source/mamba_install.sh
/autograder/source/mamba_install.sh -b
echo "export PATH=/root/mambaforge/bin:\$PATH" >> /root/.bashrc

export PATH=/root/mambaforge/bin:$PATH
export TAR="/bin/tar"

# install dependencies with mamba
mamba env create -f /autograder/source/environment.yml
mamba run -n otter-env Rscript /autograder/source/requirements.r

# set mamba shell
mamba init --all
```

Note that the line `mamba run -n otter-env Rscript /autograder/source/requirements.r` is only included if you have provided an *R requirements file*.

environment.yml

This file specifies the conda environment that Otter creates in `setup.sh`. By default, it uses Python 3.9, but this can be changed using the `--python-version` flag to Otter Generate.

```
name: otter-env
channels:
  - defaults
  - conda-forge
dependencies:
  - python=3.9
  - pip
  - nb_conda_kernels
  - pip:
      - datascience
      - jupyter_client
      - ipykernel
      - matplotlib
      - pandas
      - ipywidgets
      - scipy
      - seaborn
      - scikit-learn
      - jinja2
      - nbconvert
      - nbformat
      - dill
```

(continues on next page)

(continued from previous page)

```
      - numpy
      - gspread
      - pypdf
      - otter-grader==5.5.0
```

If you're grading a Python assignment, any dependencies in your `requirements.txt` will be added to the `pip` list in this file. If you pass `--overwrite-requirements`, your `requirements.txt` contents will be in the `pip` list instead of what's above.

If you're grading an R assignment, the `environment.yml` has additional depdencies:

```
name: otter-env
channels:
  - defaults
  - conda-forge
  - r
dependencies:
  - python=3.9
  - pip
  - nb_conda_kernels
  - r-base>=4.0.0
  - r-essentials
  - r-devtools
  - libgit2
  - libgomp
  - r-gert
  - r-usethis
  - r-testthat
  - r-startup
  - r-rmarkdown
  - r-stringi
  - r-ottr==1.5.0
  - pip:
      - datascience
      - jupyter_client
      - ipykernel
      - matplotlib
      - pandas
      - ipywidgets
      - scipy
      - seaborn
      - scikit-learn
      - jinja2
      - nbconvert
      - nbformat
      - dill
      - numpy
      - gspread
      - pypdf
      - otter-grader==5.5.0
      - rpy2
```

requirements.r

If you're grading an R assignment, this file will be included if you specify it in the `--requirements` argument, and is just a copy of the file you provide. It should use functions like `install.packages` to install any additional dependencies your assignment requires. (Alternatively, if the depdendencies are available via conda, you can specify them in an `environment.yml`.)

run_autograder

This is the file that Gradescope uses to actually run the autograder when a student submits. Otter provides this file as an executable that activates the conda environment and then calls `/autograder/source/run_otter.py`:

```
#!/usr/bin/env bash
export PATH="/root/mambaforge/bin:$PATH"
source /root/mambaforge/etc/profile.d/conda.sh
source /root/mambaforge/etc/profile.d/mamba.sh
mamba activate {{ otter_env_name }}
python {{ autograder_dir }}/source/run_otter.py
```

run_otter.py

This file contains the logic to start the grading process by importing and running `otter.run.run_autograder.main`:

```
"""Runs Otter-Grader's autograding process"""

from otter.run.run_autograder import main as run_autograder

if __name__ == "__main__":
    run_autograder("{{ autograder_dir }}")
```

otter_config.json

This file contains any user configurations for grading. It has no template but is populated with the any non-default values you specify for these configurations. When debugging grading via SSH on Gradescope, a helpful tip is to set the `debug` key of this JSON file to `true`; this will stop the autograding from ignoring errors when running students' code, and can be helpful in debugging specific submission issues.

tests

This is a directory containing the test files that you provide. All `.py` (or `.R`) files in the tests directory path that you provide are copied into this directory and are made available to submissions when the autograder runs.

files

This directory, not present in all autograder zip files, contains any support files that you provide to be made available to submissions.

This section details how to generate the configuration files needed for preparing Otter autograders. Note that this step can be accomplished automatically if you're using *Otter Assign*.

To use Otter to autograde an assignment, you must first generate a zip file that Otter will use to create a Docker image with which to grade submissions. Otter's command line utility `otter generate` allows instructors to create this zip file from their machines.

5.1.2 Before Using Otter Generate

Before using Otter Generate, you should already have written *tests* for the assignment and collected extra requirements into a `requirements.txt` file (see *here*). (Note: the default requirements can be overwritten by your requirements by passing the `--overwrite-requirements` flag.)

5.1.3 Directory Structure

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

```
hw00-dev
├── data.csv
├── hidden-tests
│   ├── q1.py
│   └── q2.py  # etc.
├── hw00-sol.ipynb
├── hw00.ipynb
├── requirements.txt
├── tests
│   ├── q1.py
│   └── q2.py  # etc.
└── utils.py
```

Also assume that the working directory is `hw00-dev`.

5.1.4 Usage

The general usage of `otter generate` is to create a zip file at some output directory (`-o` flag, default `./`) which you will then use to create the grading image. Otter Generate has a few optional flags, described in the *CLI Reference*.

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. If you do not specify `-e`, Otter will use the default `environment.yml`. 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). To autograde an R assignment, pass the `-l r` flag to indicate that the language of the assignment is R.

The simplest usage in our example would be

```
otter generate
```

This would create a zip file `autograder.zip` 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 generate 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 generate -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 Generate call:

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

If this was instead an R assignment, I would run

```
otter generate -t hidden-tests -l r data.csv
```

5.1.5 Grading Configurations

There are several configurable behaviors that Otter supports during grading. Each has default values, but these can be configured by creating an Otter config JSON file and passing the path to this file to the `-c` flag (`./otter_config.json` is automatically added if found and `-c` is unspecified).

The supported keys and their default values are configured in a fica configuration class (`otter.run.run_autograder.autograder_config.AutograderConfig`). The available configurations are documented below.

```
{
  "score_threshold": null,                    // a score threshold for pass-fail␣
↪assignments
  "points_possible": null,                    // a custom total score for the␣
↪assignment; if unspecified the sum of question point values is used.
  "show_stdout": false,                       // whether to display the autograding␣
↪process stdout to students on Gradescope
  "show_hidden": false,                       // whether to display the results of␣
↪hidden tests to students on Gradescope
  "show_all_public": false,                   // whether to display all test results␣
↪if all tests are public tests
  "seed": null,                               // a random seed for intercell seeding
  "seed_variable": null,                      // a variable name to override with the␣
↪seed
  "grade_from_log": false,                    // whether to re-assemble the student's␣
↪environment from the log rather than by re-executing their submission
  "serialized_variables": null,               // a mapping of variable names to type␣
↪strings for validating a deserialized student environment
  "pdf": false,                               // whether to generate a PDF of the␣
↪notebook when not using Gradescope auto-upload
  "token": null,                              // a Gradescope token for uploading a␣
↪PDF of the notebook
```

(continues on next page)

(continued from previous page)

```
  "course_id": null,                            // a Gradescope course ID for uploading␣
↪a PDF of the notebook
  "assignment_id": null,                        // a Gradescope assignment ID for␣
↪uploading a PDF of the notebook
  "filtering": false,                           // whether the generated PDF should have␣
↪cells filtered out
  "pagebreaks": false,                          // whether the generated PDF should have␣
↪pagebreaks between filtered sections
  "debug": false,                               // whether to run the autograder in␣
↪debug mode (without ignoring errors)
  "autograder_dir": "/autograder",              // the directory in which autograding is␣
↪taking place
  "lang": "python",                             // the language of the assignment; one␣
↪of {'python', 'r'}
  "miniconda_path": "/root/mambaforge",         // the path to the mamba install␣
↪directory
  "plugins": [],                                // a list of plugin names and␣
↪configuration details for grading
  "logo": true,                                 // whether to print the Otter logo to␣
↪stdout
  "print_summary": true,                        // whether to print the grading summary
  "print_score": true,                          // whether to print out the submission␣
↪score in the grading summary
  "zips": false,                                // whether zip files are being graded
  "log_level": null,                            // a log level for logging messages; any␣
↪value suitable for ``logging.Logger.setLevel``
  "assignment_name": null,                      // a name for the assignment to ensure␣
↪that students submit to the correct autograder
  "warn_missing_pdf": false,                    // whether to add a 0-point public test␣
↪to the Gradescope output to indicate to students whether a PDF was found/generated for␣
↪this assignment
  "force_public_test_summary": true,            // whether to show a summary of public␣
↪test case results when show_hidden is true
  "submit_blank_pdf_on_export_failure": false,  // whether to submit a blank PDF to the␣
↪manual-grading Gradescope assignment if a PDF cannot be generated from the submission
  "use_submission_pdf": false                   // use the PDF in the submission zip␣
↪file instead of exporting a new one; if no PDF is present, a new one is generated␣
↪anyway; assumes there is only 1 PDF file in the submission
}
```

Grading with Environments

Otter can grade assignments using saved environments in the log in the Gradescope container. *This behavior is not supported for R assignments.* This works by deserializing the environment stored in each check entry of Otter's log and grading against it. The notebook is parsed and only its import statements are executed. For more inforamtion about saving and using environments, see *Logging*.

To configure this behavior, two things are required:

- the use of the `grade_from_log` key in your config JSON file
- providing students with an *Otter configuration file* that has `save_environments` set to `true`

This will tell Otter to shelve the global environment each time a student calls `Notebook.check` (pruning the environments of old calls each time it is called on the same question). When the assignment is exported using `Notebook.export`, the log file (at `./.OTTER_LOG`) is also exported with the global environments. These environments are read in in the Gradescope container and are then used for grading. Because one environment is saved for each check call, variable name collisions can be averted, since each question is graded using the global environment at the time it was checked. Note that any requirements needed for execution need to be installed in the Gradescope container, because Otter's shelving mechanism does not store module objects.

Autosubmission of Notebook PDFs

Otter Generate allows instructors to automatically generate PDFs of students' notebooks and upload these as submissions to a separate Gradescope assignment. This requires a Gradescope token or entering your Gradescope account credentials when prompted (for details see the *notebook metadata section*). Otter Generate also needs the course ID and assignment ID of the assignment to which PDFs should be submitted – a separate assignment from your autograder assignment of type "Homework / Problem Set." This information can be gathered from the assignment URL on Gradescope:

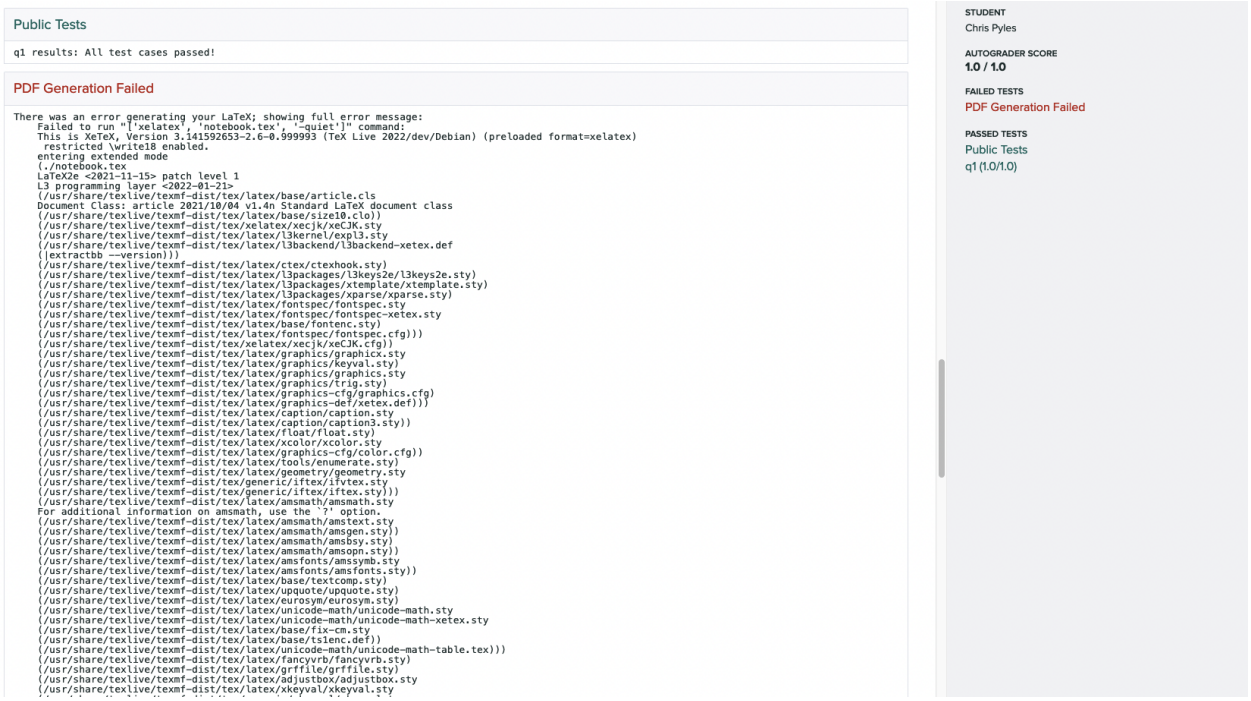
```
https://www.gradescope.com/courses/{COURSE ID}/assignments/{ASSIGNMENT ID}
```

To configure this behavior, set the `course_id` and `assignment_id` configurations in your config JSON file. When Otter Generate is run, you will be prompted to enter your Gradescope email and password. Alternatively, you can provide these via the command-line with the `--username` and `--password` flags, respectively:

```
otter generate --username someemail@domain.com --password thisisnotasecurepassword
```

Currently, this action supports *HTML comment filtering* with pagebreaks, but these can be disabled with the `filtering` and `pagebreaks` keys of your config.

For cases in which the generation or submission of this PDF fails, you can optionally relay this information to students using the `warn_missing_pdf` configuration. If this is set to true, a 0-point failing test case will be displayed to the student with the error thrown while trying to generate or submit the PDF:



Pass/Fail Thresholds

The configuration 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` key:

```
{
    "threshold": 0.25
}
```

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_possible` key, 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 when `points_possible` is set to 2.

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

```
{
    "points_possible": 3
}
```

Intercell Seeding

The autograder supports intercell seeding with the use of the `seed` key. *This behavior is not supported for Rmd and R script assignments, but is supported for R Jupyter notebooks.* Passing it an integer will cause the autograder to seed NumPy and Python's `random` library or call `set.seed` in R between *every* pair of code cells. This is useful for writing deterministic hidden tests. More information about Otter seeding can be found *here*. As an example, you can set an intercell seed of 42 with

```
{
    "seed": 42
}
```

Showing Autograder Results

The generator 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 setting `show_stdout` to `true`.

```
{
    "show_stdout": true
}
```

If `show_stdout` is passed, the stdout will be made available to students *only after grades are published on Gradescope*. The same can be done for hidden test outputs using the `show_hidden` key.

The *Grading on Gradescope* section details more about how output on Gradescope is formatted. Note that this behavior has no effect on any platform besides Gradescope.

Plugins

To use plugins during grading, list the plugins and their configurations in the `plugin` key of your config. If a plugin requires no configurations, it should be listed as a string. If it does, it should be listed as a dictionary with a single key, the plugin name, that maps to a subdictionary of configurations.

```
{
    "plugins": [
        "somepackage.SomeOtterPlugin",
        {
            "somepackage.SomeOtherOtterPlugin": {
            "some_config_key": "some_config_value"
            }
        }
    ]
}
```

For more information about Plugins, see *here*.

5.1.6 Generating with Otter Assign

Otter Assign comes with an option to generate this zip file automatically when the distribution notebooks are created via the `generate` key of the assignment metadata. See *Creating Assignments* for more details.

5.2 Executing Submissions

5.2.1 Grading Locally

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.

A Note on Docker

In previous versions of Otter (pre-v5), Otter had its own Docker image called `ucbdsinfra/otter-grader` which was required to be used as the base image for the containers that Otter used to grade assignments. That image has since been removed, and instead Otter relies on the same scripts it uses to set up its grading image on Gradescope to set up container images for local grading. The new default image for local grading is `ubuntu:22.04`, and all dependencies are re-downloaded when the image for an assignment is built for the first time.

Assignment Names

Whenever you use Otter Grade, you must specify an assignment name with the `-n` or `--name` flag. This assignment name is used as the tag for the Docker image that Otter creates (so you will have an image called `otter-grade:{assignment name}` for each assignment). These assignment names are required so that Otter can make effective user of Docker's image layer cache. This means that if you make changes to tests or need to grade an assignment twice, Docker doesn't need to reinstall all of the dependencies Otter defines.

These images can be quite large (~4GB), so Otter provides a way to easily prune all of the Docker images it has created:

```
otter grade --prune
```

This will prompt you to confirm that you would like to delete all of the images, which cannot be undone. After doing this, there may be some layers still in Docker's image cache (this command only deletes images with the `otter-grade` name), so if you need to free up space, you can delete these dangling images with `docker image prune`.

Configuration Files

Otter grades students submissions in individual Docker containers that are based on a Docker image generated through the use of a configuration zip file. Before grading assignments locally, an instructor should create such a zip file by using a tool such as *Otter Assign* or *Otter Generate*.

Using the CLI

Before using the command line utility, you should have

- written tests for the assignment,
- generated a configuration zip file from those tests, and
- downloaded submissions

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

Basic Usage

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

```
grading
├── autograder.zip
├── nb0.ipynb
├── nb1.ipynb
├── nb2.ipynb  # etc.
```

(continues on next page)

(continued from previous page)

```
└── tests
    ├── q1.py
    ├── q2.py
    └── q3.py  # etc.
```

Otter Grade has only one required flag, the `-n` flag for the assignment name (which we'll call `hw01`), so in the case above, our otter command is very simple:

```
otter grade -n hw01 *.ipynb
```

Because our configuration file is at `./autograder.zip`, and we don't mind output to `./`, we can use the defualt values of the `-a` and `-o` flags. This leaves only `-n` and the submission paths as the required arguments.

Note that the submission path(s) can also be specified as directories, in which case the `--ext` flag is used to determine the extension of the submission files (which defaults to `ipynb`). Therefore, the following command is equivalent to the one above in this scenario:

```
otter grade -n hw01 .
```

After grader, our directory will look like this:

```
grading
├── autograder.zip
├── final_grades.csv
├── nb0.ipynb
├── nb1.ipynb
├── nb2.ipynb  # etc.
└── tests
    ├── q1.py
    ├── q2.py
    └── q3.py  # etc.
```

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

If we wanted to generate PDFs for manual grading, we wouldadd the `--pdfs` flag to tell Otter to copy the PDFs out of the containers:

```
otter grade -n hw01 --pdfs .
```

and at the end of grading we would have

```
grading
├── autograder.zip
├── final_grades.csv
├── nb0.ipynb
├── nb1.ipynb
├── nb2.ipynb    # etc.
├── submission_pdfs
│   ├── nb0.pdf
│   ├── nb1.pdf
│   └── nb2.pdf  # etc.
└── tests
    ├── q1.py
```

(continues on next page)

(continued from previous page)

```
    ├── q2.py
    └── q3.py    # etc.
```

When a single file path is passed to `otter grade`, the submission score as a percentage is returned to the command line as well.

```
otter grade -n hw01 ./nb0.ipynb
```

To grade submissions that aren't notebook files, use the `--ext` flag, which accepts the file extension to search for submissions with. For example, if we had the same example as above but with Rmd files:

```
otter grade --ext Rmd .
```

If you're grading submission export zip files (those generated by `otter.Notebook.export` or `ottr::export`), you should pass `--ext zip` to `otter grade`.

```
otter grade --ext zip .
```

5.2.2 Grading on Gradescope

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

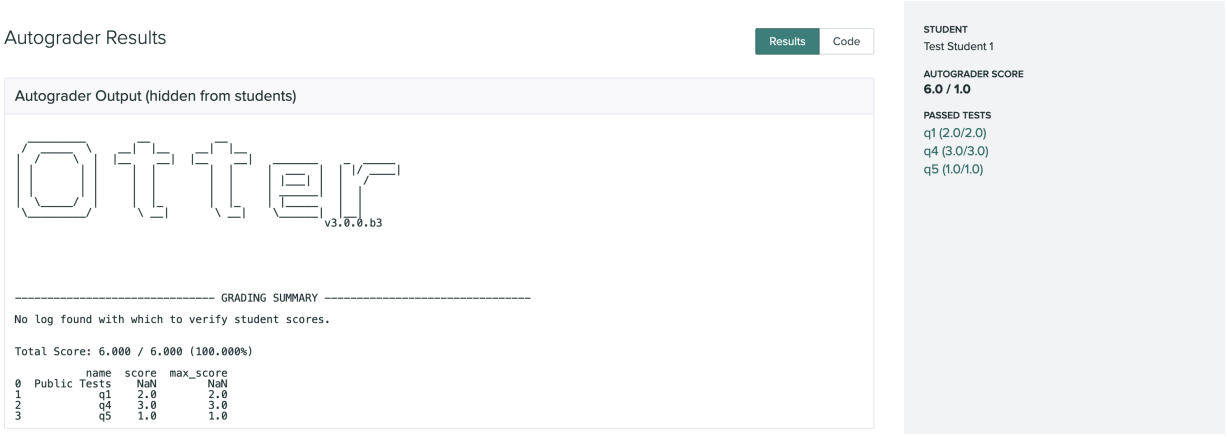
Writing Tests for Gradescope

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 `"hidden"` key of each test case indicates the visibility of that specific test case.

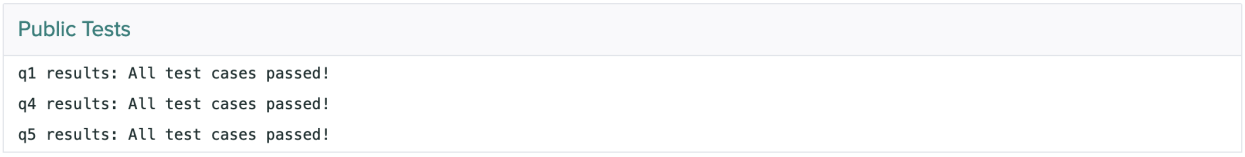
Results

Once a student's submission has been autograded, 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 information from verifying the student's scores against the logged scores, a dataframe that contains the student's score breakdown by question, and a summary of the information about test output visibility:



If `show_stdout` was true in your `otter_config.json`, then the autograder output 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.

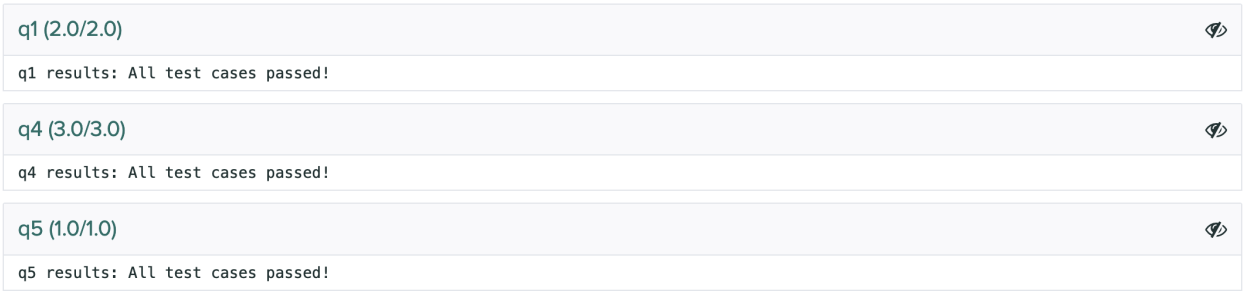
Below the autograder output, each test case is broken down into boxes. The first box is called "Public Tests" and summarizes the results of the *public* test cases only. This box is visible to students. If a student fails a hidden test but no public tests, then this box will show "All tests passed" for that question, even though the student failed a hidden test.



If the student fails a public test, then the output of the failed public test will be displayed here, but no information about hidden tests will be included. In the example below, the student failed a hidden test case in `q1` and a public test case in `q4`.

Below the "Public Tests" box will be boxes for each question. These are hidden from the student unless `show_hidden` was true in your `otter_config.json`; if that is the case, then they will become visible to students after grades are published. These boxes show the results for each question, *including the results of hidden test cases*.

If all tests cases are passed, these boxes will indicate so:



If the studen fails any test cases, that information will be presented here. In the example below, `q1 - 4` and `q1 - 5` were hidden test cases; this screenshot corresponds to the second "Public Tests" section screenshot above.

q1 (0.75/2.0)

```
q1 results:
    q1 - 1 result:
        Test case passed!

    q1 - 2 result:
        Test case passed!

    q1 - 3 result:
        Test case passed!

    q1 - 4 result:
        Trying:
            sieve(20) == {2, 3, 5, 7, 11, 13, 17, 19}
        Expecting:
            True
        **********************************************************************
        Line 1, in q1 3
        Failed example:
            sieve(20) == {2, 3, 5, 7, 11, 13, 17, 19}
        Expected:
            True
        Got:
            False

    q1 - 5 result:
        Trying:
            sieve(100) == {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
        Expecting:
            True
        **********************************************************************
        Line 1, in q1 4
        Failed example:
            sieve(100) == {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
        Expected:
            True
        Got:
            False
```

5.2.3 Non-containerized Grading

Otter supports programmatic or command-line grading of assignments without requiring the use of Docker as an intermediary. This functionality is designed to allow Otter to run in environments that do not support containerization, such as on a user's JupyterHub account. **If Docker is available, it is recommended that Otter Grade is used instead, as non-containerized grading is less secure.**

To grade locally, Otter exposes the `otter run` command for the command line or the module `otter.api` for running Otter programmatically. The use of both is described in this section. Before using Otter Run, you should have generated an *autograder configuration zip file*.

Otter Run works by creating a temporary grading directory using the `tempfile` library and replicating the autograder tree structure in that folder. It then runs the autograder there as normal. Note that Otter Run does not run environment setup files (e.g. `setup.sh`) or install requirements, so any requirements should be available in the environment being used for grading.

Grading from the Command Line

To grade a single submission from the command line, use the `otter run` utility. This has one required argument, the path to the submission to be graded, and will run Otter in a separate directory structure created using `tempfile`. Use the optional `-a` flag to specify a path to your configuration zip file if it is not at the default path `./autograder.zip`. Otter Run will write a JSON file, the results of grading, at `{output_path}/results.json` (`output_path` can be configured with the `-o` flag, and defaults to `./`).

If I wanted to use Otter Run on `hw00.ipynb`, I would run

```
otter run hw00.ipynb
```

If my autograder configuration file was at `../autograder.zip`, I would run

```
otter run -a ../autograder.zip hw00.ipynb
```

Either of the above will produce the results file at `./results.json`.

For more information on the command-line interface for Otter Run, see the *CLI Reference*.

Grading Programmatically

Otter includes an API through which users can grade assignments from inside a Python session, encapsulated in the submodule `otter.api`. The main method of the API is `otter.api.grade_submission`, which takes in an autograder configuration file path and a submission path and grades the submission, returning the `GradingResults` object that was produced during grading.

For example, to grade `hw00.ipynb` with an autograder configuration file in `autograder.zip`, I would run

```
from otter.api import grade_submission
grade_submission("hw00.ipynb", "autograder.zip")
```

`grade_submission` has an optional argument `quiet` which will suppress anything printed to the console by the grading process during execution when set to `True` (default `False`).

For more information about grading programmatically, see the *API reference*.

Grading Results

This section describes the object that Otter uses to store and manage test case scores when grading.

class `otter.test_files.`**GradingResults**(*test_files: List[TestFile]*, *notebook: NotebookNode | None = None*)

Stores and wrangles test result objects

Initialize with a list of `otter.test_files.abstract_test.TestFile` subclass objects and this class will store the results as named tuples so that they can be accessed/manipulated easily. Also contains methods to put the results into a nice `dict` format or into the correct format for Gradescope.

Parameters
results (`list[TestFile]`) – the list of test file objects summarized in this grade

all_hidden: bool

whether all results should be hidden from the student on Gradescope

clear_results()

Empties the dictionary of results.

classmethod from_ottr_json(*ottr_output*)

Creates a `GradingResults` object from the JSON output of Ottr (Otter's R client).

Parameters
ottr_output (`str`) – the JSON output of Ottr as a string

Returns
the Ottr grading results

Return type
`GradingResults`

get_plugin_data(*plugin_name*, *default=None*)

Retrieves data for plugin `plugin_name` in the results.

This method uses `dict.get` to retrive the data, so a `KeyError` is never raised if `plugin_name` is not found; rather, it returns `None`.

Parameters

- **plugin_name** (str) – the importable name of a plugin
- **default** (*any*) – a default value to return if plugin_name is not found

Returns
the data stored for plugin_name if found

Return type
any

get_result(*test_name*)

Returns the TestFile corresponding to the test with name test_name

Parameters
test_name (str) – the name of the desired test

Returns
the graded test file object

Return type
TestFile

get_score(*test_name*)

Returns the score of a test tracked by these results

Parameters
test_name (str) – the name of the test

Returns
the score

Return type
int or float

has_catastrophic_failure()

Returns whether these results contain a catastrophic error (i.e. an error that prevented submission results from being generated or read).

Returns
whether there is such an error

Return type
bool

hide_everything()

Indicates that all results should be hidden from students on Gradescope.

notebook: NotebookNode | None

the executed notebook with outputs that gave these results

output: str | None

a string to include in the output field for Gradescope

property passed_all_public

whether all public tests in these results passed

Type
bool

pdf_error: Exception | None

an error thrown while generating/submitting a PDF of the submission to display to students in the Gradescope results

property possible

the total points possible

Type
`int | float`

results: Dict[str, TestFile]

maps test/question names to their `TestFile` objects (which store the results)

set_output(*output*)

Updates the `output` field of the results JSON with text relevant to the entire submission. See https://gradescope-autograders.readthedocs.io/en/latest/specs/ for more information.

Parameters
output (`str`) – the output text

set_pdf_error(*error: Exception*)

Set a PDF generation error to be displayed as a failed (0-point) test on Gradescope.

Parameters
error (`Exception`) – the error thrown

set_plugin_data(*plugin_name*, *data*)

Stores plugin data for plugin `plugin_name` in the results. `data` must be picklable.

Parameters

- **plugin_name** (`str`) – the importable name of a plugin
- **data** (*any*) – the data to store; must be serializable with `pickle`

summary(*public_only=False*)

Generate a summary of these results and return it as a string.

Parameters
public_only (`bool`) – whether only public test cases should be included

Returns
the summary of results

Return type
`str`

property test_files

the names of all test files tracked in these grading results

Type
`list[TestFile]`

to_dict()

Converts these results into a dictinary, extending the fields of the named tuples in `results` into key, value pairs in a `dict`.

Returns
the results in dictionary form

Return type
`dict`

`to_gradescope_dict`(*ag_config*)

Convert these results into a dictionary formatted for Gradescope's autograder.

Parameters
ag_config (`otter.run.run_autograder.autograder_config.AutograderConfig`) – the autograder config

Returns
the results formatted for Gradescope

Return type
`dict`

`to_report_str`()

Returns these results as a report string generated using the `__repr__` of the `TestFile` class.

Returns
the report

Return type
`str`

`property total`

the total points earned

Type
`int | float`

`update_score`(*test_name*, *new_score*)

Override the score for the specified test file.

Parameters

- **test_name** (`str`) – the name of the test file
- **new_score** (`int | float`) – the new score

`verify_against_log`(*log*, *ignore_hidden=True*) → List[str]

Verifies these scores against the results stored in this log using the results returned by `Log.get_results` for comparison. A discrepancy occurs if the scores differ by more than the default tolerance of `math.isclose`. If `ignore_hidden` is `True`, hidden tests are ignored when verifying scores.

Parameters

- **log** (`otter.check.logs.Log`) – the log to verify against
- **ignore_hidden** (`bool`) – whether to ignore hidden tests during verification

Returns

a list of error messages for discrepancies; if none were found, the list
is empty

Return type
`list[str]`

`classmethod without_results`(*e*)

Creates an empty results object that represents an execution failure during autograding.

The returned results object will alert students and instructors to this failure, providing the error message and traceback to instructors, and report a score of 0 on Gradescope.

Parameters
e (`Exception`) – the error that was thrown

Returns
the results object

Return type
`GradingResults`

This section describes how students' submissions can be executed using the configuration file from *the previous part*. There are three main options for executing students' submissions:

- **local grading**, in which the assignments are downloaded onto the instructor's machine and graded in parallelized Docker containers,
- **Gradescope**, in which the zip file is uploaded to a Programming Assignment and students submit to the Gradescope web interface, or
- **non-containerized local grading**, in which assignments are graded in a temporary directory structure on the user's machine

The first two options are recommended as they provide better security by sandboxing students' submissions in containerized environments. The third option is present for users who are running on environments that don't have access to Docker (e.g. running on a JupyterHub account).

5.2.4 Execution

All of the options listed above go about autograding the submission in the same way; this section describes the steps taken by the autograder to generate a grade for the student's autogradable work.

The steps taken by the autograder are:

1. Copies the tests and support files from the autograder source (the contents of the autograder configuration zip file)
2. Globs all `.ipynb` files in the submission directory and ensures there are >= 1, taking that file as the submission
3. Reads in the log from the submission if it exists
4. Executes the notebook using the tests provided in the autograder source (or the log if indicated)
5. Looks for discrepancies between the logged scores and the public test scores and warns about these if present
6. If indicated, exports the notebook as a PDF and submits this PDF to the PDF Gradescope assignment
7. Makes adjustments to the scores and visibility based on the configurations
8. Generates a Gradescope-formatted JSON file for results and serializes the results object to a pickle file
9. Prints the results as a dataframe to stdout

Assignment Name Verification

To ensure that students have uploaded submissions to the correct assignment on your LMS, you can configure Otter to check for an assignment name in the notebook metadata of the submission. If you set the `assignment_name` key of your `otter_config.json` to a string, Otter will check that the submission has this name as `nb["metadata"]["otter"]["assignment_name"]` (or, in the case of R Markdown submissions, the `assignment_name` key of the YAML header) before grading it. (This metadata can be set automatically with Otter Assign.) If the name doesn't match or there is no name, an error will be raised and the assignment will not be graded. If the `assignment_name` key is not present in your `otter_config.json`, this validation is turned off.

The workflow for grading with Otter is very standard irrespective of the medium by which instructors will collect students' submissions. In general, it is a four-step process:

1. Generate a configuration zip file
2. Collect submissions via LMS (Canvas, Gradescope, etc.)
3. Run the autograder on each submission
4. Grade written questions via LMS

Steps 1 and 3 above are covered in this section of the documentation. All Otter autograders require a configuration zip file, the creation of which is described in *the next section*, and this zip file is used to create a grading environment. There are various options for where and how to grade submissions, both containerized and non-containerized, described in *Executing Submissions*.

CHAPTER

SIX

SUBMISSION EXECUTION

This section of the documentation describes the process by which submissions are executed in the autograder. Regardless of the method by which Otter is used, the autograding internals are all the same, although the execution differs slightly based on the file type of the submission.

6.1 Python

All Python submissions, regardless of type, are converted to notebooks and graded with nbformat's `ExecutePreprocessor`. For Python scripts, they are converted by creating a notebook with a single code cell that contains the script's contents.

The notebooks are executed as follows:

1. The `before_execution` *plugin* event is run.
2. Cells tagged with `otter_ignore` or with `otter[ignore]` set to `true` in the metadata are removed from the notebook.
3. (If grading from an Otter log) all lines of code which are not import statements are removed from the submission and replaced with code to unpack the serialized environments from the log.
4. Additional checks indicated in the cell metadata (`otter[tests]`) are added those cells.
5. (If intercell seeding is being used) a cell importing `numpy` and `random` is added to the top of the notebook and each code cell is prepended with the seeding code.
6. The current working directory is added to `sys.path`.
7. Cells to initialize Otter and export grading results are added at the beginning and end of the notebook, respectively.
8. The notebook is executed with the `ExecutePreprocessor`. If running in debug mode, errors are not ignored.
9. The results exported by Otter from the notebook are loaded.
10. The `after_grading` *plugin* event is run.

CHAPTER

SEVEN

DEBUGGING SUBMISSIONS

To help debug submissions that are not passing tests or which are behaving oddly, it can be helpful to use Otter's debug mode. The main use for debug mode is to examine errors that are being thrown in the submission and which Otter swallows during normal execution. This is a guide for setting up an environment where Otter's debug mode is enabled and for using it.

7.1 Setting Up a Debugging Environment

The first step is to set up a debugging environment. This step depends on how you're executing the submissions. If you're using containerized local grading (`otter grade`) or Gradescope, follow the steps in this section. If you're using non-containerized grading (`otter run`), follow the steps in *Debugging with Otter Run*.

7.1.1 Otter Grade Setup

To create a container for debugging a submission, use the Docker image that was created for your assignment when you ran `otter grade`. The image's tag will use the assignment name (the value of the `-n` or `--name` flag). If you passed `-n hw01`, the image name will be `otter-grade:hw01`. Create a container using this image and run bash:

```
docker run -it otter-grade:<assignment name> bash
```

This should create the container and drop you into a bash prompt inside of it. Now, in a separate terminal, copy the submission file into the container you've just created:

```
docker cp path/to/subm.ipynb <container id>:/autograder/submission/subm.ipynb
```

You can get the container ID either from the prompt in the container's bash shell or from the `docker container ls` command.

Now that you've set up the grading environment, continue with the debugging steps *below*. Once you're finished debugging, remember to stop the container and remove it.

7.1.2 Gradescope Setup

Find the submission you would like to debug on Gradescope and use Gradescope's Debug via SSH feature to create a container with that submission and SSH into it, then continue with the debugging steps *below*.

7.2 Debugging in a Container

In the container, `cd` into the `/autograder/source` directory. There should be an `otter_config.json` file in this directory containing Otter's autograder configuration. (If there isn't one present, make one.) Edit this file to set `"debug"` to `true` (this enables debug mode).

Now `cd` into `/autograder` and run

```
./run_autograder
```

This shell script runs Otter and its results should be printed to the console. If an error is thrown while executing the notebook, it will be shown and the autograder will stop. If you're running on Gradescope, no changes or autograder runs made here will affect the submission or its grade.

If you need to edit the submission file(s), they are located in the `/autograder/submission` directory.

7.3 Debugging with Otter Run

Read the *previous section* first. Because Otter Run relies on an autograder zip file for its configuration intsead of a Docker container, you will need to manually edit the `otter_config.json` file in your autograder zip file to set `"debug"` to `true`. Then, re-zip the zip file's contents and use this new autograder zip file for `otter run`.

CHAPTER

EIGHT

PLUGINS

8.1 Built-In Plugins

8.1.1 Google Sheets Grade Override

This plugin allows you to override test case scores during grading by specifying the new score in a Google Sheet that is pulled in. To use this plugin, you must set up a Google Cloud project and obtain credentials for the Google Sheets API. This plugin relies on `gspread` to interact with the Google Sheets API and its documentation contains instructions on how to obtain credentials. Once you have created credentials, download them as a JSON file.

This plugin requires two configurations: the path to the credentials JSON file and the URL of the Google Sheet. The former should be entered with the key `credentials_json_path` and the latter `sheet_url`; for example, in Otter Assign:

```
plugins:
    - otter.plugins.builtin.GoogleSheetsGradeOverride:
        credentials_json_path: /path/to/google/credentials.json
        sheet_url: https://docs.google.com/some/google/sheet
```

The first tab in the sheet is assumed to be the override information.

During Otter Generate, the plugin will read in this JSON file and store the relevant data in the `otter_config.json` for use during grading. The Google Sheet should have the following format:

Assignment ID	Email	Test Case	Points	PDF
123456	student@univ.edu	q1a - 1	1	false

- `Assignment ID` should be the ID of the assignment on Gradescope (to allow one sheet to be used for multiple assignments)
- `Email` should be the student's email address on Gradescope
- `Test Case` should be the name of the test case as a string (e.g. if you have a test file `q1a.py` with 3 test cases, overriding the second case would be `q1a - 2`)
- `Points` should be the point value to assign the student for that test case
- `PDF` should be whether or not a PDF should be re-submitted (to prevent losing manual grades on Gradescope for regrade requests)

Note that the use of this plugin requires specifying `gspread` in your requirements, as it is not included by default in Otter's container image.

`otter.plugins.builtin.GoogleSheetsGradeOverride` Reference

The actions taken by at hook for this plugin are detailed below.

8.1.2 Rate Limiting

This plugin allows instructors to limit the number of times students can submit to the autograder in any given window to prevent students from using the AG as an "oracle". This plugin has a required configuration `allowed_submissions`, the number of submissions allowed during the window, and accepts optional configurations `weeks`, `days`, `hours`, `minutes`, `seconds`, `milliseconds`, and `microseconds` (all defaulting to `0`) to configure the size of the window. For example, to specify 5 allowed submissions per-day in your `otter_config.json`:

```
{
    "plugins": [
        {
            "otter.plugins.builtin.RateLimiting": {
                "allowed_submissions": 5,
                "days": 1
            }
        }
    ]
}
```

When a student submits, the window is caculated as a `datetime.timedelta` object and if the student has at least `allowed_submissions` in that window, the submission's results are hidden and the student is only shown a message; from the example above:

```
You have exceeded the rate limit for the autograder. Students are allowed 5 submissions␣
↪every 1
days.
```

The results of submission execution are still visible to instructors.

If the student's submission is allowed based on the rate limit, the plugin outputs a message in the plugin report; from the example above:

```
Students are allowed 5 submissions every 1 days. You have {number of previous␣
↪submissions}
submissions in that period.
```

`otter.plugins.builtin.RateLimiting` Reference

The actions taken by at hook for this plugin are detailed below.

8.1.3 Gmail Notifications

This plugin sends students an email summary of the results of their public test cases using the Gmail API. To use this plugin, you must set up a Google Cloud project and create an OAuth2 Client that can be used as a proxy for another email address.

Setup

Before using this plugin, you must first create Google OAuth2 credentials on a Google Cloud project. Using these credentials, you must then obtain a refresh token that connects these credentials to the account from which you want to send the emails. Once you have the credentials, use the `gmail_oauth2` CLI included with Otter to generate the refresh token. Use the command below to perform this action, substituting in your credentials client ID and secret.

```
gmail_oauth2 --generate_oauth2_token --client_id="<client_id>" --client_secret="<client_
↪secret>" --scope="https://www.googleapis.com/auth/gmail.send"
```

This command will prompt you to visit a URL and authenticate with Google, which will then provide you with a verification code that you should enter back in the CLI. You will then be provided with your refresh token.

Configuration

This plugin requires four configurations: the Google client ID, client secret, refresh token, and the email address from which emails are being sent (the same one used to authenticate for the refresh token).

```
plugins:
    - otter.plugins.builtin.GmailNotifications:
        client_id: client_id-1234567.apps.googleusercontent.com
        client_secret: abc123
        refresh_token: 1//abc123
        email: course@univ.edu
```

Optionally, this plugin also accepts a `catch_api_error` configuration, which is a boolean indicating whether Google API errors should be ignored.

Output

The email sent to students uses the following jinja2 template:

```
<p>Hello {{ student_name }},</p>

<p>Your submission {% if zip_name %}<code>{{ zip_name }}</code>{% endif %} for assignment
<strong>{{ assignment_title }}</strong> submitted at {{ submission_timestamp }} received␣
↪the
following scores on public tests:</p>

<pre>{{ score_report }}</pre>

<p>If you have any questions or notice any issues, please contact your instructors.</p>

<hr>

<p>This message was automatically generated by Otter-Grader.</p>
```

The resulting email looks like this:

Hello ,

Your submission `lab07_2021_06_04T11_11_22_292646.zip` for assignment **Test Email Notifications** submitted at 2021-06-04T11:11:34.889963-07:00 received the following scores on public tests:

```
q1_1 results: All test cases passed!

q1_2 results:
    q1_2 - 1 result:
        Trying:
            0 <= len(nash_equilibria) <= 4
        Expecting:
            True
        **********************************************************************
        Line 1, in q1_2 0
        Failed example:
            0 <= len(nash_equilibria) <= 4
        Exception raised:
            Traceback (most recent call last):
              File "/root/miniconda3/envs/otter-env/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "", line 1, in
                0 <= len(nash_equilibria) <= 4
            TypeError: object of type 'NoneType' has no len()

    q1_2 - 1 message: wrong # of nash equilibria

    q1_2 - 2 result:
        Trying:
            for val in nash_equilibria:
                assert val % 1 == 0
        Expecting nothing
        **********************************************************************
        Line 1, in q1_2 1
        Failed example:
            for val in nash_equilibria:
                assert val % 1 == 0
        Exception raised:
            Traceback (most recent call last):
              File "/root/miniconda3/envs/otter-env/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "", line 1, in
                for val in nash_equilibria:
            TypeError: 'NoneType' object is not iterable

    q1_2 - 2 message: values not integers

q2_1 results: All test cases passed!

q2_2 results: All test cases passed!

q2_3 results: All test cases passed!

q2_4 results: All test cases passed!

q2_5 results: All test cases passed!

q2_6 results: All test cases passed!

feedback results: All test cases passed!
```

If you have any questions or notice any issues, please contact your instructors.

This message was automatically generated by Otter-Grader.

Note that the report only includes the results of **public** tests.

`otter.plugins.builtin.GmailNotifications` Reference

Otter comes with a few built-in plugins to optionally extend its own functionality. These plugins are documented here. **Note:** To use these plugins, specify the importable names *exactly* as seen here (i.e. `otter.plugins.builtin.GoogleSheetsGradeOverride`, not `otter.plugins.builtin.grade_override.GoogleSheetsGradeOverride`).

8.2 Creating Plugins

Plugins can be created by exposing an importable class that inherits from `otter.plugins.AbstractOtterPlugin`. This class defines the base implementations of the plugin event methods and sets up the other information needed to run the plugin.

8.2.1 Plugin Configurations

When instantiated, plugins will receive three arguments: `submission_path`, the absolute path to the submission being executed, `submission_metadata`, the information about the submission as read in from `submission_metadata.json`, and `plugin_config`, the parsed configurations from the user-specified list of plugins in the `plugins` key of `otter_config.json`. The default implementation of `__init__`, which should have the signature shown *below*, sets these two values to the instance variables `self.submission_metadata` and `self.plugin_config`, respectively. *Note that some events are executed in contexts in which some or all of these variables are unavailable, and so these events should not rely on the use of these instance variables.* In cases in which these are unavailable, they will be set for the falsey version of their type, e.g. `{}` for `self.submission_metadata`.

8.2.2 Plugin Events

Plugins currently support five events. Most events are expected to modify the state of the autograder and should not return anything, unless otherwise noted. Any plugins not supported should raise a `otter.plugins.PluginEventNotSupportedException`, as the default implementations in `AbstractOtterPlugin` do. Note that all methods should have the signatures described *below*.

`during_assign`

The `during_assign` method will be called when Otter Assign is run. It will receive the `otter.assign.assignment.Assignment` instance with configurations for this assignment. The plugin is run after output directories are written.

`during_generate`

The `during_generate` method will be called when Otter Generate is run. It will receive the parsed dictionary of configurations from `otter_config.json` as its argument. Any changes made to the configurations will be written to the `otter_config.json` that will be stored in the configuration zip file generated by Otter Generate.

from_notebook

The `from_notebook` method will be called when a student makes a call to `otter.Notebook.run_plugin` with the plugin name. Any additional args and kwargs passed to `Notebook.run_plugin` will be passed to the plugin method. This method should not return anything but can modify state or provide feedback to students. **Note that because this plugin is not run in the grading environment, it will *not* be passed the submission path, submission metadata, or a plugin config.** All information needed to run this plugin event must be gathered from the arguments passed to it.

notebook_export

The `notebook_export` method will be called when a student makes a call to `otter.Notebook.add_plugin_files` with the plugin name. Any additional args and kwargs passed to `Notebook.add_plugin_files` will be passed to the plugin mtehod. This method should return a list of strings that correspond to file paths that should be included in the zip file generated by `otter.Notebook.export` when it is run. The `Notebook` instance tracks these files in a list. **Note that because this plugin is not run in the grading environment, it will *not* be passed the submission path, submission metadata, or a plugin config.** All information needed to run this plugin event must be gathered from the arguments passed to it.

before_grading

The `before_grading` method will be called before grading occurs, just after the plugins are instantiated, and will be passed an instance of the `otter.run.run_autograder.autograder_config.AutograderConfig` class, a fica configurations class. Any changes made to the options will be used during grading.

before_execution

The `before_execution` method will be called before execution occurs and will be passed the student's submission. If the submission is a notebook, the object passed will be a `nbformat.NotebookNode` from the parsed JSON. If the submission is a script, the objet passed will be a string containing the file text. This method should return a properly-formatted `NotebookNode` or string that will be executed in place of the student's original submission.

after_grading

The `after_grading` method will be called after grading has occurred (meaning all tests have been run and the results object has been created). It will be passed the `otter.test_files.GradingResults` object that contains the results of the student's submission against the tests. Any changes made to this object will be reflected in the results returned by the autograder. For more information about this object, see *Non-containerized Grading*.

generate_report

The `generate_report` method will be called after results have been written. It receives no arguments but should **return a value**. The return value of this method should be a string that will be printed to stdout by Otter as part of the grading report.

8.2.3 `AbstractOtterPlugin` Reference

class otter.plugins.**AbstractOtterPlugin**(*submission_path*, *submission_metadata*, *plugin_config*)

Abstract base class for Otter plugins to inherit from. Includes the following methods:

- `during_assign`: run during Otter Assign after output directories are written
- `during_generate`: run during Otter Generate while all files are in-memory and before the the *tmp* directory is created
- `from_notebook`: run as students as they work through the notebook; see `Notebook.run_plugin`
- `notebook_export`: run by `Notebook.export` for adding files to the export zip file
- `before_grading`: run before the submission is executed for altering configurations
- `before_execution`: run before the submission is executed for altering the submission
- `after_grading`: run after all tests are run and scores are assigned
- `generate_report`: run after results are written

See the docstring for the default implementation of each method for more information, including arguments and return values. If plugin has no actions for any event above, that method should raise `otter.plugins.PluginEventNotSupportedException`. (This is the default behavior of this ABC, so inheriting from this class will do this for you for any methods you don't overwrite.)

If this plugin requires metadata, it should be included in the plugin configuration of the `otter_config.json` file as a subdictionary with key a key corresponding to the importable name of the plugin. If no configurations are required, the plugin name should be listed as a string. For example, the config below provides configurations for `MyOtterPlugin` but not `MyOtherOtterPlugin`.

```
{
    "plugins": [
        {
            "my_otter_plugin_package.MyOtterPlugin": {
                "some_metadata_key": "some_value"
            }
        },
        "my_otter_plugin_package.MyOtherOtterPlugin"
    ]
}
```

Parameters

- **submission_path** (`str`) – the absolute path to the submission being graded
- **submission_metadata** (`dict`) – submission metadata; if on Gradescope, see https://gradescope-autograders.readthedocs.io/en/latest/submission_metadata/
- **plugin_config** (`dict`) – configurations from the `otter_config.json` for this plugin, pulled from `otter_config["plugins"][][PLUGIN_NAME]` if `otter_config["plugins"][]` is a `dict`

submission_path

the absolute path to the submission being graded

Type
`str`

submission_metadata

submission metadata; if on Gradescope, see https://gradescope-autograders.readthedocs.io/en/latest/submission_metadata/

Type
dict

plugin_config

configurations from the `otter_config.json` for this plugin, pulled from `otter_config["plugins"][][PLUGIN_NAME]` if `otter_config["plugins"][]` is a `dict`

Type
dict

after_grading(*results*)

Plugin event run after all tests are run on the resulting environment that gets passed the `otter.test_files.GradingResults` object that stores the grading results for each test case.

Parameters
results (`otter.test_files.GradingResults`) – the results of all test cases

Raises
PluginEventNotSupportedException – if the event is not supported by this plugin

before_execution(*submission*)

Plugin event run before the execution of the submission which can modify the submission itself. This method should return a properly-formatted `NotebookNode` or string that will be executed in place of the student's original submission.

Parameters
submission (`nbformat.NotebookNode` or `str`) – the submission for grading; if it is a notebook, this will be the JSON-parsed `dict` of its contents; if it is a script, this will be a string containing the code

Returns
the altered submission to be executed

Return type
`nbformat.NotebookNode` or `str`

Raises
PluginEventNotSupportedException – if the event is not supported by this plugin

before_grading(*config*)

Plugin event run before the execution of the submission which can modify the dictionary of grading configurations.

Parameters
config (`otter.run.run_autograder.autograder_config.AutograderConfig`) – the autograder config

Raises
PluginEventNotSupportedException – if the event is not supported by this plugin

during_assign(*assignment*)

Plugin event run during the execution of Otter Assign after output directories are wrriten. Assignment configurations are passed in via the `assignment` argument.

Parameters
assignment (`otter.assign.assignment.Assignment`) – the `Assignment` instance with

configurations for the assignment; used similar to an `AttrDict` where keys are accessed with the dot syntax (e.g. `assignment.master` is the path to the master notebook)

Raises
PluginEventNotSupportedException – if the event is not supported by this plugin

during_generate(*otter_config*, *assignment*)

Plugin event run during the execution of Otter Generate that can modify the configurations to be written to `otter_config.json`.

Parameters

- **otter_config** (`dict`) – the dictionary of Otter configurations to be written to `otter_config.json` in the zip file
- **assignment** (`otter.assign.assignment.Assignment`) – the `Assignment` instance with configurations for the assignment if Otter Assign was used to generate this zip file; will be set to `None` if Otter Assign is not being used

Raises
PluginEventNotSupportedException – if the event is not supported by this plugin

from_notebook(**args*, ***kwargs*)

Plugin event run by students as they work through a notebook via the `Notebook` API (see `Notebook.run_plugin`). Accepts arbitrary arguments and has no return.

Parameters

- ***args** – arguments for the plugin event
- ****kwargs** – keyword arguments for the plugin event

Raises
PluginEventNotSupportedException – if the event is not supported by this plugin

generate_report()

Plugin event run after grades are written to disk. This event should return a string that gets printed to stdout as a part of the plugin report.

Returns
the string to be included in the report

Return type
`str`

Raises
PluginEventNotSupportedException – if the event is not supported by this plugin

notebook_export(**args*, ***kwargs*)

Plugin event run when a student calls `Notebook.export`. Accepts arbitrary arguments and should return a list of file paths to include in the exported zip file.

Parameters

- ***args** – arguments for the plugin event
- ****kwargs** – keyword arguments for the plugin event

Returns
the list of file paths to include in the export

Return type
`list[str]`

Raises
PluginEventNotSupportedException – if the event is not supported by this plugin

Otter-Grader supports grading plugins that allow users to override the default behavior of Otter by injecting an importable class that is able to alter the state of execution.

8.3 Using Plugins

Plugins can be configured in your `otter_config.json` file by listing the plugins as their fully-qualified importable names in the `plugins` key:

```
{
    "plugins": [
        "mypackage.MyOtterPlugin"
    ]
}
```

To supply configurations for the plugins, add the plugin to `plugins` as a dictionary mapping a single key, the plugin importable name, to a dictionary of configurations. For example, if we needed `mypackage.MyOtterPlugin` with configurations but `mypackage.MyOtherOtterPlugin` required none, the configurations for these plugins would look like

```
{
    "plugins": [
        {
            "mypackage.MyOtterPlugin": {
                "key1": "value1",
                "key2": "value2"
            }
        },
        "mypackage.MyOtherOtterPlugin"
    ]
}
```

8.4 Building Plugins

Plugins can be created as importable classes in packages that inherit from the `otter.plugins.AbstractOtterPlugin` class. For more information about creating plugins, see *Creating Plugins*.

CHAPTER

NINE

PDF GENERATION AND FILTERING

Otter includes a tool for generating PDFs of notebooks that optionally incorporates notebook filtering for generating PDFs for manual grading. There are two options for exporting PDFs:

- **PDF via LaTeX:** this uses nbconvert, pandoc, and LaTeX to generate PDFs from TeX files
- **PDF via HTML:** this uses wkhtmltopdf and the Python packages pdfkit and pypdf to generate PDFs from HTML files

Otter Export is used by Otter Assign to generate Gradescope PDF templates and solutions, in the Gradescope autograder to generate the PDFs of notebooks, by `otter.Notebook` to generate PDFs and in Otter Grade when PDFs are requested.

9.1 Cell Filtering

Otter Export uses HTML comments in Markdown cells to perform cell filtering. Filtering is the default behavior of most exporter implementations, but not of the exporter itself.

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. When using Otter Export, you can also optionally add page breaks after an `<!-- END QUESTION -->` by setting `pagebreaks=True` in `otter.export.export_notebook` or using the corresponding flags/arguments in whichever utility is calling Otter Export.

CHAPTER

TEN

INTERCELL SEEDING

The Otter suite of tools supports intercell seeding, a process by which notebooks and scripts can be seeded for the generation of pseudorandom numbers, which is very advantageous for writing deterministic hidden tests. This section discusses the inner mechanics of intercell seeding, while the flags and other UI aspects are discussed in the sections corresponding to the Otter tool you're using.

10.1 Seeding Mechanics

This section describes at a high-level how seeding is implemented in the autograder at the layer of code execution. There are two methods of seeding: the use of a seed variable, and seeding the libraries themselves.

The examples in this section are all in Python, but they also work the same in R.

10.1.1 Seed Variables

The use of seed variables is configured by setting both the `seed` and `seed_variable` configurations in your `otter_config.json`:

```
{
    "seed": 42,
    "seed_variable": "rng_seed"
}
```

Notebooks

When seeding in notebooks, an assignment of the variable name indicated by `seed_variable` is added at the top of every cell. In this way, cells that use the seed variable to create an RNG or seed libraries can have their seed changed from the default at runtime.

For example, let's say we have the configuration above and the two cells below:

```
x = 2 ** np.arange(5)
```

and

```
rng = np.random.default_rng(rng_seed)
y = rng.normal(100)
```

They would be sent to the autograder as:

```
rng_seed = 42
x = 2 ** np.arange(5)
```

and

```
rng_seed = 42
rng = np.random.default_rng(rng_seed)
y = rng.normal(100)
```

Note that the initial value of the seed variable must be set in a separate cell from any of its uses, or the original value will override autograder value.

Python Scripts

Seed variables currently do not support Python scripts.

10.1.2 Seeding Libraries

Seeding libraries is configured by setting the `seed` configuration in your `otter_config.json`:

```
{
    "seed": 42
}
```

Notebooks

When seeding in notebooks, both NumPy and `random` are seeded using an integer provided by the instructor. The seeding code is added to each cell's source before running it through the executor, meaning that the results of *every* cell are seeded with the same seed. For example, let's say we have the configuration above and the two cells below:

```
x = 2 ** np.arange(5)
```

and

```
y = np.random.normal(100)
```

They would be sent to the autograder as:

```
np.random.seed(42)
random.seed(42)
x = 2 ** np.arange(5)
```

and

```
np.random.seed(42)
random.seed(42)
y = np.random.normal(100)
```

Python Scripts

Seeding Python files is relatively more simple. The implementation is similar to that of notebooks, but the script is only seeded once, at the beginning. Thus, the Python file below:

```
import numpy as np

def sigmoid(t):
    return 1 / (1 + np.exp(-1 * t))
```

would be sent to the autograder as

```
np.random.seed(42)
random.seed(42)
import numpy as np

def sigmoid(t):
    return 1 / (1 + np.exp(-1 * t))
```

You don't need to worry about importing NumPy and `random` before seeding as these modules are loaded by the autograder and provided in the global environment that the script is executed against.

10.2 Caveats

Remekber, when writing assignments or using assignment generation tools like Otter Assign, the instructor must **seed the solutions themselves** before writing hidden tests in order to ensure they are grading the correct values. Also, students will not have access to the random seed, so any values they compute in the notebook may be different from the results of their submission when it is run through the autograder.

CHAPTER

ELEVEN

LOGGING

In order to assist with debugging students' checks, Otter automatically logs events when called from `otter.Notebook` or `otter check`. The events are logged in the following methods of `otter.Notebook`:

- `__init__`
- `_auth`
- `check`
- `check_all`
- `export`
- `submit`
- `to_pdf`

The events are stored as `otter.logs.LogEntry` objects which are pickled and appended to a file called `.OTTER_LOG`. To interact with a log, the `otter.logs.Log` class is provided; logs can be read in using the class method `Log.from_file`:

```
from otter.check.logs import Log
log = Log.from_file(".OTTER_LOG")
```

The log order defaults to chronological (the order in which they were appended to the file) but this can be reversed by setting the `ascending` argument to `False`. To get the most recent results for a specific question, use `Log.get_results`:

```
log.get_results("q1")
```

Note that the `otter.logs.Log` class does not support editing the log file, only reading and interacting with it.

11.1 Logging Environments

Whenever a student runs a check cell, Otter can store their current global environment as a part of the log. The purpose of this is twofold: 1) to allow the grading of assignments to occur based on variables whose creation requires access to resources not possessed by the grading environment, and 2) to allow instructors to debug students' assignments by inspecting their global environment at the time of the check. This behavior must be preconfigured with an *Otter configuration file* that has its `save_environment` key set to `true`.

Shelving is accomplished by using the dill library to pickle (almost) everything in the global environment, with the notable exception of modules (so libraries will need to be reimported in the instructor's environment). The environment (a dictionary) is pickled and the resulting file is then stored as a byte string in one of the fields of the log entry.

Environments can be saved to a log entry by passing the environment (as a dictionary) to `LogEntry.shelve`. Any variables that can't be shelved (or are ignored) are added to the `unshelved` attribute of the entry.

```
from otter.logs import LogEntry
entry = LogEntry()
entry.shelve(globals())
```

The `shelve` method also optionally takes a parameter `variables` that is a dictionary mapping variable names to fully-qualified type strings. If passed, only variables whose names are keys in this dictionary and whose types match their corresponding values will be stored in the environment. This helps from serializing unnecessary objects and prevents students from injecting malicious code into the autograder. To get the type string, use the function `otter.utils.get_variable_type`. As an example, the type string for a pandas `DataFrame` is `"pandas.core.frame.DataFrame"`:

```
>>> import pandas as pd
>>> from otter.utils import get_variable_type
>>> df = pd.DataFrame()
>>> get_variable_type(df)
'pandas.core.frame.DataFrame'
```

With this, we can tell the log entry to only shelve dataframes named `df`:

```
from otter.logs import LogEntry
variables = {"df": "pandas.core.frame.DataFrame"}
entry = LogEntry()
entry.shelve(globals(), variables=variables)
```

If you are grading from the log and are utilizing `variables`, you **must** include this dictionary as a JSON string in your configuration, otherwise the autograder will deserialize anything that the student submits. This configuration is set in two places: in the *Otter configuration file* that you distribute with your notebook and in the autograder. Both of these are handled for you if you use *Otter Assign* to generate your distribution files.

To retrieve a shelved environment from an entry, use the `LogEntry.unshelve` method. During the process of unshelving, all functions have their `__globals__` updated to include everything in the unshelved environment and, optionally, anything in the environment passed to `global_env`.

```
>>> env = entry.unshelve() # this will have everything in the shelf in it -- but not␣
↪factorial
>>> from math import factorial
>>> env_with_factorial = entry.unshelve({"factorial": factorial}) # add factorial to all␣
↪fn __globals__
>>> "factorial" in env_with_factorial["some_fn"].__globals__
True
>>> factorial is env_with_factorial["some_fn"].__globals__["factorial"]
True
```

See the reference *below* for more information about the arguments to `LogEntry.shelve` and `LogEntry.unshelve`.

11.2 Debugging with the Log

The log is useful to help students debug tests that they are repeatedly failing. Log entries store any errors thrown by the process tracked by that entry and, if the log is a call to `otter.Notebook.check`, also the test results. Any errors held by the log entry can be re-thrown by calling `LogEntry.raise_error`:

```
from otter.logs import Log
log = Log.from_file(".OTTER_LOG")
entry = log.entries[0]
entry.raise_error()
```

The test results of an entry can be returned using `LogEntry.get_results`:

```
entry.get_results()
```

11.3 Grading from the Log

As noted earlier, the environments stored in logs can be used to grade students' assignments. If the grading environment does not have the dependencies necessary to run all code, the environment saved in the log entries will be used to run tests against. For example, if the execution hub has access to a large SQL server that cannot be accessed by a Gradescope grading container, these questions can still be graded using the log of checks run by the students and the environments pickled therein.

To configure these pregraded questions, include an *Otter configuration file* in the assignment directory that defines the notebook name and that the saving of environments should be turned on:

```
{
    "notebook": "hw00.ipynb",
    "save_environment": true
}
```

If you are restricting the variables serialized during checks, also set the `variables` or `ignore_modules` parameters. If you are grading on Gradescope, you must also tell the autograder to grade from the log using the `--grade-from-log` flag when running or the `grade_from_log` subkey of `generate` if using Otter Assign.

11.4 Otter Logs Reference

11.4.1 otter.logs.Log

class otter.logs.**Log**(*entries*, *ascending=True*)

A class for reading and interacting with a log. Allows you to iterate over the entries in the log and supports integer indexing. *Does not support editing the log file.*

Parameters

- **entries** (list of LogEntry) – the list of entries for this log
- **ascending** (bool, optional) – whether the log is sorted in ascending (chronological) order; default True

`entries`
the list of log entries in this log

Type
`list` of `LogEntry`

`ascending`
whether `entries` is sorted chronologically; `False` indicates reverse- chronological order

Type
`bool`

`classmethod from_file`(*filename*, *ascending=True*)
Loads and sorts a log from a file.

Parameters

- **`filename`** (`str`) – the path to the log
- **`ascending`** (`bool`, optional) – whether the log should be sorted in ascending (chronological) order; default `True`

Returns
the `Log` instance created from the file

Return type
`Log`

`get_question_entry`(*question*)
Gets the most recent entry corresponding to the question `question`

Parameters
`question` (`str`) – the question to get

Returns
the most recent log entry for `question`

Return type
`LogEntry`

Raises
`QuestionNotInLogException` – if the question is not in the log

`get_questions`()
Returns a sorted list of all question names that have entries in this log.

Returns
the questions in this log

Return type
`list` of `str`

`get_results`(*question*)
Gets the most recent grading result for a specified question from this log

Parameters
`question` (`str`) – the question name to look up

Returns
the most recent result for the question

Return type
`otter.test_files.abstract_test.TestCollectionResults`

Raises
QuestionNotInLogException – if the question is not found

`question_iterator()`

Returns an iterator over the most recent entries for each question.

Returns
the iterator

Return type
`QuestionLogIterator`

`sort(ascending=True)`

Sorts this logs entries by timestmap using `LogEntry.sort_log`.

Parameters
ascending (`bool`, optional) – whether to sort the log chronologically; defaults to `True`

11.4.2 otter.logs.LogEntry

`class otter.logs.LogEntry(event_type: EventType, shelf: bytes | None = None, results: GradingResults | None = None, question: str | None = None, success: bool = True, error: Exception | None = None)`

An entry in Otter's log. Tracks event type, grading results, success of operation, and errors thrown.

Parameters

- **event_type** (`EventType`) – the type of event for this entry
- **results** (`otter.test_files.TestFile | otter.test_files.GradingResults | None`) – the results of grading if this is an `EventType.CHECK` or `EventType.END_CHECK_ALL` record
- **question** (`str`) – the question name for an `EventType.CHECK` record
- **success** (`bool`) – whether the operation was successful
- **error** (`Exception`) – an error thrown by the process being logged if any

`error: Exception | None`

an error thrown by the tracked process if applicable

`event_type: EventType`

the entry type

`flush_to_file(filename)`

Appends this log entry (pickled) to a file

Parameters
filename (`str`) – the path to the file to append this entry

`get_results()`

Get the results stored in this log entry

Returns

the results at this
entry if this is an `EventType.CHECK` record

Return type
`list` of `otter.test_files.abstract_test.TestCollectionResults`

`get_score_perc`()

Returns the percentage score for the results of this entry

Returns
the percentage score

Return type
`float`

`static log_from_file`(*filename*, *ascending=True*)

Reads a log file and returns a sorted list of the log entries pickled in that file

Parameters

- **filename** (`str`) – the path to the log
- **ascending** (`bool`) – whether the log should be sorted in ascending (chronological) order

Returns
the sorted log

Return type
`list[LogEntry]`

`not_shelved: List[str]`

a list of variable names that were not added to the shelf

`question: str | None`

question name if this is a check entry

`raise_error`()

Raises the error stored in this entry

Raises
Exception – the error stored at this entry, if present

`results: GradingResults | None`

grading results if this is an `EventType.CHECK` entry

`shelf: bytes | None`

a pickled environment stored as a bytes string

`shelve`(*env*, *delete=False*, *filename=None*, *ignore_modules=[]*, *variables=None*)

Stores an environment `env` in this log entry using dill as a `bytes` object in this entry as the `shelf` attribute. Writes names of any variables in `env` that are not stored to the `not_shelved` attribute.

If `delete` is `True`, old environments in the log at `filename` for this question are cleared before writing `env`. Any module names in `ignore_modules` will have their functions ignored during pickling.

Parameters

- **env** (`dict`) – the environment to pickle
- **delete** (`bool`, optional) – whether to delete old environments
- **filename** (`str`, optional) – path to log file; ignored if `delete` is `False`
- **ignore_modules** (`list` of `str`, optional) – module names to ignore during pickling
- **variables** (`dict`, optional) – map of variable name to type string indicating **only** variables to include (all variables not in this dictionary will be ignored)

Returns
this entry

Return type
LogEntry

`static shelve_environment`(*env*, *variables=None*, *ignore_modules=[]*)

Pickles an environment `env` using dill, ignoring any functions whose module is listed in `ignore_modules`. Returns the pickle file contents as a `bytes` object and a list of variable names that were unable to be shelved/ignored during shelving.

Parameters

- **env** (`dict`) – the environment to shelve
- **variables** (`dict` *or* `list`, optional) – a map of variable name to type string indicating **only** variables to include (all variables not in this dictionary will be ignored) or a list of variable names to include regardless of tpye
- **ignore_modules** (`list` of `str`, optional) – the module names to igonre

Returns

the pickled environment and list of variable names that were not shelved

Return type
`tuple[bytes, list[str]`

`static sort_log`(*log*, *ascending=True*)

Sorts a list of log entries by timestamp

Parameters

- **log** (`list` of `LogEntry`) – the log to sort
- **ascending** (`bool`, optional) – whether the log should be sorted in ascending (chronological) order

Returns
the sorted log

Return type
`list[LogEntry]`

`success: bool`

whether the operation tracked by this entry was successful

`timestamp: datetime`

timestamp of event in UTC

`unshelve`(*global_env={}*)

Parses a `bytes` object stored in the `shelf` attribute and unpickles the object stored there using dill. Updates the `__globals__` of any functions in `shelf` to include elements in the shelf. Optionally includes the env passed in as `global_env`.

Parameters
global_env (`dict`, optional) – a global env to include in unpickled function globals

Returns
the shelved environment

Return type
`dict`

11.4.3 otter.logs.EventType

class otter.logs.**EventType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum of event types for log entries

AUTH = 1

an auth event

BEGIN_CHECK_ALL = 2

beginning of a check-all cell

BEGIN_EXPORT = 3

beginning of an assignment export

CHECK = 4

a check of a single question

END_CHECK_ALL = 5

ending of a check-all cell

END_EXPORT = 6

ending of an assignment export

INIT = 7

initialization of an *otter.check.notebook.Notebook* object

SUBMIT = 8

submission of an assignment (unused since Otter Service was removed)

TO_PDF = 9

PDF export of a notebook (not used during a submission export)

CHAPTER

TWELVE

OTTER.API REFERENCE

A programmatic API for using Otter-Grader

otter.api.**export_notebook**(*nb_path*, *dest=None*, *exporter_type=None*, ***kwargs*)

Exports a notebook file at `nb_path` to a PDF with optional filtering and pagebreaks. Accepts other `kwargs` passed to the exporter class's `convert_notebook` class method.

Parameters

- **nb_path** (`str`) – path to notebook
- **dest** (`str`, optional) – path to write PDF
- **exporter_type** (`str`, optional) – the type of exporter to use; one of `['html', 'latex']`
- ****kwargs** – additional configurations passed to exporter

Returns

the path at which the PDF was written

Return type

`str`

otter.api.**grade_submission**(*submission_path*, *ag_path='autograder.zip'*, *quiet=False*, *debug=False*)

Runs non-containerized grading on a single submission at `submission_path` using the autograder configuration file at `ag_path`.

Creates a temporary grading directory using the `tempfile` library and grades the submission by replicating the autograder tree structure in that folder and running the autograder there. Does not run environment setup files (e.g. `setup.sh`) or install requirements, so any requirements should be available in the environment being used for grading.

Print statements executed during grading can be suppressed with `quiet`.

Parameters

- **submission_path** (`str`) – path to submission file
- **ag_path** (`str`) – path to autograder zip file
- **quiet** (`bool`, optional) – whether to suppress print statements during grading; default `False`
- **debug** (`bool`, optional) – whether to run the submission in debug mode (without ignoring errors)

Returns

the results object produced during the grading of the submission.

Return type
otter.test_files.GradingResults

CHAPTER

THIRTEEN

CLI REFERENCE

13.1 otter

Command-line utility for Otter-Grader, a Python-based autograder for Jupyter Notebooks, RMarkdown files, and Python and R scripts. For more information, see https://otter-grader.readthedocs.io/.

```
otter [OPTIONS] COMMAND [ARGS]...
```

Options

--version

Show the version and exit

13.1.1 assign

Create distribution versions of the Otter Assign formatted notebook MASTER and write the results to the directory RESULT, which will be created if it does not already exist.

```
otter assign [OPTIONS] MASTER RESULT
```

Options

-v, --verbose

Verbosity of the logged output

--no-run-tests

Do not run the tests against the autograder notebook

--no-pdfs

Do not generate PDFs; overrides assignment config

--username <username>

Gradescope username for generating a token

--password <password>

Gradescope password for generating a token

--debug

Do not ignore errors in running tests for debugging

Arguments

MASTER
Required argument

RESULT
Required argument

13.1.2 check

Check the Python script or Jupyter Notebook FILE against tests.

```
otter check [OPTIONS] FILE
```

Options

-v, --verbose
Verbosity of the logged output

-q, --question <question>
A specific quetsion to grade

-t, --tests-path <tests_path>
Path to the direcotry of test files

--seed <seed>
A random seed to be executed before each cell

Arguments

FILE
Required argument

13.1.3 export

Export a Jupyter Notebook SRC as a PDF at DEST with optional filtering.

If unspecified, DEST is assumed to be the basename of SRC with a .pdf extension.

```
otter export [OPTIONS] SRC [DEST]
```

Options

-v, --verbose
Verbosity of the logged output

--filtering
Whether the PDF should be filtered

--pagebreaks
: Whether the PDF should have pagebreaks between questions

-s, --save
: Save intermediate file(s) as well

-e, --exporter <exporter>
: Type of PDF exporter to use

Options
latex | html

--xecjk
: Enable xeCJK in Otter's LaTeX template

Arguments

SRC
: Required argument

DEST
: Optional argument

13.1.4 generate

Generate a zip file to configure an Otter autograder, including FILES as support files.

```
otter generate [OPTIONS] [FILES]...
```

Options

-v, --verbose
: Verbosity of the logged output

-t, --tests-dir <tests_dir>
: Path to test files

-o, --output-path <output_path>
: Path at which to write autograder zip file

-c, --config <config>
: Path to otter configuration file; ./otter_config.json automatically checked

--no-config
: Disable auto-inclusion of unspecified Otter config file at ./otter_config.json

-r, --requirements <requirements>
: Path to requirements.txt file; ./requirements.txt automatically checked

--no-requirements
: Disable auto-inclusion of unespecified requirements file at ./requirements.txt

--overwrite-requirements

Overwrite (rather than append to) default requirements for Gradescope; ignored if no REQUIREMENTS argument

-e, --environment <environment>

Path to environment.yml file; ./environment.yml automatically checked (overwrite)

--no-environment

Disable auto-inclusion of unespecified environment file at ./environment.yml

-l, --lang <lang>

Assignment programming language; defaults to Python

--username <username>

Gradescope username for generating a token

--password <password>

Gradescope password for generating a token

--token <token>

Gradescope token for uploading PDFs

--python-version <python_version>

Python version to use in the grading image

--channel-priority-strict

Whether to set conda's channel_priority to strict in the setup.sh file

--exclude-conda-defaults

Whether to exlucde conda's defaults channel from the environment.yml file

Arguments

FILES

Optional argument(s)

13.1.5 grade

Grade submissions in PATHS locally using Docker containers. PATHS can be individual file paths or directories containing submissions ending with extension EXT.

```
otter grade [OPTIONS] [PATHS]...
```

Options

-v, --verbose

Verbosity of the logged output

-n, --name <name>

An assignment name to use in the Docker image tag

-a, --autograder <autograder>

Path to autograder zip file

-o, --output-dir <output_dir>
Directory to which to write output

--ext <ext>
The extension to glob for submissions

Options
ipynb | py | Rmd | R | r | zip

--pdfs
Whether to copy notebook PDFs out of containers

--containers <containers>
Specify number of containers to run in parallel

--image <image>
A Docker image tag to use as the base image

--timeout <timeout>
Submission execution timeout in seconds

--no-network
Disable networking in the containers

--no-kill
Do not kill containers after grading

--debug
Run in debug mode (without ignoring errors thrown during execution)

--prune
Prune all of Otter's grading images

-f, --force
Force action (don't ask for confirmation)

Arguments

PATHS
Optional argument(s)

13.1.6 run

Run non-containerized Otter on a single submission.

```
otter run [OPTIONS] SUBMISSION
```

Options

-v, --verbose
Verbosity of the logged output

-a, --autograder <autograder>
Path to autograder zip file

-o, --output-dir <output_dir>
Directory to which to write output

--no-logo
Suppress Otter logo in stdout

--debug
Do not ignore errors when running submission

Arguments

SUBMISSION
Required argument

CHAPTER

FOURTEEN

RESOURCES

Here are some resources and examples for using Otter-Grader:

- Some demos for using Otter + R that demonstrate how to autograde R scripts, R Jupyter notebooks, and Rmd files, including how to use Otter Assign on the latter two formats.
- A quickstart guide to autograding R with Otter
- Otter-Grader JupyterLab Extension: A JupyterLab extension for authoring Otter Assign-formatted notebooks.
- Chris Pyles and Suraj Rampure presenting at Jupytercon 2023 along with a Jupyterlite demo file

Otter Grader is a light-weight, modular open-source autograder developed by the Data Science Education Program at UC Berkeley. It is designed to grade Python and R assignments for classes at any scale by abstracting away the autograding internals in a way that is compatible with any instructor's assignment distribution and collection pipeline. Otter supports local grading through parallel Docker containers, grading using the autograding platforms of 3rd-party learning management systems (LMSs), non-containerized grading on an instructor's machine, and a client package that allows students to check and instructors to grade assignments their own machines. Otter is designed to grade Python and R executables, Jupyter Notebooks, and RMarkdown documents and is compatible with a few different LMSs, including Canvas and Gradescope.

The core abstraction of Otter, as compared to other autograders like nbgrader and OkPy, is this: you provide the compute, and Otter takes care of the rest. All a instructor needs to do in order to autograde is find a place to run Otter (a server, a JupyterHub, their laptop, etc.) and Otter will take care of generating assignments and tests, creating and managing grading environents, and grading submissions. Otter is platform-agnostic, allowing you to put and grade your assignments anywhere you want.

Otter is organized into six components based on the different stages of the assignment pipeline, each with a command-line interface:

- *Otter Assign* is an assignment development and distribution tool that allows instructors to create assignments with prompts, solutions, and tests in a simple notebook format that it then converts into santized versions for distribution to students and autograders.
- *Otter Generate* creates the necessary setup files so that instructors can autograde assignments.
- *Otter Check* allows students to run publically distributed tests written by instructors against their solutions as they work through assignments to verify their thought processes and implementations.
- *Otter Export* generates PDFs with optional filtering of Jupyter Notebooks for manually grading portions of assignments.
- *Otter Run* grades students' assignments locally on the instructor's machine without containerization and supports grading on a JupyterHub account.
- *Otter Grade* grades students' assignments locally on the instructor's machine in parallel Docker containers, returning grade breakdowns as a CSV file.

CHAPTER

FIFTEEN

INSTALLATION

Otter is a Python package that is compatible with Python 3.6+. The PDF export internals require either LaTeX and Pandoc or wkhtmltopdf to be installed. Docker is also required to grade assignments locally with containerization. Otter's Python package can be installed using pip. To install the current stable version, install with

```
pip install otter-grader
```

Installing the Python package will install the `otter` binary so that Otter can be called from the command line. You can also call Otter as a Python module with `python3 -m otter`.

If you are going to be autograding R, you must also install the R package `ottr`:

```
install.packages("ottr")
```

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W