aga

Riley Shahar

Oct 03, 2023

CONTENTS

CHAPTER

ONE

TUTORIAL

1.1 Preliminaries

Ensure the aga CLI is available in your environment (`which aga`) and updated to the current version (`aga --version`). If not, you can install it with `pip` or a tool like `poetry`.

1.2 Getting Started

We're going to write a simple autograder for a basic problem: implementing a function to square an integer.

Whenever you write an aga autograder, you start by writing a reference implementation, which aga calls the *golden solution*. The library is based on the idea that reference implementations have uniquely beneficial properties for autograding homework; see motivation. So, here's our implementation:

```
def square(x: int) -> int:
    """Square x."""
    return x * x
```

The type annotations and docstring are just because they're good practice; as of now, aga does nothing with them. You might put, for example, the text of the problem that you're giving to students, so it's there for easy reference.

Now we need to tell aga to turn this into a problem. We do that with the `problem` decorator:

```
from aga import problem

@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

Aga's API is based around decorators; if you're not familiar with them, I suggest finding at least a brief introduction. `problem` will always be the first decorator you apply to any golden solution.

Now if we save this as `square.py`, we could run `aga gen square.py` in that directory, which would generate a `problem.zip` file. However, we're not quite done: we haven't given aga any test inputs yet! Let's do that:

```
from aga import problem, test_case

@test_case(-2)
@test_case(2)
@problem()
```

(continues on next page)

(continued from previous page)

```
def square(x: int) -> int:
    """Square x."""
    return x * x
```

Now re-run `aga gen square.py` and upload the resultant file to Gradescope.

There are a couple of things to know about this behavior.

First, there must be exactly one problem present in `square.py`. This is a limitation that will hopefully be relaxed in the future.

Second, while the student can upload any number of files, precisely one of them must contain a python object matching the name of the reference solution; in this case, `square` (note that the reference solution object's name is used even if another name is assigned to the problem itself via the `name` argument to the decorator). Otherwise, the solution will be rejected. It's extremely important to communicate this restriction to students.

Third, each test case will be run against the student's submission and the golden solution. If the outputs differ, the test will be marked as failing. The score of each test case will be half of the total score of the problem; by default, each test case has equal weight. Modifying this default will be discussed in Customizing Test Case Score.

You can use a similar syntax for multiple arguments, or keyword arguments:

```
@test_case(2, 1)  # defaults work as expected
@test_case(2, 1, sign=False)
@test_case(-3, 4, sign=False)
@problem()
def add_or_subtract(x: int, y: int, sign: bool = True) -> int:
    """If sign, add x and y; otherwise, subtract them."""
    if sign:
        return x + y
    else:
        return x - y
```

As a final note, you often won't want to upload the autograder to gradescope just to see the output that's given to students. You can use the `aga run` command to manually check a student submission in the command line.

1.3 Testing the Golden Solution

We still have a single point of failure: the golden solution. *Golden tests* are aga's main tool for testing the golden solution. They work like simple unit tests; you declare an input and expected output, which aga tests against your golden solution. We expect that any cases you want to use to test your golden solution will also be good test cases for student submissions, hence the following syntax:

```
@test_case(-2, aga_expect = 4)
@test_case(2, aga_expect = 4)
@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

Note that we prefix all keyword arguments to the `test_case` decorator with `aga_`, so that you can still declare test inputs for problems with actual keyword arguments.

`aga` can now check golden stdout now as well! Just add `aga_expect_stdout` to the test case(s). The format for the `aga_expect_stdout` is either a `str` or a `Iterable` of `str`.

When a `str` is given, the given string will be checked against all the captured output. When an `Iterable` is given, the captured output string will be divided using `splitlines`, meaning each string in the `Iterable` should contain NO `\n` characters.

The following examples will show.

```
@test_case(10, 20, aga_expect_stdout="the result is 30\n", aga_expect=30)
@problem()
def add(a: int, b: int) -> int:
    """Add two numbers."""
    print("the result is", a + b)
    return a + b
```

```
@test_case("Bob", aga_expect_stdout=["What is your name? ", "Hello, world! Bob!"])
@problem(script=True)
def hello_world() -> None:
    """Print 'Hello, world!'."""
    name = input("What is your name? ")
    print(f"Hello, world! {name}!")
```

If you run `aga check square`, it will run all golden tests (i.e., all test cases with declared `aga_expect`), displaying any which fail. This also happens by default when you run `aga gen square.py`, so you don't accidentally upload a golden solution which fails unit testing.

1.4 Customizing Test Case Score

By default, aga takes the problem's total score (configured on Gradescope) and divides it evenly among each problem. This division is weighted by a parameter, `aga_weight`, of `test_case`, which defaults to 1. If our total score is 20, and we want the 2 test case to be worth 15 and the -2 to be worth 5, we can do this:

```
@test_case(-2, aga_expect = 4)
@test_case(2, aga_expect = 4, aga_weight = 3)
@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

It is also possible to directly control the value of test cases:

```
@test_case(-2, aga_expect = 4)  # will get 100% of (total - 15) points
@test_case(2, aga_expect = 4, aga_weight = 0, aga_value = 15)
@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

However, this is not recommended, because it can lead to strange results if there is incongruity between the values assigned via aga and the total score assigned via Gradescope.

For complete semantics of score determination, see *Determining Score*.

1.5 Generating Test Cases

You can check out `examples/inputs_for_test_cases.py` in the GitHub repo for more complete examples and comparisons.

If we want many test cases, we probably don't want to enumerate all of them by hand. Aga therefore provides the `test_cases` decorator, which makes it easy to collect python generators (lists, `range`, etc.) into test cases.

Let's start by testing an arbitrary set of inputs:

```
from aga import problem, test_cases

@test_cases(-3, -2, 0, 1, 2, 100)
@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

This will generate six test cases, one for each element in the list. Test cases generated like this must share configuration, so while you can pass e.x. `aga_weight` to the decorator, it will cause *each* test case to have that weight, rather than dividing the weight among the test cases.

The `@test_cases(-3, -2, 0, 1, 2, 100)` is equivalent to

```
from aga import param, test_cases, problem

@test_cases(param(-3), param(-2), param(0), param(1), param(2), param(100))
@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

The directive `param` is used to wrap parameters to a function. Each `param` object is considered as a test case.

Similarly, we can generate tests for all inputs from -5 to 10:

```
@test_cases(*range(-5, 11))
@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

This will generate 16 test cases, one for each value in the range.

Or, we can generate tests programmatically, say from a file:

```
from typing import Iterator

def inputs() -> Iterator[int]:
    with open("INPUTS.txt", "r", encoding="UTF-8") as f:
        for s in f.readlines():
            yield int(s.strip())

@test_cases(*inputs())
@problem()
def square(x: int) -> int:
```

(continues on next page)

(continued from previous page)

```
    """Square x."""
    return x * x
```

The generation happens when you run `aga gen` on your local machine, so you can rely on resources (network, files, etc) not available in the Gradescope environment.

1.5.1 Multiple Arguments

Basics of Multiple Arguments

Say we want to generate inputs for multiple arguments (or keyword arguments), e.x. for a difference function. We can use the natural syntax:

```
@test_cases([(-3, 2), (-2, 1), (0, 0)], aga_params=True)
@problem()
def difference(x: int, y: int) -> int:
    """Compute x - y."""
    return x - y
```

There are four ways you can specify a batch of test cases: `params`, `zip` and `product`.

- `aga_params` will only take one iterable object, and each element in the iterable object will be unfolded when applied to the function. The example above will generate 3 tests, each to be `difference(-3, 2)`, `difference(-2, 1)` and `difference(0, 0)`. In the case where you want to add keyword arguments, you can use the `param` directive.

  ```
  from aga import problem, test_cases, param
  @test_cases([param(-3, y=2), param(-2, y=1), param(0, y=0)], aga_params=True)
  @problem()
  def difference(x: int, y: int) -> int:
      """Compute x - y."""
      return x - y
  ```

  which is equivalent to

  ```
  from aga import problem, test_cases, param
  @test_cases([(-3, 2), (-2, 1), (0, 0)], aga_params=True)
  @problem()
  def difference(x: int, y: int) -> int:
      """Compute x - y."""
      return x - y
  ```

- <no-flag> Note that this is different from the one above with `aga_params` flag. The example blow will generate 3 tests as well, but each to be `difference((-3, 2))`, `difference((-2, 1))` and `difference((0, 0))`.

```
@test_cases((-3, 2), (-2, 1), (0, 0))
@problem()
def difference(tp) -> int:
    """Compute x - y."""
    x, y = tp
    return x - y
```

- `aga_singular_params` works similarly to `aga_params`. The following code is equivalent to `difference((-3, 2))`, `difference((-2, 1))` and `difference((0, 0))`. (Note that the `aga_params` flag is not needed.)

```
from aga import problem, test_cases, param
@test_cases([(-3, 2), (-2, 1), (0, 0)], aga_singular_params=True)
@problem()
def difference(tp: Tuple[int, int]) -> int:
    """Compute x - y."""
    x, y = tp
    return x - y
```

It comes useful when you have a iterable of things where each single thing is going to serve as a parameter.

```
from aga import problem, test_cases, param
@test_cases(range(5), aga_singular_params=True)
@problem()
def square(x: int) -> int:
    """Compute x - y."""
    return x * x
```

The `@test_cases(range(5), aga_singular_params=True)` is equivalent to expanding the generator in the no flag version `@test_cases(*range(5))`. Note that `@test_cases(range(5), aga_params=True)` is not valid.

- `aga_product` will take the cartesian product of all the arguments. In the above example, there will be 15 test cases, one for each combination of the arguments.

```
@test_cases([-5, 0, 1, 3, 4], [-1, 0, 2], aga_product=True)
@problem()
def difference(x: int, y: int) -> int:
    """Compute x - y."""
    return x - y
```

- `aga_zip` will take the zip of all the arguments. In the example below, there will be 3 test cases, one for each pair of the arguments. This will short-circuit when the smaller iterator ends, so this will generate three test cases: `(-5, -1)`, `(0, 0)`, and `(1, 2)`.

```
@test_cases([-5, 0, 1, 3, 4], [-1, 0, 2], aga_zip=True)
@problem()
def difference(x: int, y: int) -> int:
    """Compute x - y."""
    return x - y
```

Shorthands

You will find typing all the `aga_product` etc. to be tedious. In that case, you can use the shorthands provided. There are two ways you can write it simpler.

-
```
from aga import problem, test_cases

@test_cases([-5, 0, 1, 3, 4], [-1, 0, 2])
@problem()
def fn() -> None:
```

(continues on next page)

(continued from previous page)

```
    # this is the same as @test_cases(...)
    ...

@test_cases.params([-5, 0, 1, 3, 4], [-1, 0, 2])
@problem()
def fn() -> None:
    # this is the same as @test_cases(..., aga_params=True)
    ...

@test_cases.product([-5, 0, 1, 3, 4], [-1, 0, 2])
@problem()
def fn() -> None:
    # this is the same as @test_cases(..., aga_product=True)
    ...

@test_cases.zip([-5, 0, 1, 3, 4], [-1, 0, 2])
@problem()
def fn() -> None:
    # this is the same as @test_cases(..., aga_zip=True)
    ...

@test_cases.singular_params(([-5, 0, 1, 3, 4], [-1, 0, 2]))
@problem()
def fn() -> None:
    # this is the same as @test_cases(..., aga_singular_params=True)
    ...
```

-

```
from aga import problem, test_cases_params, test_cases_product, test_cases_zip

@test_cases_params([-5, 0, 1, 3, 4], [-1, 0, 2])
@problem()
def fn() -> None:
    # this is the same as @test_cases(..., aga_params=True)
    ...

@test_cases_product([-5, 0, 1, 3, 4], [-1, 0, 2])
@problem()
def fn() -> None:
    # this is the same as @test_cases(..., aga_product=True)
    ...

@test_cases_zip([-5, 0, 1, 3, 4], [-1, 0, 2])
@problem()
def fn() -> None:
    # this is the same as @test_cases(..., aga_zip=True)
    ...
```

Note on aga_* keyword arguments

At this point, you might wonder what could be the input to `aga_*` keyword arguments. The good news is that you can do both singletons or iterables. When singleton is given, `aga` will match the number with the number of test cases. When an iterable is given, the number of elements must match the number of test cases and `aga` will check that.

Foe example, if you want to set a series of tests to hidden and define a bunch of golden outputs for them, we can do

```
@test_cases([1, 2, 3], aga_hidden=True, aga_expect=[1, 4, 9])
@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

`@test_cases(1, 2, 3, aga_expect=[1, 1, 4, 4, 9, 9])` since the numbers don't match.

1.6 Checking Scripts

Sometimes, submissions look like python scripts, meant to be run from the command-line, as opposed to importable libraries. To test a script, provide the `script=True` argument to the `problem` decorator:

```
@test_case("Alice", "Bob")
@test_case("world", "me")
@problem(script=True)
def hello_name() -> None:
    """A simple interactive script."""
    listener = input("Listener? ")
    print(f"Hello, {listener}.")

    speaker = input("Speaker ?")
    print(f"I'm {speaker}.")
```

This has three implications:

1. Aga will load the student submission as a script, instead of looking for a function with a matching name.
2. Aga will compare the standard output of the student submission to the standard output of the golden solution.
3. Aga will interpret the arguments to `test_case` as mocked outputs of the built in `input()` function. For example, for the "Alice","Bob" test case, aga will expect this standard output:

```
Hello, Alice.
I'm Bob.
```

1.7 Creating Pipelines

When testing against a class or an object, you can create a pipeline of functions to be called. This is useful if you want to test on the same object using different a sequence of actions.

A pipeline is a sequence of function (which sometimes is referred as a process) that accepts two inputs, the object it's testing on and the previous result generated by the proceeding function, and outputs a result. The pipeline will be run on the golden solution and students' solution, and the output results will be compared individually. You can create a pipeline from any of the following directives.

```
from aga import test_case, param, test_cases, problem
from aga.core.utils import initializer

def fn1(obj, previous_result):
    ...

def fn2(obj, previous_result):
    ...

@test_case.pipeline(initializer, fn1, fn2)
@test_cases(param.pipeline(initializer, fn1, fn2))
@problem()
class TestProblem:
    ...
```

The library provides several useful functions. They can be imported from `aga.core.utils`, like the `initializer` function above. One can use `initializer` to initialize the class under testing. Note that if you want to initialize the class with arguments, you can *ONLY* use `initializer`.

You can use the following linked list code as an example. It will generate a test case of multiple actions and outputs.

```
from __future__ import annotations
from aga import test_case, problem
from aga.core.utils import initializer, MethodCallerFactory, PropertyGetterFactory

prepend = MethodCallerFactory("prepend")
display = MethodCallerFactory("display")
pop = MethodCallerFactory("pop")
get_prop = PropertyGetterFactory()

actions_and_outputs = {
    initializer: None,
    prepend(10): None,
    display(): None,
    prepend(20): None,
    display(): None,
    prepend(30): None,
    display(): None,
    get_prop("first.value"): 30,
    get_prop("first", "next", "value"): 20,
    get_prop("first", ".next", ".value"): 20,
    get_prop(".first", "next", "value"): 20,
    pop(): 30,
    pop(): 20,
    pop(): 10,
}


class Node:
    """A node in a linked list."""

    def __init__(self, value: int, next_node: Node | None = None) -> None:
        self.value = value
```

(continues on next page)

(continued from previous page)

```
        self.next = next_node


@test_case.pipeline(
    *actions_and_outputs.keys(),
    aga_expect_stdout="< 10 >\n< 20 10 >\n< 30 20 10 >\n",
    aga_expect=list(actions_and_outputs.values()),
)
@problem()
class LL:
    """A linked list for testing."""

    def __init__(self) -> None:
        self.first: Node | None = None

    def __repr__(self) -> str:
        """Return a string representation of the list."""
        return f"< {self._chain_nodes(self.first)}>"

    def _chain_nodes(self, node: Node | None) -> str:
        if node is None:
            return ""
        else:
            return f"{node.value} {self._chain_nodes(node.next)}"

    def display(self) -> None:
        """Print the list."""
        print(self)

    def prepend(self, value: int) -> None:
        """Add a new element to the front of the list."""
        self.first = Node(value, self.first)

    def pop(self) -> int:
        """Remove the first element from the list and return it."""
        if self.first is None:
            raise IndexError("Cannot pop from an empty list")

        value = self.first.value
        self.first = self.first.next
        return value
```

CHAPTER

TWO

CONFIGURATION

Aga is configured in a simple `toml` format. By default, it looks for `aga.toml` in the current working directory; this is overridden by the `--config` CLI option.

Here is the full list of configuration options, with defaults:

```toml
# This file contains all the default configuration options.

[test]
# Configuration related to formatting and execution of test cases.

# The separator for test case name generation.
name_sep = ","

# The format string for generating test case names.
#
# Supported format specifiers:
#  - `args`: a separated list of the test case arguments, i.e. `0,3`
#  - `kwargs`: a separated list of the test case keyword arguments, i.e. `x=0,y=3`
#  - `sep`: a copy of the separator if there are both arguments and keyword arguments,
#           empty otherwise
name_fmt = "Test on {args}{sep}{kwargs}."

# The format string for generating failure messages.
#
# Supported format specifiers:
#  - `input`: a formatted repr of the test case inputs.
#  - `output`: the repr of the student submission's output.
#  - `expected`: the repr of the golden solution's output.
#  - `diff` if a diff is available (i.e. the output is a string), a text diff.
#  - `diff_explanation`: if a diff is available, the value of diff_explanation_msg, else␣
↪empty.
failure_msg = "Your submission didn't give the output we expected. We checked it with
↪{input} and got {output}, but we expected {expected}.{diff_explanation}{diff}"

# The format string for generating error messages.
#
# Supported format specifiers:
#  - `type`: the kind of python error, e.g. NameError.
#  - `message`: the error message.
#  - `traceback`: the error traceback.
error_msg = "A python {type} occured while running your submission: {message}.\n\nHere's␣
↪what was running when it happened:{traceback}."
```

(continues on next page)

(continued from previous page)

```
# The message to print if `check_stdout` is true and the stdouts differ.
#
# Supported format specifiers:
#  - `input`: a formatted repr of the test case inputs.
#  - `output`: the repr of the student submission's output.
#  - `expected`: the repr of the golden solution's output.
#  - `diff` a text diff.
#  - `diff_explanation`: the value of diff_explanation_msg.
stdout_differ_msg = "Your submission printed something different from what we expected.␣
↪We checked it with {input}.{diff_explanation}{diff}"

diff_explanation_msg = "\n\nHere's a detailed look at the difference between the strings.
↪ Lines starting with `-` are what we got from you, lines starting with `+` are what we␣
↪expected, and `_`s in lines starting with `?` denote characters that are different. Be␣
↪wary for spaces, which don't show up well in this format.\n\n"

[submission]
# Configuration related to student submissions.

# The global message to show if any tests failed.
failed_tests_msg = "It looks like some tests failed; take a look and see if you can fix␣
↪them!"

# The global message to show if any hidden tests failed.
failed_hidden_tests_msg = "Some of those tests were hidden tests, for which you won't␣
↪know the inputs. In the real world, we don't always know exactly how or why our code␣
↪is failing. Try to test edge cases and see if you can find the bugs!"

# The global message to show if no tests failed.
no_failed_tests_msg = "Great work! Looks like you're passing all the tests."

[loader]
# Configuration related to rerors loading student submissions.

# The message to show on errors that prevented the submission from being run.
#
# Supported format specifiers:
#  - `message`: the error message.
import_error_msg = "Looks like there's a python error in your code that prevented us␣
↪from running tests: {message}. Please fix this error, test your code again, and then␣
↪resubmit."

# The message to show if there's no symbol with the right name located.
#
# Supported format specifiers:
#  - `name`: the expected symbol name.
no_match_msg = "It looks like you didn't include the right object; we were looking for␣
↪something named `{name}`. Please resumbit with the correct name."

# The message to show if there's multiple symbols matching the expected name, i.e. in␣
↪multiple submitted files.
```

(continues on next page)

(continued from previous page)

```
#
# Supported format specifiers:
#  - `name`: the expected symbol name.
too_many_matches_msg = "It looks like multiple files you submitted have objects named `
↪{name}`; unfortunately, we can't figure out which one is supposed to be the real␣
↪submission. Please remove all but one of them and resumbit."

# The message to show if no script is found.
no_script_error_msg = "It looks like you didn't upload a python script. Please make sure␣
↪your script ends in `.py`."

# The message to show if multiple scripts are found.
multiple_scripts_error_msg = "It looks like you uploaded multiple python scripts. Please␣
↪make sure you only upload one file ending in `.py`."


[problem]
# Configration for problem settings.

# If true, check that the stdout of the problem and submission both match.
check_stdout = false

# If true, test case arguments will be interpreted as outputs for successive calls of␣
↪`input()`.
mock_input = false
```

CHAPTER

THREE

INJECTION

3.1 What is injection and why?

Users of aga find they need to copy and paste snippets of scripts to each of problem description python file, which is creating a lot of redundant code. Take the following example. The prize_fn has to be copied every time a new problem is created.

```
def prize_fn(tests: list[TcOutput], _: SubmissionMetadata) -> tuple[float, str]:
    """Check that all tests passed."""
    # HUNDREDS OF LINES OF CODE HERE !!!!!
    if all(t.is_correct() for t in tests):
        return 1.0, "Good work! You earned these points since all tests passed."
    else:
        return 0.0, "To earn these points, make sure all tests pass."


@prize(prize_fn, value=10)
@problem()
def add(x: int, y: int) -> int:
    """Add x and y."""
    return x + y
```

To solve this problem, we introduce the concept of injection, so that the shared code can be written in one place and be injected in every problem description file. So that the code above can be rewritten as follows, and no duplicated code will be generated.

```
# shared_prize_func.py

def prize_fn(tests: list[TcOutput], _: SubmissionMetadata) -> tuple[float, str]:
    """Check that all tests passed."""
    # HUNDREDS OF LINES OF CODE HERE !!!!!
    if all(t.is_correct() for t in tests):
        return 1.0, "Good work! You earned these points since all tests passed."
    else:
        return 0.0, "To earn these points, make sure all tests pass."
```

```
# problem 1
# ... necessary imports
from aga.injection import prize_fn
```

(continues on next page)

(continued from previous page)

```
@prize(prize_fn, value=10)
@problem()
def add(x: int, y: int) -> int:
    """Add x and y."""
    return x + y
```

```
# problem 2
# ... necessary imports
from aga.injection import prize_fn

@prize(prize_fn, value=10)
@problem()
def multiply(x: int, y: int) -> int:
    """Multiply x and y."""
    return x * y
```

3.2 How to use injection

There are several commands related to injection. You can find the help and description in the CLI help message. It's duplicated down below for the convenience of reading.

```
 --inject                    PATH  Inject a util file into the submission directory.
 --inject-all                PATH  Inject all util files in the specified folder into␣
↪the submission directory.
 --injection-module          TEXT  The name of the module to import from the injection␣
↪directory. [default: injection]
 --auto-inject                     Find the first injection directory recursively and␣
↪automatically.
```

You can specify a specific file to inject using `--inject <file_path>` or inject all files in a folder using `--inject-all <dir_path>`. You can also specify the name of the injection module, which is defaulted to `injection` so that the injection imports will be `from aga.injection import ...`. When changed to `my_injection` for example, it will make the import command to be `from aga.my_injection import ...`.

You can also use the `--auto-inject` flag to automatically find ***the first injection directory*** (this will likely be changed to ***all injection directories*** in the future) upward recursively. `aga` finds `aga_injection` folder starting from the current working directory, which is the folder in which you entered `aga gen/check/run` commands. For example, considering the following dir tree:

```
.
└── courses/
    └── csci121/
        ├── hw1/
        │   ├── aga_injection/
        │   │   └── jims_prize_fn.py
        │   └── pb1.py
        ├── hw2/
        │   ├── aga_injection/
        │   │   └── jams_prize_fn.py
```

(continues on next page)

(continued from previous page)

```
        │   └── pb2.py
        └── aga_injection/
            └── jems_prize_fn.py
```

If `aga check --auto-inject pb1.py` is run in `hw1` directory, `jims_prize_fn.py` will be used. However, if `aga check --auto-inject ./hw1/pb1.py` is run in the `csci121` directory, `jems_prize_fn.py` will be used.

CHAPTER

FOUR

ADVANCED FEATURES

4.1 Prizes

If you want finer control over the points allocation of problems, you can add points prizes to them, which let you run custom functions over the list of completed tests in order to assign points values:

```
from aga import problem, test_case
from aga.prize import prize, TcOutput, SubmissionMetadata

def all_correct(
    tests: list[TcOutput], _: SubmissionMetadata
) -> tuple[float, str]:
"""Check that all tests passed."""
    if all(t.is_correct() for t in tests):
        return 1.0, "Good work! You earned these points since all tests passed."
    else:
        return 0.0, "To earn these points, make sure all tests pass."

@prize(all_correct, name="Prize")
@test_case(0)
@test_case(2)
@problem()
def square(x: int) -> int:
    """Square x."""
                return x * x
```

If only one of the `0` or `2` test cases pass, the student will receive 1/3 credit for this problem. If both pass, they will receive full credit.

We provide more details and several pre-written prize functions in the `prize`(reference.html#module-aga.prize) documentation.

4.2 Overriding the Equality Check

By default, aga uses unittest's `assertEqual`, or `assertAlmostEqual` for floats, to test equality. This can be overridden with the `aga_override_check` argument to `test_case`. This argument takes a function of three arguments: a `unittest.TestCase` object (which you should use to make assertions), the golden solution's output, and the student submission output. For example, to test a higher-order function:

```
from typing import Callable

from aga import problem, test_case

def _make_n_check(case, golden, student):
    # here `golden` and `student` are the inner functions returned by the
    # submissions, so they have type int -> int`
    for i in range(10):
        case.assertEqual(golden(i), student(i), f"Solutions differed on input {i}.")

@test_cases(-3, -2, 16, 20, aga_override_check=_make_n_check)
@test_case(0, aga_override_check=_make_n_check)
@test_case(2, aga_override_check=_make_n_check)
@problem()
def make_n_adder(n: int) -> Callable[[int], int]:
    def inner(x: int) -> int:
        return x + n
    return inner
```

4.3 Overriding the Entire Test

If you want even more granular control, you can also override the entire test. The `aga_override_test` argument to `test_case` takes a function of three arguments: the same `unittest.TestCase` object, the golden solution (the solution itself, *not* its output), and the student solution (ditto). For example, to mock some library:

```
from unittest.mock import patch

from aga import problem, test_case


def mocked_test(case, golden, student):
    with patch("foo") as mocked_foo:
        case.assertEqual(golden(0), student(0), "test failed")


@test_case(aga_override_test=mocked_test)
@problem()
def call_foo(n):
    foo(n)
```

A common use-case is to disallow the use of certain constructs. For convenience, aga provides the `Disallow` class. For example, to force the student to use a `lambda` instead of a `def`:

```
import ast

from aga import problem, test_case
from aga.checks import Disallow

# I recommend you use `aga_name` here, because the generated one won't be very good
@test_case(
    aga_name="Use lambda, not def!",
    aga_override_test=Disallow(nodes=[ast.FunctionDef]).to_test()
)
@problem()
def is_even_lambda(x: int) -> bool:
    return x % 2 == 0
```

For full details on `Disallow`, see the reference.

If you wish to write your own checks, you can use the methods provided by `unittest.TestCase`. For example, the override function can be written as:

```
def my_check(case, golden, student):
    case.assertEqual(golden(*case.args), student(*case.args), "test failed")
```

The `case` exposes `args` arguments and `kwargs` variables which are passed from `test_case` decorator. For example, `test_case(3, 4, z = 10)` will create a case with `args = (3, 4)` and `kwargs = {"z": 10}`. All the `aga_*` kwargs will be strip away in the building process.

The `case` also exposes `name` and `description` variables which are the name of the test case and the description of the test case. Changing those variables is equivalent to changing `aga_name` and `aga_description` but this means you can set it dynamically during the testing.

4.4 Capture Context Values

Sometimes a piece of assignment file includes multiple classes, and even though only one class is eventually tested, the other parts of students' answers can be crucial. For example, consider the following file. You can specify in the `ctx` argument of `problem` decorator to capture the `GasStation` class, and in the override check function, you can reference the `GasStation` class in the student's answer.

```
from aga import problem, test_case

def override_check(case, golden, student):
    # use case.ctx.GasStation to reference student's GasStation class implementation
    ...


@test_case(aga_override_check=override_check)
@problem(ctx=['GasStation'])
class Car:
    # uses gas station somewhere in the code
    ...

class GasStation:
    ...
```

Essentially, `ctx` argument takes in an iterable of strings, and aga will search the corresponding fields in the students' submitted module (file).

Note that `ctx` is should not be modified during overriden check functions, since the changes will persist to all the following test cases, which might not be the intended behavior.

CHAPTER

FIVE

DETERMINING SCORE

This section describes the complete semantics for computing the score of a test case. Each test case is scored as all-or-nothing.

Each problem is sorted into a specific group by virtue of the `group` decorator. A group consists of every test case or prize (prizes and test cases work the same for the purposes of this algorithm) underneath that group, and before the next group decorator. There is an implicit group consisting of all test cases preceding the first decorator. For example, in the following setup, there are three groups; one consists of the negative cases, one of `0`, and one of the positive cases.

```
@test_case(-2)
@test_case(-1)
@group()
@test_case(0)
@group()
@test_case(1)
@test_case(2)
@problem()
def square(x: int) -> int:
    return x * x
```

Each group is first assigned a total score, and then each test case in a group is assigned a score. These processes work identically; we will think of either a group or a test case as a scorable object. Scorable objects possess a *value* (default 0), which is absolute, and a *weight* (default 1), which is relative. There is some pot of points, the *total score* which is available as an input to the algorithm; this is determined by the classroom frontend for the group case, and the output of the group algorithm for the indivial test case.

For each object, the algorithm first sets its score to its value, decrementing the total score by that value. The algorithm allows for the sum of values to potentially be larger than the total available score; in this case, extra credit will be available, and relative weights will have no effect. The algorithm then divides the remaining total score according to weight.

For example, consider the following problem with total score 20.

```
@test_case(-2, aga_weight = 2)
@test_case(-1, aga_weight = 0, aga_value = 2.0)
@test_case(0, aga_weight = 2, aga_value = 4.0)
@test_case(1, aga_value = 2.0)
@test_case(2)
@problem()
def square(x: int) -> int:
    return x * x
```

Every test case is in the implicit group, which has weight one and value zero, and so it is assigned all 20 points. We have the following weights and values:

Case	Weight	Value
-2	2	0.0
-1	0	2.0
0	2	4.0
1	1	2.0
2	1	0.0

First, processing values leaves total score 12 and gives the following temporary scores:

Case	Score
-2	0.0
-1	2.0
0	4.0
1	2.0
2	0.0

Next, we divide the remaining 12 units of score amongst the 6 units of weight, so each unit of weight represents 2 units of score. This give the final scores.

Case	Score
-2	4.0
-1	2.0
0	8.0
1	4.0
2	2.0

CHAPTER

SIX

REFERENCE

Aga grades assignments: a library for easily producing autograders for code.

Anything not explicitly documented here should not be used directly by clients and is only exposed for testing, the CLI, and type hinting.

aga.**group**(*weight=1*, *value=0.0*, *extra_credit=0.0*)
Declare a group of problems.

Parameters

- **weight** (*int*) – The group's relative weight to the problem's score. See *Determining Score* for details.
- **value** (*float*) – The group's absolute score. See *Determining Score* for details.
- **extra_credit** (*float*) – The group's extra credit. See *Determining Score* for details.

Returns A decorator which adds the group to a problem.

Return type Callable[[*Problem*[T]], *Problem*[T]]

aga.**param**
alias of *aga.core.parameter._TestParam*

aga.**problem**(*name=None*, *script=False*, *check_stdout=None*, *mock_input=None*, *ctx=()*)
Declare a function as the golden solution to a problem.

This method should decorate a function which is known to produce the correct outputs, which we refer to as the "golden solution". It also provides facilities for testing that solution, via golden test cases, constructed by passing the output argument to the test_case decorator.

Parameters

- **name** (*Optional[str]*) – The problem's name. If None (the default), the wrapped function's name will be used.
- **script** (*bool*) – Whether the problem represents a script, as opposed to a function. Implies `check_stdout` and `mock_input` unless they are passed explicitly.
- **check_stdout** (*Optional[bool]*) – Overrides the `problem.check_stdout` configuration option. If True, check the golden solution's stdout against the student submission's for all test cases.
- **mock_input** (*Optional[bool]*) – Overrides the `problem.mock_input` configuration option. If True, test cases for this problem will be interpreted as mocked outputs of `builtins.input`, rather than inputs to the function.
- **ctx** (*Iterable[str]*) – The context values required in the submission and will be captured

Returns A decorator which turns a golden solution into a problem.

Return type Callable[[Callable[ProblemInput, T]], *Problem*[T]]

aga.**test_case**
alias of *aga.core.parameter._TestParam*

aga.**test_cases**
alias of *aga.core.parameter._TestParams*

6.1 Core

The core library functionality.

class aga.core.**AgaTestCase**(*test_input*, *golden*, *under_test*, *metadata*)
A TestCase which runs a single test of a Problem.

property description: str | None
Get the problem's description.

Return type UnionType[str, None]

property metadata: *aga.core.suite.TestMetadata*
Get the test's metadata.

Return type *TestMetadata*

property name: str
Format the name of the test case.

Return type str

runTest()
Run the test case.

Return type None

shortDescription()
Dynamically generate the test name.

This method is called by unittest.

Return type str

property test_input: aga.core.suite._TestInputs[aga.core.suite.Output]
Get the test input.

Return type _TestInputs[~Output]

class aga.core.**AgaTestSuite**(*config*, *tests*)
A thin wrapper around TestSuite that store a config.

class aga.core.**Problem**(*golden*, *name*, *config*, *is_script*, *ctx_targets=()*)
Stores tests for a single problem.

add_group(*grp*)
Add a group to the problem.

Return type None

add_prize(*prize*)
Add a prize to the current group.

Return type None

add_test_case(*param*)
Add a test case to the current group.

Student solutions will be checked against the golden solution; i.e., this method does _not_ produce a test of the golden solution.

Return type None

check()
Check that the problem is correct.

Currently, this runs all tests of the golden solution.

Return type None

config()
Get access to the problem's config.

Return type AgaConfig

expected_symbol()
Get the name of the symbol that should be tested against.

Return type str

generate_test_suite(*under_test*, *metadata*)
Generate a TestSuite for the student submitted function.

Neither the generated test suite nor the body of this function will run golden tests; instead, golden test cases are treated as equivalent to ordinary ones. To test the golden function, check should be used instead.

Parameters

- **under_test** (*Callable[ProblemParamSpec, ProblemOutputType]*) – The student submitted function.
- **metadata** (SubmissionMetadata) – The submission metadata.

Return type tuple[*AgaTestSuite*, list[ScoredPrize]]

Returns

- *AgaTestSuite* – A unittest test suite containing one test for each TestInput in this problem, checking the result of the problem's golden solution against under_test.
- *list[ScorePrize]* – The prizes for the problem, with scores assigned.

property golden: Callable[[~ProblemParamSpec], aga.core.problem.ProblemOutputType]
The gold solution property.

Return type Callable[[ParamSpec], ~ProblemOutputType]

name()
Get the problem's name.

Return type str

property submission_context: aga.core.context.SubmissionContext
The environment values captured from the problem module.

Return type SubmissionContext

update_config_weak(*config*)
Update any non-default items in self.config.

Return type None

class aga.core.**SubmissionMetadata**(*total_score*, *time_since_due*, *previous_submissions*)
Metadata for testing a submission, collected from the frontend.

total_score
The problem's total score.

Type float

time_since-due
The delta _from_ the due date _to_ the submission date, i.e. it's negative if the problem was submitted before the due date.

Type timedelta

previous_submissions
The number of previous submissions.

Type int

is_on_time()
Return true of the submission was on time.

Return type bool

class aga.core.**TestMetadata**(*max_score*, *config*, *check_stdout*, *mock_input*, *hidden=False*)
Stores metadata about a specific test case.

aga.core.**group**(*weight=1*, *value=0.0*, *extra_credit=0.0*)
Declare a group of problems.

Parameters

- **weight** (*int*) – The group's relative weight to the problem's score. See *Determining Score* for details.
- **value** (*float*) – The group's absolute score. See *Determining Score* for details.
- **extra_credit** (*float*) – The group's extra credit. See *Determining Score* for details.

Returns A decorator which adds the group to a problem.

Return type Callable[[*Problem*[T]], *Problem*[T]]

aga.core.**param**
alias of *aga.core.parameter._TestParam*

aga.core.**problem**(*name=None*, *script=False*, *check_stdout=None*, *mock_input=None*, *ctx=()*)
Declare a function as the golden solution to a problem.

This method should decorate a function which is known to produce the correct outputs, which we refer to as the "golden solution". It also provides facilities for testing that solution, via golden test cases, constructed by passing the output argument to the test_case decorator.

Parameters

- **name** (*Optional[str]*) – The problem's name. If None (the default), the wrapped function's name will be used.
- **script** (*bool*) – Whether the problem represents a script, as opposed to a function. Implies `check_stdout` and `mock_input` unless they are passed explicitly.
- **check_stdout** (*Optional[bool]*) – Overrides the `problem.check_stdout` configuration option. If True, check the golden solution's stdout against the student submission's for all test cases.

- **mock_input** (*Optional[bool]*) – Overrides the `problem.mock_input` configuration option. If True, test cases for this problem will be interpreted as mocked outputs of `builtins.input`, rather than inputs to the function.
- **ctx** (*Iterable[str]*) – The context values required in the submission and will be captured

Returns A decorator which turns a golden solution into a problem.

Return type Callable[[Callable[ProblemInput, T]], *Problem*[T]]

aga.core.**test_case**
alias of *aga.core.parameter._TestParam*

aga.core.**test_cases**
alias of *aga.core.parameter._TestParams*

6.2 Core - Parameters

class aga.core.parameter.**_TestParam**(**args: Any*, *aga_expect: Any = 'None'*, *aga_expect_stdout: Optional[Union[str, Sequence[str]]] = 'None'*, *aga_hidden: bool = 'False'*, *aga_name: str | None = 'None'*, *aga_description: str | None = 'None'*, *aga_weight: int = '1'*, *aga_value: float = '0.0'*, *aga_extra_credit: float = '0.0'*, *aga_override_check: Optional[Callable[[...], Any]] = 'None'*, *aga_override_test: Optional[Callable[[...], Any]] = 'None'*, *aga_is_pipeline: bool = 'False'*, ***kwargs: Any*)
class aga.core.parameter.**_TestParam**(**args: Any*, ***kwargs: Any*)

property aga_kwargs: aga.core.parameter.AgaKeywordDictType
Return the aga_* keyword arguments of the test.

Return type AgaKeywordDictType

aga_kwargs_repr(*sep=','*)
Return a string representation of the test's aga_* keyword arguments.

Return type str

property args: Tuple[Any, ...]
Return the arguments to be passed to the functions under test.

Return type Tuple[Any, ...]

args_repr(*sep=','*)
Return a string representation of the test's arguments.

Return type str

property description: str | None
Get the description of the test case.

Return type UnionType[str, None]

ensure_aga_kwargs()
Ensure that the aga_* keywords are handled correct.

Return type AgaKeywordContainer

ensure_default_aga_values()
Ensure that the aga_* keywords all have default.

Return type AgaKeywordContainer

ensure_valid_kwargs()
Ensure that the aga_* keywords are handled correct.

Return type *_TestParam*

property expect: Any
Get the expected aga_expect of the test case.

Return type Any

property expect_stdout: str | None
Get the expected aga_expect_stdout of the test case.

Return type UnionType[str, None]

property extra_credit: float
Get the extra credit aga_extra_credit of the test case.

Return type float

generate_test_case(*prob*)
Generate a test case for the given problem.

Return type *Problem*[ProblemParamSpec, ProblemOutputType]

property hidden: bool
Get the hidden aga_hidden of the test case.

Return type bool

property is_pipeline: bool
Get the is_pipeline aga_is_pipeline of the test case.

Return type bool

property kwargs: Dict[str, Any]
Return the keyword arguments to be passed to the functions under test.

Return type Dict[str, Any]

kwargs_repr(*sep=','*)
Return appropriate string representation of the test's keyword arguments.

Return type str

property name: str | None
Get the name of the test case.

Return type UnionType[str, None]

property override_check: Optional[Callable[[...], Any]]
Get the override_check aga_override_check of the test case.

Return type Optional[Callable[..., Any], None]

property override_test: Optional[Callable[[...], Any]]
Get the override_test aga_override_test of the test case.

Return type Optional[Callable[..., Any], None]

sep_repr(*sep=','*)
Return sep if both exist, "" otherwise.

Return type str

update_aga_kwargs(***kwargs*)
Update the keyword arguments to be passed to the functions under test.

Return type AgaKeywordContainer

property value: float
Get the value aga_value of the test case.

Return type float

property weight: int
Get the weight aga_weight of the test case.

Return type int

class aga.core.parameter.**_TestParams**(**args: Any*, *aga_expect: Any = 'None'*, *aga_expect_stdout: Optional[Union[str, Sequence[str]]] = 'None'*, *aga_hidden: bool = 'False'*, *aga_name: str | None = 'None'*, *aga_description: str | None = 'None'*, *aga_weight: int = '1'*, *aga_value: float = '0.0'*, *aga_extra_credit: float = '0.0'*, *aga_override_check: Optional[Callable[[...], Any]] = 'None'*, *aga_override_test: Optional[Callable[[...], Any]] = 'None'*, *aga_is_pipeline: bool = 'False'*, *aga_product: bool = 'False'*, *aga_zip: bool = 'False'*, *aga_params: bool = 'False'*, *aga_singular_params: bool = 'False'*, ***kwargs: Any*)

class aga.core.parameter.**_TestParams**(**args: Any*, ***kwargs: Any*)
A class to store the parameters for a test.

static add_aga_kwargs(*aga_kwargs*, *final_params*)
Add aga_kwargs to the finalized parameters.

Return type None

params: ClassVar[functools.partial[*aga.core.parameter._TestParams*]] = functools.partial(<class 'aga.core.parameter._TestParams'>, aga_params=True)

static parse_no_flag(**args*, ***kwargs*)
Parse the parameters for no flag.

Return type List[*_TestParam*]

static parse_params(**args*, ***kwargs*)
Parse the parameters for param sequence.

Return type List[*_TestParam*]

static parse_singular_params(**args*, ***kwargs*)
Parse the parameters for param sequence.

Return type List[*_TestParam*]

static parse_zip_or_product(**args*, *aga_product=False*, *aga_zip=False*, ***kwargs*)
Parse parameters for zip or product.

Return type List[*_TestParam*]

product: ClassVar[functools.partial[*aga.core.parameter._TestParams*]] = functools.partial(<class 'aga.core.parameter._TestParams'>, aga_product=True)

singular_params: ClassVar[functools.partial[*aga.core.parameter._TestParams*]] = functools.partial(<class 'aga.core.parameter._TestParams'>, aga_singular_params=True)

zip: ClassVar[functools.partial[aga.core.parameter._TestParams]] = functools.partial(<class 'aga.core.parameter._TestParams'>, aga_zip=True)

6.3 Prizes

Add points prizes to problems.

This module contains the `prize` decorator, which lets you define custom post-test-run points hooks for things like correctness and lateness. It also contains several prizes, defined for convenience.

class aga.prize.**SubmissionMetadata**(*total_score*, *time_since_due*, *previous_submissions*)

Metadata for testing a submission, collected from the frontend.

total_score

The problem's total score.

Type float

time_since-due

The delta _from_ the due date _to_ the submission date, i.e. it's negative if the problem was submitted before the due date.

Type timedelta

previous_submissions

The number of previous submissions.

Type int

is_on_time()

Return true of the submission was on time.

Return type bool

class aga.prize.**TcOutput**(*score*, *max_score*, *name*, *status=None*, *hidden=False*, *description=None*, *error_description=None*)

Stores information about a completed test case.

score

The test's score.

Type float

max_score

The max score for the test.

Type float

name

The test's name.

Type str

description

Human-readable text description of the test. Some frontends distinguish between no output and empty output, i.e. in terms of showing UI elements.

Type Optional[str]

error_description

Human-readable error description of the test.

Type Optional[str]

hidden
The test's visibility.

Type bool

static format_description(*desc*)
Format a description.

Return type str

static format_error_description(*error*)
Format an error description.

Return type str

static format_rich_output(*description=None*, *error_description=None*)
Format a rich output.

Return type str

is_correct()
Check whether the problem received full credit.

Return type bool

property rich_output: str
Output of all the descriptions.

Return type str

aga.prize.**all_correct**(*tests*, *_*)
1.0 if all tests passed, 0.0 otherwise.

For use as a prize.

Return type tuple[float, str]

aga.prize.**correct_and_on_time**(*tests*, *metadata*)
1.0 if the submission was correct and passed all tests, 0.0 otherwise.

For use as a prize.

Return type tuple[float, str]

aga.prize.**on_time**(*_*, *metadata*)
1.0 if the submission was on time, 0.0 otherwise.

For use as a prize.

Return type tuple[float, str]

aga.prize.**prize**(*criteria*, *name='Prize'*, *weight=1*, *value=0.0*, *extra_credit=0.0*, *hidden=False*)
Add a points prize to the problem.

Parameters

- **criteria** (*Callable[[list[TcOutput], SubmissionMetadata], tuple[float, str]*) – The criteria for awarding the prize's points. The first returned value should be a float from 0 to 1 which determines the fraction of points to assign. The second should be a string which will be displayed to the student.
- **name** (*str*) – The name of the prize, to be displayed to the student.
- **weight** (*int*) – The prize's weight. See *Determining Score* for details.
- **value** (*int*) – The prize's absolute score. See *Determining Score* for details.

- **extra_credit** (*int*) – The prize's extra credit. See *Determining Score* for details.
- **hidden** (*bool*) – Whether the prize should be hidden from the student.

Returns A decorator which adds the prize to a problem.

Return type Callable[[*Problem*[T]], *Problem*[T]]

6.4 Checks

Additional checks and filters for problems.

class aga.checks.**Disallow**(*functions=None*, *binops=None*, *nodes=None*)
A list of items to disallow in code.

functions
The names of functions which the student should not be able to call.

Type list[str]

binops
The types of binary operations wihch the student should not be able to use. E.x., to forbid floating-point division, use `ast.Div`. See here for a list.

Type list[type]

nodes
The types of any ast nodes wihch the student should not be able to use. E.x., to forbid for loops, use `ast.For`. See the docs for a list.

Type list[type]

Examples

To disallow the built-in `map` function: `Disallow(functions=["map"])`.

To disallow the built-in `str.map` function: `Disallow(functions=["count"])`. Note that for class method names, you just use the name of the function.

Note that there is no way to disallow += without also disallowing + with this API.

search_on_object(*obj*)
Search for disallowed AST objects in a python object.

Return type `Iterable[tuple[str, int]]`

search_on_src(*src*)
Search for disallowed AST objects in a source string.

Return type `Iterable[tuple[str, int]]`

to_test()
Generate a test method suitable for `aga_override_test` of `test_case`.

You can pass the output of this method directly to `aga_override_test`.

You can also use the lower-level methods `search_on_object` or `search_on_src` if you want to generate your own error message.

Return type `Callable[[TestCase, Callable[..., ~Output], Callable[..., ~Output]], None]`

aga.checks.**Site**
alias of tuple[str, int]

CHAPTER

SEVEN

COMMAND-LINE INTERFACE

The command-line interface allows checking (via test cases with provided `aga_expect`) the validity of golden solutions, as well as generating the autograder file from a problem.

7.1 CLI Reference

7.1.1 aga

```
aga [OPTIONS] COMMAND [ARGS]...
```

Options

--install-completion
Install completion for the current shell.

--show-completion
Show completion for the current shell, to copy it or customize the installation.

check

Check a problem against test cases with an `aga_expect`.

```
aga check [OPTIONS] SOURCE
```

Options

-c, --config <config_file>
The path to the aga config file.

Default aga.toml

--inject <inject>
Inject a util file into the submission directory.

Default

--inject-all <inject_all>
Inject all util files in the specified folder into the submission directory.

Default

--injection-module <injection_module>
The name of the module to import from the injection directory.

Default injection

--auto-inject
Find the first injection directory recursively and automatically.

Default False

Arguments

SOURCE
Required argument

gen

Generate an autograder file for a problem.

```
aga gen [OPTIONS] SOURCE
```

Options

-f, --frontend <frontend>
The frontend to use. Currently only gradescope is supported.

Default gradescope

-o, --output <output>
The path to place the output file(s).

-c, --config <config_file>
The path to the aga config file.

Default aga.toml

--inject <inject>
Inject a util file into the submission directory.

Default

--inject-all <inject_all>
Inject all util files in the specified folder into the submission directory.

Default

--injection-module <injection_module>
The name of the module to import from the injection directory.

Default injection

--auto-inject
Find the first injection directory recursively and automatically.

Default False

Arguments

SOURCE
Required argument

run

Run the autograder on an example submission.

```
aga run [OPTIONS] SOURCE SUBMISSION
```

Options

-c, --config <config_file>
The path to the aga config file.

Default aga.toml

--points <points>
The total number of points for the problem.

Default 20.0

--due <due>
The problem due date.

Default now

--submitted <submitted>
The problem submission date.

Default now

--previous_submissions <previous_submissions>
The number of previous submissions.

Default 0

--inject <inject>
Inject a util file into the submission directory.

Default

--inject-all <inject_all>
Inject all util files in the specified folder into the submission directory.

Default

--injection-module <injection_module>
The name of the module to import from the injection directory.

Default injection

--auto-inject
Find the first injection directory recursively and automatically.

Default False

Arguments

SOURCE
Required argument

SUBMISSION
Required argument

CHAPTER

EIGHT

DEVELOPMENT

We have tooling for a modern development workflow provided in an environment based around `poetry`. If there's another workflow you like better, feel free to use it, but just make sure you're writing good code, passing tests, and not introducing additional linter errors. In particular, I will enforce conformance with `black`.

8.1 Setup

To set up the development environment:

1. Clone the repo: `git clone git@github.com:nihilistkitten/aga.git && cd aga`.
2. Install poetry.
3. Install dependencies: `poetry install`.
4. Activate the development environment: `poetry shell`.

I encourage you to set up integration between our dev tools and your editor, but it's not strictly necessary; you can use them as you please, from their CLIs or (I suppose) not at all. Regardless, the environment includes python-lsp-server, which I personally use for this purpose, and can be used via `lsp-mode` in emacs, `atom-languageclient` in atom, or the built-in lsp support in neovim and vscode.

8.2 Nox

The tool nox runs tooling in virtualized environments. To both run tests and lints, run `nox -r` (`-r` prevents nox from reinstalling the environments across multiple runs, which saves significant time.)

8.3 Testing

Testing depends on pytest. To run tests, simply run `pytest` from within the poetry shell. To run tests via nox, run `nox -rs test`.

Code coverage information can be generated by `pytest --cov`. This happens by default in nox runs.

There are some network-bound end-to-end tests which are not fun by default. You can run these with `pytest -m slow` or `nox -rs test_slow`.

8.4 Linting

A number of static analysis tools are available in the development environment:

- mypy, a static type analysis tool.
- pylint, a general-purpose linter.
- flake8, a highly modular linter.
- flake8-black, a code formatting
- checker. flake8-bugbear, which
- makes `flake8` stricter. pydocstyle, a
- documentation linter.

These tend to be quite strict, but after a while you'll get used to them, and they help write *much* better code.

To run all lints, run `nox -rs lint`.

8.5 Formatting

We use two tools to enforce a uniform code style:

- black, a highly opinionated code formatter.
- isort, an import sorter.

To run both formatters, run `nox -rs fmt`. This is *not* run by default runs of nox.

CHAPTER

NINE

MAINTENANCE

Here I describe how to do common/regular maintenance tasks.

9.1 Bump python version

We like to keep the default python version under which tests are run as the most recent stable version (currently 3.10), so that students don't unknowingly rely on new language features and have to debug versioning differences between their machine and the autograder. Right now, to fix this, we need to:

1. Update the documentation: just grep for `3.10` and replace it with the new version.
2. Update the gradescope build: this is handled in `setup.sh`; you need to install a different version and change the version we execute the scripts with.
3. Add the new version to be tested in the `noxfile`.
4. Adjust the shebang of the `run_autograder` executable.
5. Adjust `.readthedocs.yml` to build the docs on the newest python.

9.2 Add dependencies

Right now, we have a kind of janky setup where we maintain our own `setup.py` for installing the library on gradescope, in `aga/resources/gradescope/setup.py`. Whenever we add a dependency, we need to update this file accordingly.

CHAPTER

TEN

LICENSE

CHAPTER

ELEVEN

AGA GRADES ASSIGNMENTS

aga (**a**ga **g**rades **a**ssignments) is a tool for easily producing autograders for python programming assignments, originally developed for Reed College's CS1 course.

11.1 Motivation

Unlike traditional software testing, where there is likely no *a priori* known-correct implementation, there is always such an implementation (or one can be easily written by course staff) in homework grading. Therefore, applying traditional software testing frameworks to homework grading is limited. Relying on reference implementations (what aga calls *golden solutions*) has several benefits:

1. Reliability: having a reference solution gives a second layer of confirmation for the correctness of expected outputs. Aga supports *golden tests*, which function as traditional unit tests of the golden solution.
2. Test case generation: many complex test cases can easily be generated via the reference solution, instead of needing to work out the expected output by hand. Aga supports generating test cases from inputs without explcitly referring to an expected output, and supports collecting test cases from python generators.
3. Property testing: unit testing libraries like hypothesis allow testing large sets of arbitrary inputs for certain properties, and identifying simple inputs which reproduce violations of those properties. This is traditionally unreliable, because identifying specific properties to test is difficult. In homework grading, the property can simply be "the input matches the golden solution's output." Support for hypothesis is a long-term goal of aga.

11.2 Installation

Install from pip:

```
pip install aga
```

or with the python dependency manager of your choice (I like poetry), for example:

```
curl -sSL https://install.python-poetry.org | python3 -
echo "cd into aga repo"
cd aga
poetry install && poetry shell
```

11.3 Example

In `square.py` (or any python file), write:

```
from aga import problem, test_case, test_cases

@test_cases(-3, 100)
@test_case(2, aga_expect=4)
@test_case(-2, aga_expect=4)
@problem()
def square(x: int) -> int:
    """Square x."""
    return x * x
```

Then run `aga gen square.py` from the directory with `square.py`. This will generate a ZIP file suitable for upload to Gradescope.

11.4 Usage

Aga relies on the notion of a *golden solution* to a given problem which is known to be correct. The main work of the library is to compare the output of this golden solution on some family of test inputs against the output of a student submission. To that end, aga integrates with frontends: existing classroom software which allow submission of student code. Currently, only Gradescope is supported.

To use aga:

1. Write a golden solution to some programming problem.
2. Decorate this solution with the `problem` decorator.
3. Decorate this problem with any number of `test_case` decorators, which take arbitrary positional or keyword arguments and pass them verbatim to the golden and submitted functions.
4. Generate the autograder using the CLI: `aga gen <file_name>`.

The `test_case` decorator may optionally take a special keyword argument called `aga_expect`. This allows easy testing of the golden solution: aga will not successfully produce an autograder unless the golden solution's output matches the `aga_expect`. You should use these as sanity checks to ensure your golden solution is implemented correctly.

For more info, see the tutorial.

For complete documentation, including configuring problem and test case metadata, see the API reference.

For CLI documentation, run `aga --help`, or access the docs online.

11.5 Contributing

Bug reports, feature requests, and pull requests are all welcome. For details on our test suite, development environment, and more, see the developer documentation.

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

M

N

O

P

R

S

T

U

V

W

Z