CIS 1100

Functional Programming
in Python (Cont.)
& Recursion Start

Python
Fall 2024
University of Pennsylvania

Reminders: HW06

We were a bit unclear about HW06 being due
(with no further late tokens usable) last Friday.

1. Anyone can use HW6 as their dropped assignment independent of the score that you receive on it. Of course, you should still try to complete it if you have time. It will help you understand things and give you more practice on testable topics. To that end...
2. We will reopen HW6 submissions on 11/18 and 11/19 only. This is right before your exam, and so anyone who got all or most of the way through HW6 already should absolutely use the time to just study for the exam instead of submitting things.

Reminders: HW07

HW07 is still due on Wednesday with normal late token policy

HW07 is still due on Wednesday with normal late token policy

HW07 is still due on Wednesday with normal late token policy

HW07 is still due on Wednesday with normal late token policy

HW07 is still due on Wednesday with normal late token policy

Reminder: Recitations

In observance of Election Day, recitation is canceled on Tuesday, 11/5.
Monday recitation (11/4) will be held as normal, but attendance is optional.

If you can't make your recitation feel free for this week to attend another open Monday recitation (211, 212; locations on the website). Additionally, we will host an optional recitation on Tuesday night from 8-9:30 p.m. in Berger Auditorium (in Skirkanich).

Review:

We covered three general purpose higher-order functions:

- `filter`
- `map`
- `reduce`

and a new language feature: `lambda`

Why?

Why are we talking about Higher Order Functions (HOF)?

It turns out that a LOT of problems we want to solve in computer science can reduce down to one of the three funnctions we have shown

- `filter`
- `map` (sometimes called `transform`)
- `reduce` (sometimes called `fold`, `accumulate`, `aggregate` or other terms)

These are sort of "fundamental" patterns in computer science, showing up in many programming languages.

If you want to take more CIS courses (e.g. CIS 1200) then this is a core topic.

Applying Higher Order Functions to past problems

Remember the `caesar` homework?

We wrote a function called `string_to_symbol_list()` that took a string and returned a list but all characters were converted to symbols. `ord(charater) - 65`

What does this sound like?

Try implementing it with a higher order function (L11):

```
def string_to_symbol_list(string):
    # TODO
```

Applying HOF to Caesar

```python
def shift(symbol, n):
    return (symbol + n) % 26 if 0 <= symbol <= 25 else symbol

# assume `n` is the amount we want to shift each symbol by
def encrypt(to_encode, n):
    symbols = string_to_symbol_list(to_encode)
    # TODO: put something here
    return symbol_list_to_string(symbols)
```

Which of these would work for encrypt? (M6)

- (A) `symbols = list(map(shift, symbols))`
- (B) `symbols = list(map(lambda char : shift(char, n), symbols))`
- (C) `symbols = list(map(shift(n), symbols))`
- (D) `symbols = list(reduce(shift, symbols, []))`

(Bonus: When we implement `symbol_list_to_string`, which HOF will we need?)

Practice Code writing!

Given a list of characters, take only the ones which are singular lowercase letters and mash them up into character*3 in a string. e.g. [a,A,B,c,b,travis] becomes "aaacccbbb".

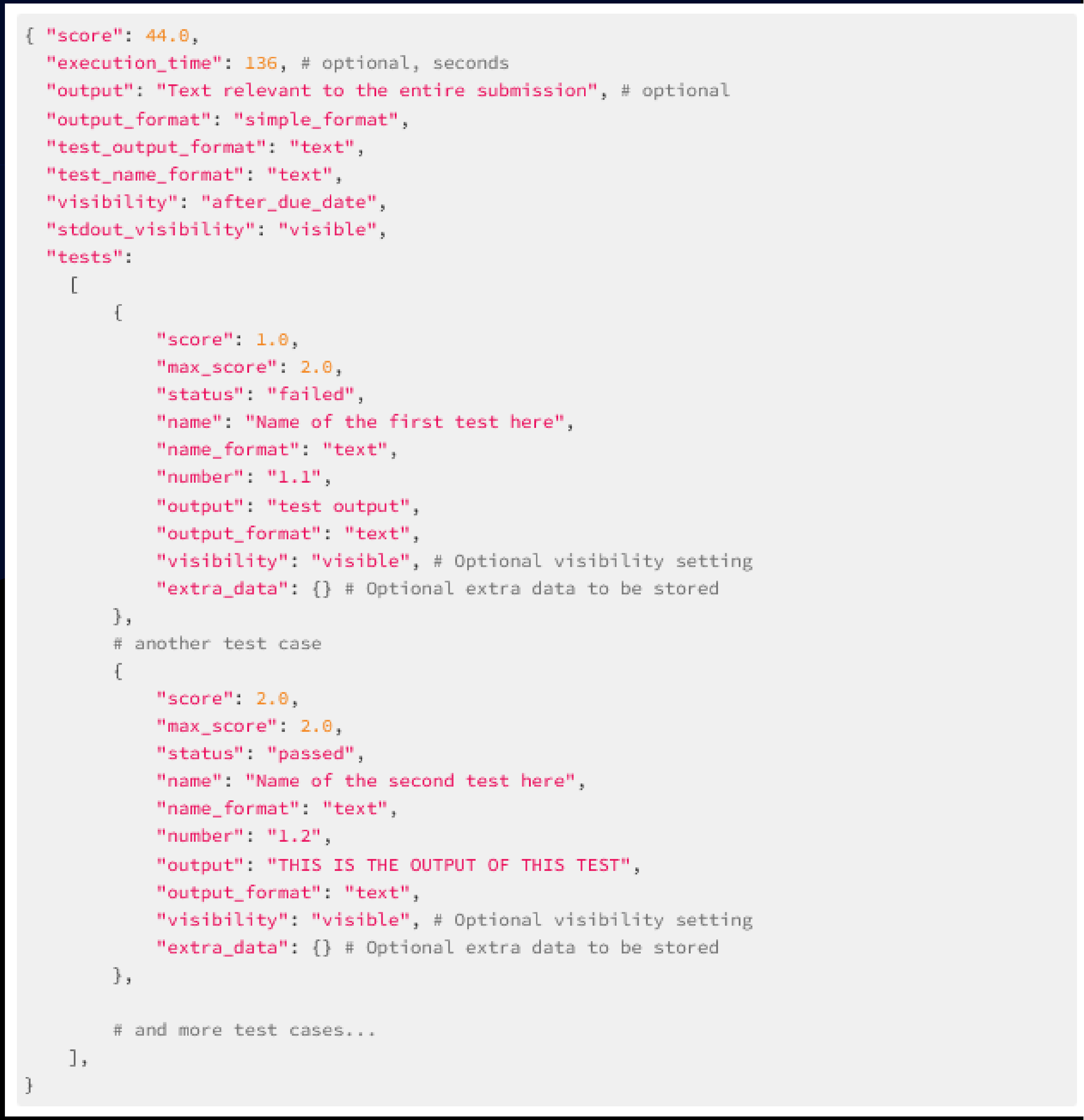
(C12)

Applied to more complex structures

Remember the structure we used in the check-in?

We asked you to implement the `def get_test_names(autograder_results):` which got a list of all tests in that structure.

Which higher order functions would apply here?



```
{ "score": 44.0,
  "execution_time": 136, # optional, seconds
  "output": "Text relevant to the entire submission", # optional
  "output_format": "simple_format",
  "test_output_format": "text",
  "test_name_format": "text",
  "visibility": "after_due_date",
  "stdout_visibility": "visible",
  "tests":
    [
        {
            "score": 1.0,
            "max_score": 2.0,
            "status": "failed",
            "name": "Name of the first test here",
            "name_format": "text",
            "number": "1.1",
            "output": "test output",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },
        # another test case
        {
            "score": 2.0,
            "max_score": 2.0,
            "status": "passed",
            "name": "Name of the second test here",
            "name_format": "text",
            "number": "1.2",
            "output": "THIS IS THE OUTPUT OF THIS TEST",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },

        # and more test cases...
    ],
}
```

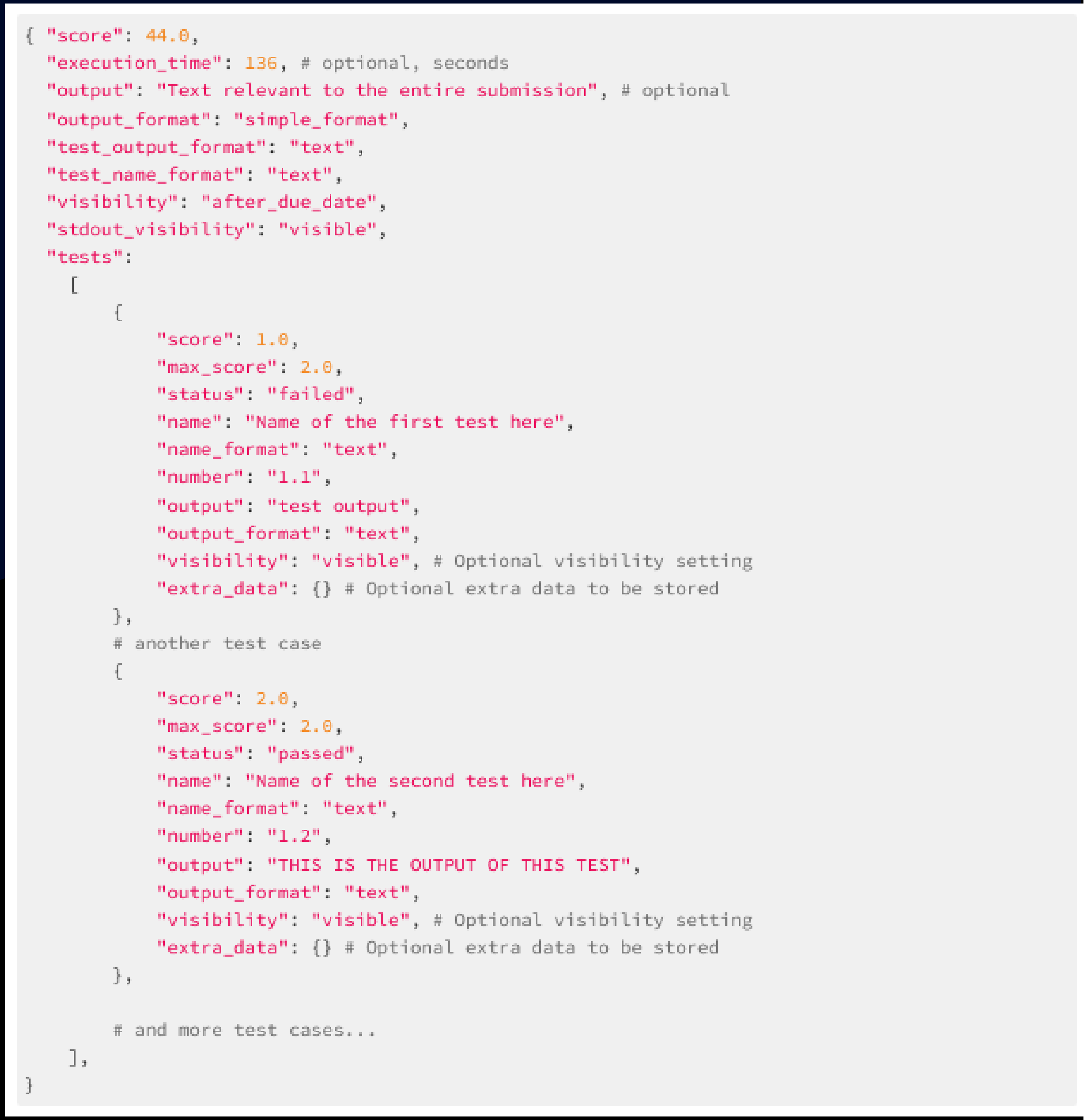
Applied to more complex structures

Remember the structure we used in the check-in?

We asked you to implement the `def get_test_names(autograder_results):` which got a list of all tests in that structure.

If we used HOF we could do:

```
def get_test_names(autograder_results):
    return list(map(lambda test : test['name'], autograder_results['tests']))
```



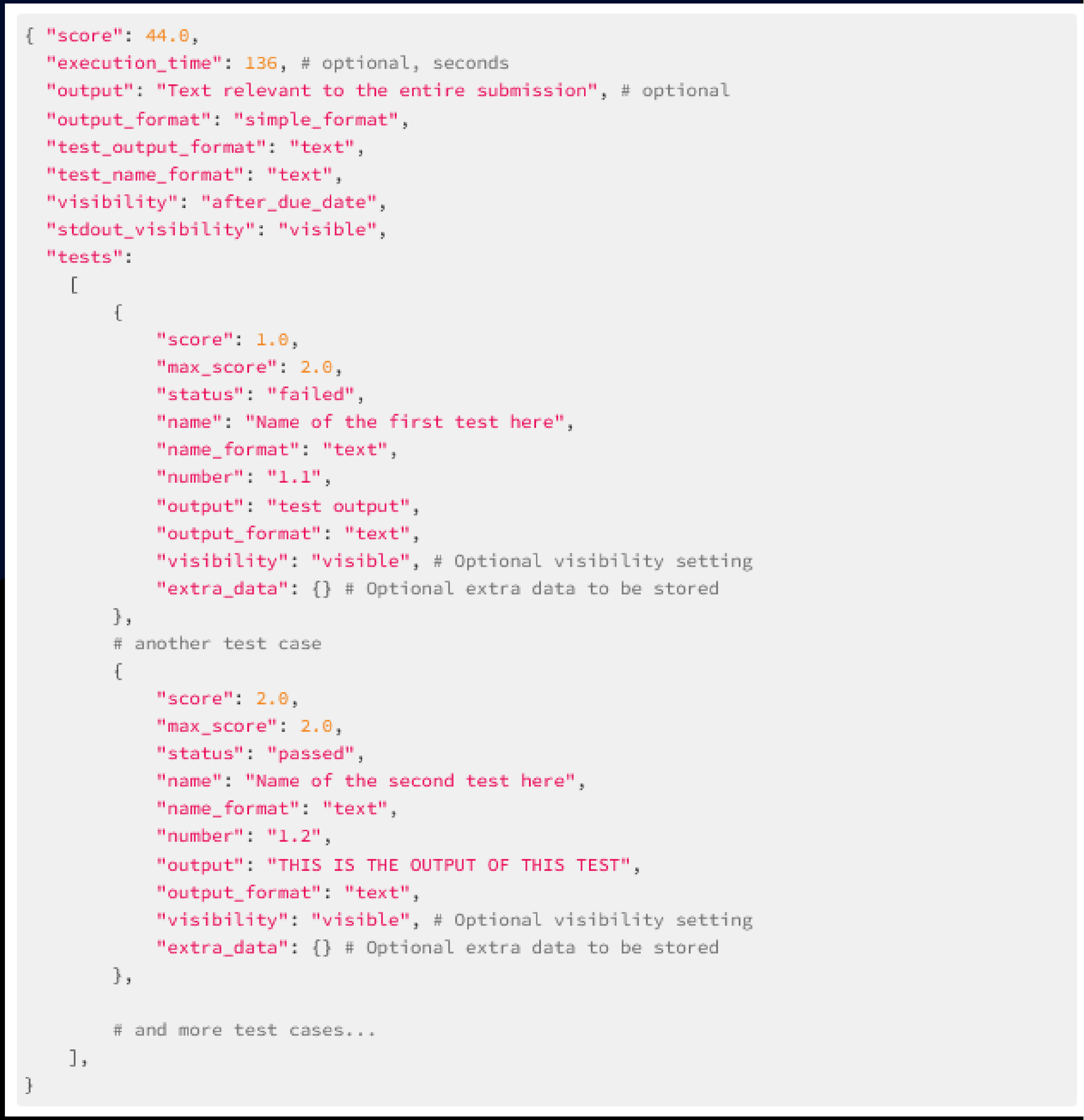
```
{ "score": 44.0,
  "execution_time": 136, # optional, seconds
  "output": "Text relevant to the entire submission", # optional
  "output_format": "simple_format",
  "test_output_format": "text",
  "test_name_format": "text",
  "visibility": "after_due_date",
  "stdout_visibility": "visible",
  "tests":
    [
        {
            "score": 1.0,
            "max_score": 2.0,
            "status": "failed",
            "name": "Name of the first test here",
            "name_format": "text",
            "number": "1.1",
            "output": "test output",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },
        # another test case
        {
            "score": 2.0,
            "max_score": 2.0,
            "status": "passed",
            "name": "Name of the second test here",
            "name_format": "text",
            "number": "1.2",
            "output": "THIS IS THE OUTPUT OF THIS TEST",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },

        # and more test cases...
    ],
}
```

Applied to more complex structures

Remember the structure we used in the check-in?

We asked you to implement the `def get_failing_test_names(autograder_results):` which got a list of all failing tests in that structure.
(e.g. the status was "failed")

How would we do it with HOF?
(C14)



```
{ "score": 44.0,
  "execution_time": 136, # optional, seconds
  "output": "Text relevant to the entire submission", # optional
  "output_format": "simple_format",
  "test_output_format": "text",
  "test_name_format": "text",
  "visibility": "after_due_date",
  "stdout_visibility": "visible",
  "tests":
    [
        {
            "score": 1.0,
            "max_score": 2.0,
            "status": "failed",
            "name": "Name of the first test here",
            "name_format": "text",
            "number": "1.1",
            "output": "test output",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },
        # another test case
        {
            "score": 2.0,
            "max_score": 2.0,
            "status": "passed",
            "name": "Name of the second test here",
            "name_format": "text",
            "number": "1.2",
            "output": "THIS IS THE OUTPUT OF THIS TEST",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },

        # and more test cases...
    ],
}
```

CIS 1100

Recursion Start

Python

Fall 2024

University of Pennsylvania

Recursive Thinking

The journey of a thousand miles starts with one mile.

And then a journey of 999 miles.

Recursive Thinking

A function is recursive if it invokes itself to do part of its work.

Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means.

Recursion reduces a problem into one or more simpler versions of itself.

Recursion

An alternate to using loops for solving problems

The core of recursion is taking a big task and breaking it up into a series of related small tasks.

- Example: handing out papers for an exam
  - Iterative: have a TA walk down a row of students, giving each person an exam
  - Recursive: A student takes one exam, pass the rest down the aisle
- Example: Which row are you in?

Breaking up a large problem

We want to write a program that prints N stars on one line, but without loops.

```
def print_stars(N):
```

Here's

```
print_stars(N) ---> print_stars(1) + print_stars(N - 1)
print_stars(3) ---> print_stars(1) + print_stars(2)
print_stars(2) ---> print_stars(1) + print_stars(1)
print_stars(1) ---> print("*");
```

Anatomy of a Recursive Function

Every recursive function needs at least one **base case** and at least one **recursive part**.

The **base case:**

- handles a simple input that can be solved without resorting to a recursive call. Can also be thought of as the case where we "end“ our recursion.

The **recursive part:**

- contains one or more recursive calls to the function.
- In every recursive call, the parameters must be in some sense "closer" to the base case than those of the original call

Writing our print_stars function:

```
def print_stars(N):
    if (N == 0):        # Base case
        # do nothing
        return
    else:               # Recursive case
        print("*")
        print_stars(N - 1)
```

Practice: Tracing

What would calling `mystrery(5)` do?

(C16)

```python
def mystery(N):
    if (N == 0):
        print("|/")  # prints |/
    else:
        space = " " * N # should be a string with N spaces
        print(f"|{space}/")
        mystery(N - 1)
```

(L13) what would be an appropriate name for the function and it's parameters?

Practice: Tracing

What would this version of mystery do?

(L15)

```python
def mystery(N):
    if (N == 0):
        print("|\\")  # prints |\
    else:
        space = " " * N # should be a string with N spaces
        mystery(N - 1)
        print(f"|{space}\\")  # print AFTER recursive call
        # Previously this was before the recursive call
```