Detecting Silent JSON Changes in Dynamic Programming Languages

Gautam Gadipudi
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
gg7148@rit.edu

***Abstract*—In this capstone project, we implement a static class in Python3 with static methods to capture details about the operations - frame metadata - performed on JSON data, log this frame metadata, and match it against target frame metadata to discover examples and scenarios of silent JSON errors in Python3 programs.**

***Index Terms*—JSON; Silent errors; Python3**

I. Introduction

JSON (JavaScript Object Notation) is an open standard data interchange format. JSON has a particular structure that is easy for humans to read and understand the underlying data and format because it is usually stored as text. This particular JSON structure also makes it easy to load data into almost all languages, perform required operations and dump output as a JSON. Hence, JSON is considered as a universal data structure [1].

A JSON file contains one of the JSON datatypes discussed below:

- *object* - an unordered key/value pair where value can be any of these JSON values.
- *array* - an ordered list one of these JSON values.
- *string* - a sequence of zero or more Unicode characters in double quotes. Backslashes can be used to escape characters. [1]
- *number* - a JavaScript floating-point value, not an integer. [2]
- *true* - boolean true
- *false* - boolean false
- *null* - represents intentional absence of any object value. [3]

Python3 provides the built-in package `json` that exposes API to `load`, `dump` and do other operations on JSON data. The method `load` is used to load JSON from a file or stream. The method `loads` is used to load JSON from a string. Both these methods use the mapping in Table I to convert JSON to built-in Python3 datatypes.

Since most JSON data does not correspond to a particular schema, when this data is loaded into a program in some language and processed, chances of errors (fatal or silent) either due to type mismatching, value mismatching or even structure mismatching are concerning. This either breaks the flow of the program at the point of operation or elsewhere later in time, or even worse silently processes the data incorrectly which leads to unexpected behavior by the application / program.

JSON datatype	Python3 datatype
Object	`dict`
Array	`list`
String	`str`
Number	`int` or `float`
true	`bool`
false	`bool`
null	`None`

TABLE I
CORRESPONDING PYTHON3 DATATYPES AFTER JSON.LOAD()

Good JSON data is the data format expected when implementing a program that processes this data.

Silent JSON errors are not exactly errors, but unexpected changes to the JSON data that makes the program to silently process data incorrectly or raise errors elsewhere in the program. One example of such silent JSON error is when the program (assuming a Python3 program) calculates the number of elements in a JSON array using `len()` method. The program expects a JSON array (mapped to `list`). Due to any reason, if a JSON string (mapped to `str`) is passed instead, the operation of `len()` is still valid, but produces unwanted results. See section III for a detailed example explanation.

This paper focuses on understanding and collecting different scenarios for a program processing JSON data in a dynamic language (Python3) that results in silent JSON errors. For these scenarios, the program logs / keeps a track of the type of operations, location of operations in the codebase and the built-in datatype on which these operations are executed (see II-C2). This metadata is collected for a good JSON input and matched using a comparator with other future JSON input to detect if that particular JSON input will or will not cause silent JSON errors for that program.

II. Implementation

A. Overload built-in datatype methods

In order to collect the previous frame metadata or capture the operations performed on these built-in Python3 datatypes,

we derive new corresponding datatypes (prefixed with 'my') that are inherited from the built-in datatypes. So, these derived datatypes will have the same functionalities and traits as the built-in datatypes, but the overloaded methods will make a function call to the static method - `Tracker.track()`, before returning the expected value(s). The functioning of this method - `track()` is explained later in the paper.

B. Control flow of the program

Every program / example will follow the below steps in order to either collect frame metadata or match against target frame metadata.

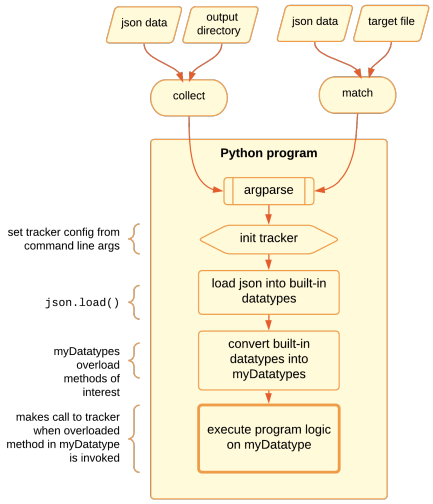

Fig. 1. Control flow of a program

1) Parse command line arguments: A program can be run in either one of the two modes:

- ***collect*** - to collect frame metadata for a particular JSON input data on a program. Requires a JSON data file (`--jsoninputpath` or `-i`) and an output directory (`--outputdirectory` or `-o`) to which the frame metadata file is to be spitted out to.
- ***match*** - to collect frame metadata for a particular JSON input data, and also match it against a target frame metadata from a file. Requires a JSON data file (`--jsoninputpath` or `-i`) to be checked, and a target frame metadata file (`--targetfile` or `-t`) against which the data collected from JSON data file to to be checked.

2) Initialize tracker: The parameters from the command line arguments are used as configuration to initialize the tracker (using `Tracker.init()`). See II-C1 for details.

3) Load JSON data for a program: We use the native approach to work with JSON data in the program i.e. use the `json` module. `json.load()` and `json.loads()` can be used to decode JSON data from a file or a Python3 string respectively. Both these methods use the mapping in Table I to decode JSON data into built-in Python3 datatypes.

4) Convert data in built-in Python3 datatypes to derived datatypes: The data loaded from `json.load()` or `json.loads()` can only be one of `dict` or `list`. But, the **values** (not *keys*) in this `dict` or `list` can be any of the Python3 datatypes mentioned in Table I. So, in an iterative fashion, we convert all values into the derived datatypes - eg. `dict` is transformed to `myDict`, `list` to `myList`, `int` to `myInt`, `str` into `myStr` and so on. This conversion is done by passing the value (built-in datatype) to the corresponding derived datatype's constructor, that returns the same value, but of the type - `myDatatype`.

5) Execute program logic on derived datatypes: We now execute the program logic, not on the built-in datatypes, but on the derived datatypes. `Tracker.track()` will be trigerred in the overloaded methods to collect frame metadata or match against target frame metadata.

C. Control flow of the tracker

Whenever `Tracker.track()` is triggered from an overloaded method, we first capture the previous frame metadata and proceed forward depending on the mode of the tracker as discussed below.

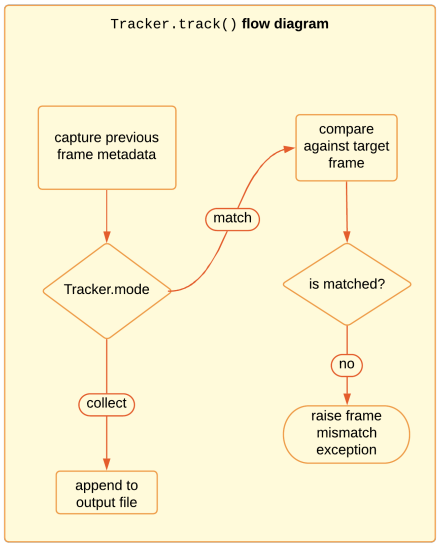

Fig. 2. Control flow of the tracker

1) Initialize tracker: This is done just once - before executing the program logic. Refer Fig. 2. We use the configuration from the command line parameters to initialize the tracker. We also assign a timestamp which is a unique id for the tracker and to distinguish the output files when the same program is run in collect mode multiple times.

If the tracker mode is to be set to 'collect', we create a nested directory of the output directory if it does not exists. The output files (frame metadata) is dumped in this directory.

If the tracker mode is to be set to 'match', we load the frame metadata from the target file into runtime variables to be used when the tracker is triggered.

2) Capture previous frame metadata: The built-in Python3 module - `inspect` is used to capture the frame details. The metadata comprises of the the following fields:

- ***datatype*** - name of the derived datatype
- ***function*** - name of the overloaded method
- ***frame_id*** - id of the frame given by the tracker
- ***previous_frame*** - details about previous frame that triggered the tracker:
  - *line_no* - line number
  - *function* - name of function in the program
  - *file_name* - name of the file / program
  - *module_name* - name of the module
  - *code_context* - code statements as a list of string that triggered the tracker

*3) Tracker mode - **collect**:* Encode the frame metadata mentioned above into json and append to the output file. The output file is collection of JSON objects stored as a *.jsonl* (JSON lines) file.

*4) Tracker mode - **match**:* Get the target frame from the target frame metadata (which was loaded into runtime when the tracker was initialized) using the current `frame_id` of the tracker. We then compare the current frame metadata, with this target frame. If they are not the same, raise a `FrameMismatchException` - meaning that this input data can cause a silent JSON error to the program due to a mismatched field in the frame metadata which is logged onto the console. If they are the same, we continue the execution of the program until there is a frame mismatch or the program has reached the end - meaning that there are no silent JSON errors possible for that data on the particular program.

III. Example Walk-through

Consider a Python3 program given in Fig. 3 of the name *program1.py*. The program is intended to get the number of elements in the array `some_list`.

```python
import sys
from util.setup import init_program


def main(input):
    list_count = len(input['some_list'])
    print(list_count)


def init(config):
    myUser = init_program(config)
    main(myUser)


if __name__ == "__main__":
    init(sys.argv[1:])
```

Fig. 3. A Python3 program - *program1.py*

The program logic lies in the `main()` method which takes the JSON data decoded into derived datatypes as a parameter.

The `init()` method takes the command line arguments as a parameter to initialize the tracker, load JSON data from a file into built-in Python3 datatypes and then convert these datatypes into the derived ones (that have methods overloaded with call to the tracker).

A. Case: collect frame metadata from good data

Consider a JSON data file in Fig. 4 of the name *good.json*. This JSON data is known to be good - meaning it does not cause neither any regular errors and exceptions nor silent JSON errors.

```json
{
    "some_list": [
        "list item 1",
        "list item 2",
        "list item 3",
        "list item 4",
        "list item 5"
    ]
}
```

Fig. 4. Good JSON data input - *good.json*

We collect frame metadata into an output directory using the *collect* command.

The output or collected frame metadata is shown in Fig. 5. See II-C2 for understanding the fields captured.

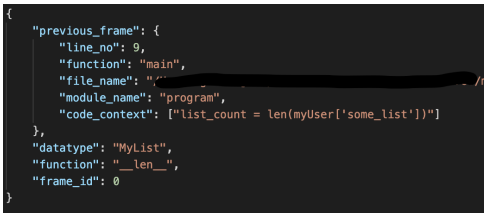

Fig. 5. Frame metadata collected from input *good.json* - *good-20211122T152114EST.jsonl*

B. Case: match frame metadata for bad data

Consider a JSON data file in Fig. 6 of the name *bad.json*. We intend to run the program logic using this data, but we also want to check if this data raises any silent JSON errors.

```json
{
    "some_list": "some string"
}
```

Fig. 6. Bad JSON data input - *bad.json*

We use the *match* command to collect frame metadata from this JSON data and match it against target frame metadata - output from Fig. 5.

The *match* command raises a `FrameMismatchException` with the following message logged onto the console as shown in Fig. 7 below.

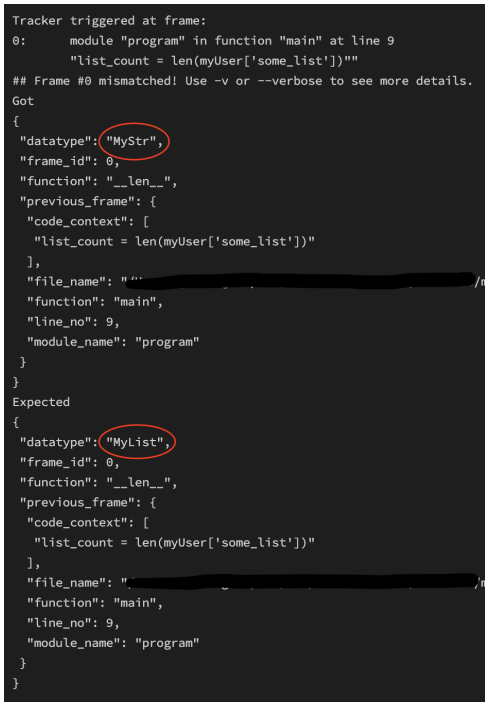

Fig. 7. Console output on running the *match* command using *bad.json* as JSON data input and *good-20211122T152114EST.jsonl* as the target file

Without the tracker, this particular JSON data - *bad.json* would silently execute program logic without any errors. This is a big concern in a bigger codebase where this would result in incorrect data processing or raise errors elsewhere in the codebase.

If we were to *match* against a good data, it would execute program logic as if there was no tracker.

IV. Source Code

Code, tests, examples and scenarios can be tried by following the README at the remote repository: *https://github.com/GautamGadipudi/tracky*.

V. Results

Table II shows valid Python3 operations on JSON data when loaded using `json.load()` or `json.loads()`. Similarly, Table III shows valid Python3 binary operations (requires two operands of same datatype) on JSON data.

In both the tables, the ✓'s along a row (operation) means that they are susceptible to silent JSON errors when there is a datatype mismatch. The ✗ means that the operation is not possible, and will raise a runtime exception at the point of evaluation of that operation.

All the silent JSON errors captured at this time are due to datatype mismatch i.e. target frame metadata was expecting a different datatype.

	Number	String	Array	Boolean	Object
+	✓	✗	✗	✓	✗
−	✓	✗	✗	✓	✗
not	✓	✓	✓	✓	✓
len()	✗	✓	✓	✗	✓
iter() or in	✗	✓	✓	✗	✓
clear()	✗	✗	✓	✗	✓

TABLE II
VALID UNARY OPERATIONS ON JSON DATA IN PYTHON3

	Number	String	Array	Boolean	Object
+	✓	✓	✓	✓	✓
−	✓	✗	✗	✓	✗
*	✓	✗	✗	✓	✗
/	✓	✗	✗	✓	✗
%	✓	✗	✗	✓	✗
== (is)	✓	✓	✓	✓	✓
== (is)	✓	✓	✓	✓	✓
!= (is not)	✓	✓	✓	✓	✓
>	✓	✓	✓	✓	✗
<	✓	✓	✓	✓	✗
<=	✓	✓	✓	✓	✗
>=	✓	✓	✓	✓	✗

TABLE III
VALID BINARY OPERATIONS ON JSON DATA IN PYTHON3

Using Table II, we collected 16 silent JSON errors that were all as a result of datatype mismatch. They are shown in Table IV.

Overloaded method	Expected	Got
len()	myList	myStr
	myList	myDict
	myDict	myList
	myDict	myStr
	myStr	myList
	myStr	myDict
iter()	myList	myStr
	myList	myDict
	myDict	myList
	myDict	myStr
	myStr	myList
	myStr	myDict
clear()	myList	myDict
	myDict	myList

TABLE IV
FRAMEMISMATCHERROR - SILENT JSON ERRORS - DUE TO DATATYPE MISMATCH

VI. Future Work

The implementation in section II falls short of correctly detecting silent JSON errors when the overloaded methods are called from the scope of a loop (`for`) or a condition (`if`, `else` and `elif`). Since, the flow of the execution of a program involving loops and conditions is dynamic, we cannot guarantee that frame metadata collected for one good input can be used as target frame metadata to match / detect silent JSON errors.

One solution or work around to detect silent JSON errors from a program involving dynamic loops is to log the frame

metadata only once (first iteration) per loop. This solution still has an edge case - when the loop is iterated zero times - that needs to be handled.

VII. Conclusion

In this project, we implement a way of collecting details (frame metadata) of operations performed on JSON data in Python3. We use this frame metadata to match against other data on the same Python3 program, to detect any silent JSON errors. We then use this implementation to discover scenarios and examples that result in silent JSON errors. Table IV accounts for only a fraction of all the silent JSON errors possible in the real world. But, our implementation of the tracker can be used to discover other silent JSON changes in almost any Python3 program by importing the `Tracker` into the program and initializing it appropriately like in Fig. 3.

Acknowledgment

The author would like to thank his advisor - Dr. Michael Mior - for the opportunity to work on this project and providing help and ideas in implementing this project. The author would also like to thank Dr. Matthew Fluet for his guidance on reading / writing technical papers and smartly organizing milestones and presentations with prompt feedback.

References

[1] "Introducing json." [Online]. Available: https://www.json.org/json-en.html
[2] "Number - javascript: Mdn." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global$_Objects/Number$
[3] "Null - javascript: Mdn." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global$_Objects/null$