Coding Pitfalls: Demystifying the Potential API Compatibility Risk of Variadic Parameters in Python

Shuai Zhang, Gangqiang He, Guanping Xiao*
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
{shuaizhang, gangqiang.he, gpxiao}@nuaa.edu.cn

***Abstract*—In Python development, developers often use variadic parameters, i.e., `*args` and `**kwargs`, to ensure backward compatibility of APIs after parameter changes or enhancements. Variadic parameters enable APIs to accept an arbitrary number of arguments. However, our preliminary investigation reveals that when an API with variadic parameters internally passes these parameters to other APIs that do not support variadic parameters, it may introduce potential compatibility issues, i.e., variadic parameter pitfalls (VPPs). In this work, we explore the prevalence and impact of VPPs by conducting an empirical study on 33 popular Python third-party libraries using a prototype tool, i.e., VPPDetector. We provide recommendations to mitigate VPPs based on our findings.**

***Index Terms*—Python, variadic parameter pitfalls, API, compatibility issues**

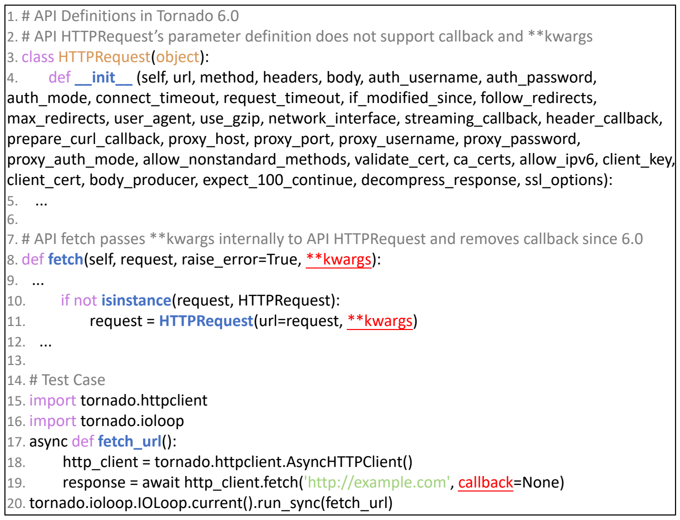
I. Introduction

Developers of Python third-party libraries often need to ensure backward compatibility when changing and maintaining APIs to ensure that existing client code continues to function correctly [1]–[4]. To achieve this goal, variadic parameters, i.e., `*args` and `**kwargs`, are frequently used in APIs, as they provide a flexible way to handle APIs with an arbitrary number of parameters.

The `*args` syntax is used to pass a variable number of positional arguments, which are stored in a tuple within an API. Similarly, `**kwargs` allows for passing a variable number of keyword arguments, preserved by a dictionary within an API. Variadic parameters provide flexible parameter-passing methods, allowing for the renaming/removal of parameters to an API without breaking existing API calls, thereby promoting stronger backward compatibility.

However, improper use of variadic parameters can inadvertently cause compatibility issues. For the example shown in Figure 1, the `callback` parameter of the `fetch` API is removed in Tornado version 6.0. If the `callback` parameter continues to be passed as defined in Tornado version 5.1.1, it will be automatically categorized into `kwargs` upon upgrading to the new version. Subsequently, within the `fetch` API, `**kwargs` is passed to another API, i.e., `HTTPRequest` [5]. Since the parameter definition of `HTTPRequest` does not include `callback`, this results in an exception, i.e., *TypeError: __init__() got an unexpected keyword argument 'callback'*.

We refer to this coding style as **variadic parameter pitfalls (VPPs)**, i.e., an API with variadic parameters internally



```
# API Definitions in Tornado 6.0
# API HTTPRequest's parameter definition does not support callback and **kwargs
class HTTPRequest(object):
    def __init__ (self, url, method, headers, body, auth_username, auth_password,
auth_mode, connect_timeout, request_timeout, if_modified_since, follow_redirects,
max_redirects, user_agent, use_gzip, network_interface, streaming_callback, header_callback,
prepare_curl_callback, proxy_host, proxy_port, proxy_username, proxy_password,
proxy_auth_mode, allow_nonstandard_methods, validate_cert, ca_certs, allow_ipv6, client_key,
client_cert, body_producer, expect_100_continue, decompress_response, ssl_options):
    ...

# API fetch passes **kwargs internally to API HTTPRequest and removes callback since 6.0
def fetch(self, request, raise_error=True, **kwargs):
  ...
    if not isinstance(request, HTTPRequest):
        request = HTTPRequest(url=request, **kwargs)
  ...

# Test Case
import tornado.httpclient
import tornado.ioloop
async def fetch_url():
    http_client = tornado.httpclient.AsyncHTTPClient()
    response = await http_client.fetch('http://example.com', callback=None)
tornado.ioloop.IOLoop.current().run_sync(fetch_url)
```

Fig. 1. A real-world example of variadic parameter pitfalls.

passes these parameters to other APIs that do not support variadic parameters. In this paper, we develop a prototype tool called VPPDetector[1] to detect VPPs in 33 popular Python third-party libraries. We also investigate the prevalence and impact of VPPs on API compatibility by examining real-world cases. Our study provides relevant coding recommendations to mitigate VPPs in Python development. To the best of our knowledge, we perform the first study to explore variadic parameters in Python.

II. Our VPPDetector Approach

Figure 2 shows the overview of VPPDetector. VPPDetector first utilizes Python's built-in AST (abstract syntax tree) module to transform the source code of Python projects into an AST. VPPDetector then traverses all `ast.FunctionDef` type nodes, which correspond to the statements of API definitions in the source code, to extract all defined APIs. Next, VPPDetector detects VPPs based on the following three criteria:

- **Rule 1.** The API's parameter definition includes variadic parameters, i.e., `*args` and/or `**kwargs`.
- **Rule 2.** Within the implementation of an API filtered by rule 1, there exists a case where variadic parameters are passed to other APIs.

*Guanping Xiao is the corresponding author.

[1]https://github.com/raise-group/VPPDetector

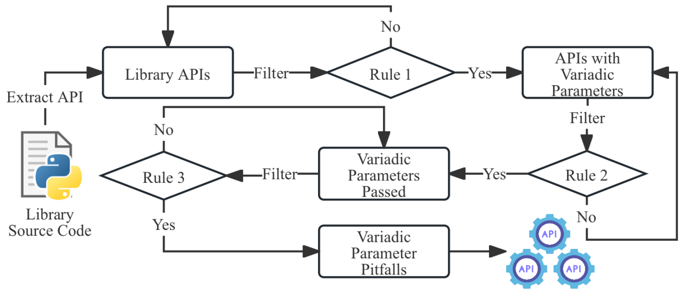


Fig. 2. Detection of variadic parameter pitfalls.

- **Rule 3.** The called API, which receives variadic parameters (as per rule 2), does not define variadic parameters in its parameter definition.

Currently, VPPDETECTOR extracts the parameter definition of the called API from the same source file as the caller API by searching the API's name. The final detection results are saved into a JSON file, which records the file and line number where each API is located in detail.

III. EVALUATION

A. Dataset

We adopted 33 popular Python libraries selected from PCART [4]. The selection criteria include GitHub stars, PyPI download counts, and the API documentation detail [4]. In the dataset, each library contains all versions up to July 2023.

B. Results and Recommendations

Through the detection on the 33 libraries by VPPDETECTOR, 26 libraries have VPPs. Table I lists the top 10 libraries by the proportion of VPPs. The table records the following metrics: the average number of APIs with variadic parameters defined across all versions per library, the average number of these APIs that pass variadic parameters to other APIs, the average number of these passed variadic parameters that are not included in the receiving API's parameter definition (i.e., the average number of VPPs in each version), and the average proportion of VPPs, i.e., $\frac{1}{n}\sum_{i=1}^{n}\frac{VariadicParamPitfalls_i}{VariadicParamsPassed_i}$, where n is the total number of versions and i is the ith version. Table I shows that TensorFlow has the highest average number (886) of APIs with variadic parameters defined across all versions. Despite this, Pillow has the highest proportion of VPPs, reaching 35.74%.

Although VPPs do not necessarily indicate true incompatibility, they pose a potential risk to API compatibility. Our further manual analysis reveals that VPPs could cause compatibility issues when APIs with variadic parameters undergo parameter renaming or removal. For instance, in Matplotlib version 3.4.3, calling `Shadow(patch, ox, oy, props=None)` is feasible. However, when upgrading to Matplotlib version 3.5.0, since the `props` parameter is removed in the new version [6], continuing to pass this parameter will also be classified into variadic parameters (i.e., `**kwargs`). When these variadic parameters are subsequently passed to the `update` API [7], it will throw an exception, i.e., *AttributeError: 'Shadow' object has no property 'props'*.

TABLE I
DISTRIBUTION OF VARIADIC PARAMETER USAGE IN THE TOP 10 LIBRARIES BY VARIADIC PARAMETER PITFALL RATE.

Library	Version Num	Avg. APIs with Variadic Params	Avg. Variadic Params Passed	Avg. Variadic Param Pitfalls	Avg. Variadic Param Pitfall Rate
Pillow	75	62	25	9	35.74%
Requests	107	43	39	7	18.12%
aiohttp	216	63	49	8	14.37%
Faker	292	21	17	2	10.23%
Gensim	48	55	46	3	8.58%
Click	53	30	27	2	6.73%
scikit-learn	42	178	119	7	6.44%
XGBoost	31	25	18	1	6.43%
SciPy	66	388	310	19	6.22%
TensorFlow	76	886	671	37	5.76%

To mitigate VPPs, our recommendations are as follows:

- **Recommendation 1.** When passing variadic parameters to an API, developers should ensure that the called API's parameter definition includes variadic parameters.
- **Recommendation 2.** Parse the variadic parameters first to identify those that will be used, and then pass only those parameters to an API, rather than simply passing variadic parameters.

IV. CONCLUSIONS AND FUTURE WORK

In this paper, we conducted a preliminary study to explore variadic parameter pitfalls (VPPs) and their potential risk to API compatibility by a prototype tool (VPPDETECTOR). While variadic parameters are intended to enhance parameter-passing flexibility and backward compatibility, improper usage can introduce subtle yet significant compatibility challenges. Our findings provide better coding recommendations for developers aiming to design robust APIs and offer preliminary insights for researchers interested in API evolution and compatibility issues. In our future work, we will further analyze the change patterns of variadic parameters and VPPs during the evolution of Python library APIs, as well as the intentions behind developers' use of these parameters.

ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China under Grant 62002163 and Natural Science Foundation of Jiangsu Province under Grant BK20200441.

REFERENCES

[1] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do python framework apis evolve? an exploratory study," in *SANER*. IEEE, 2020, pp. 81–92.
[2] J. Wang, G. Xiao, S. Zhang, H. Lei, Y. Liu, and Y. Sui, "Compatibility issues in deep learning systems: Problems and opportunities," in *ESEC/FSE*. ACM, 2023, pp. 476–488.
[3] H. Lei, S. Zhang, J. Wang, G. Xiao, Y. Liu, and Y. Sui, "Why do deep learning projects differ in compatible framework versions? an exploratory study," in *ISSRE*. IEEE, 2023, pp. 509–520.
[4] S. Zhang, G. Xiao, J. Wang, H. Lei, Y. Liu, Y. Sui, and Z. Zheng, "Pcart: Automated repair of python api parameter compatibility issues," *arXiv preprint arXiv:2406.03839*, 2024.
[5] Tornado, "Httprequest," Retrieved July 14, 2024 from https://github.com/tornadoweb/tornado/blob/v6.0.0/tornado/httpclient.py#L337, 2024.
[6] Matplotlib/Shadow, "matplotlib.patches.shadow," Retrieved July 14, 2024 from https://github.com/matplotlib/matplotlib/blob/v3.5.0/lib/matplotlib/patches.py#L637, 2024.
[7] Matplotlib/update, "matplotlib.artist.artist.update," Retrieved July 14, 2024 from https://github.com/matplotlib/matplotlib/blob/v3.5.0/lib/matplotlib/artist.py#L1046, 2024.