Exceptional Behaviors: How Frequently Are They Tested?

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

Gordon Fraser
University of Passau
Passau, Germany
gordon.fraser@uni-passau.de

***Abstract*—Exceptions allow developers to handle error cases expected to occur infrequently. Ideally, good test suites should test both normal and exceptional behaviors to catch more bugs and avoid regressions. While current research analyzes exceptions that propagate to tests, it does not explore other exceptions that do not reach the tests. In this paper, we provide an empirical study to explore how frequently exceptional behaviors are tested in real-world systems. We consider both exceptions that propagate to tests and the ones that do not reach the tests. For this purpose, we run an instrumented version of test suites, monitor their execution, and collect information about the exceptions raised at runtime. We analyze the test suites of 25 Python systems, covering 5,372 executed methods, 17.9M calls, and 1.4M raised exceptions. We find that 21.4% of the executed methods do raise exceptions at runtime. In methods that raise exceptions, on the median, 1 in 10 calls exercise exceptional behaviors. Close to 80% of the methods that raise exceptions do so infrequently, but about 20% raise exceptions more frequently. Finally, we provide implications for researchers and practitioners. We suggest developing novel tools to support exercising exceptional behaviors and refactoring expensive `try/except` blocks. We also call attention to the fact that exception-raising behaviors are not necessarily "abnormal" or rare.**

***Index Terms*—Software Testing, Exceptional Behaviors, Python**

I. Introduction

Exceptions are a programming construct that allows developers to handle error cases expected to occur infrequently, without cluttering the code with unnecessary if/else checks. By using exceptions, developers can improve the program's robustness by enabling the detection, reporting, handling, and correction of exceptional behaviors [1], [2]. Good test suites should ideally test both normal and exceptional behaviors to catch more bugs and avoid regressions [3]–[8]. However, in practice, it is well-known that developers are more likely to test normal behaviors than exceptional ones [8]–[19], which may decay the test suite's effectiveness in finding bugs [2].

Figure 1a shows method `monthrange`,[1] which returns the weekday and number of days for a given *year* and *month*, raising an exception when *month* is not between 1 and 12. This method is called 221 times by its test suite, but only 3 calls exercise the exception. In the case method `monthrange` is directly called by a test, it should be verified with an `assertRaises` to check that the exception `IllegalMonthError` gets raised. Those tests are named "exceptional tests", that is, tests that expect exceptions to be raised [1], [2]. Method `monthrange` is exercised by three exceptional tests.[2]

On the other hand, methods `_has_surrogates`[3] and `_path_is_relative_to`[4] presented in Figures 1b and 1c handle the raised exception with `try/except` blocks. Thus, the raised exceptions do not propagate to the tests. That is,

[1]https://github.com/python/cpython/blob/950fab46/Lib/calendar.py#L161

[2]https://github.com/python/cpython/blob/950fab46/Lib/test/test_calendar.py#L894-L906

[3]https://github.com/python/cpython/blob/950fab46/Lib/email/utils.py#L47

[4]https://github.com/pallets/flask/blob/2fec0b20/src/flask/sansio/scaffold.py#L709

```python
def monthrange(year, month):
    """Return weekday of first day of month (0-6 ~ Mon-Sun)
       and number of days (28-31) for year, month."""
    if not 1 <= month <= 12:
        raise IllegalMonthError(month)
    day1 = weekday(year, month, 1)
    ndays = mdays[month] + (month == FEBRUARY and isleap(year))
    return day1, ndays
```

(a) Method monthrange (CPython). Exception `IllegalMonthError` is rarely raised when executing the test suite.

```python
def _has_surrogates(s):
    """Return True if s may contain surrogate-escaped binary data."""
    # This check is based on the fact that unless there are surrogates, utf8
    # (Python's default encoding) can encode any string.  This is the fastest
    # way to check for surrogates, see bpo-11454 (moved to gh-55663) for timings.
    try:
        s.encode()
        return False
    except UnicodeEncodeError:
        return True
```

(b) Method _has_surrogates (CPython). Exception `UnicodeEncodeError` is rarely raised when executing the test suite.

```python
def _path_is_relative_to(path: pathlib.PurePath, base: str) -> bool:
    # Path.is_relative_to doesn't exist until Python 3.9
    try:
        path.relative_to(base)
        return True
    except ValueError:
        return False
```

(c) Method _path_is_relative_to (Flask). Exception `ValueError` is almost always raised when executing the test suite.

Fig. 1: Examples of methods with exceptional behaviors.

from the test perspective, the exceptions `UnicodeEncodeError` and `ValueError` are unnoticed, and we are not aware whether they get raised. One important difference between both methods is the frequency of exception raising. Method `_has_surrogates` is called 32,846 times by its test suite, but the exception is raised only in 0.6% of the calls. In contrast, method `_path_is_relative_to` raises the exception in 98% of the calls (441 out of 448 calls). These examples show that, at the method level, exceptions may be frequently or infrequently raised at runtime. Moreover, when running a test suite, multiple exceptions may be raised at runtime, including the ones that are unnoticed by the tests because they are handled locally. This means that a test suite may exercise exceptional behaviors even without explicitly asserting on the raised exceptions (*e.g.,* using `assertRaises`).

While prior research focuses on analyzing exceptions that propagate to tests (*i.e.,* exceptional tests [1], [2]), it does not explore exceptions that are *not* propagated to tests. Moreover, to our knowledge, no study has deeply explored the frequency of exception-raising at runtime from the test perspective. A better understanding of these aspects could provide insights into how developers handle exceptional behaviors and help drive the creation of novel testing tools.

In this paper, we provide an empirical study to explore how frequently exceptional behaviors are tested in real-world systems. We consider both exceptions that propagate to tests and the ones that do not reach the tests. For this purpose, we run an instrumented version of test suites, monitor their execution, and collect information about the exceptions raised at runtime. Specifically, we analyze the test suites of 25 Python systems, covering 5,372 executed methods, 17.9M calls, and 1.4M raised exceptions. We propose three research questions to explore exceptional behaviors:

- **RQ1: How many methods raise exceptions at runtime?** 21.4% of the executed methods do raise exceptions at runtime. On the median, methods that raise exceptions are called 4x more often and execute 3x more paths than those that do not raise exceptions.
- **RQ2: How frequently do calls on exception-raising methods actually lead to exceptions?** In methods that raise exceptions at runtime, 1 in 10 calls exercise exceptional behaviors on the median. Close to 80% of the methods that raise exceptions do so infrequently, while about 20% raise exceptions more frequently.
- **RQ3: How do exception-raising methods and calls vary by system?** Most systems (22 in 25) contain more exception-free than exception-raising methods. Moreover, most systems (19 in 25) have a median proportion of exception-raising calls per method below 30%.

Based on our results, we discuss practical implications. First, we envision the development of novel tools to support exercising exceptional behaviors more effectively. For example, such tools could identify the tests that cover exceptional cases, including exceptions not propagated to tests. Second, we reveal that a test that exercises an exception-raising method does not necessarily indicate it is testing an "abnormal" behavior. For some methods, raising an exception may be part of the method's "normal" behavior. Therefore, researchers working on exceptional behavior testing [1], [2], [20]–[22] should be aware of such methods to avoid failing to detect abnormal behaviors. Third, we recommend a refactoring to replace expensive `try/except` blocks. We foresee that future research on refactoring [23]–[28] could leverage the execution frequency of some language constructors (*e.g.,* `try/except` blocks) to detect refactoring opportunities

TABLE I: Selected systems.

System	Short Description	Methods	Tests
Pylint	static code analyzer	1,537	1,822
Rich	rich text library	589	758
Error Corrector	popular console error corrector	550	1,887
BentoML	machine learning platform	319	169
Flask	web framework	284	478
DateUtil	date and time library	241	2,029
Requests	HTTP library	174	578
Jupyter Client	Jupyter protocol client APIs	138	87
Cookiecutter	template handler	66	322
Six	compatibility library	32	199
email	email message manager	381	1,666
logging	logging facility	215	208
argparse	command-line interfaces	126	1,685
collections	datatype container	112	111
pathlib	OO filesystem paths	97	449
tarfile	tar reading and writing	89	496
configparser	configuration file parser	82	341
calendar	calendar helpers	63	72
ftplib	FTP protocol client	51	94
difflib	diff library	47	51
imaplib	IMAP4 protocol client	47	103
smtplib	SMTP protocol client	43	82
os	operating system interfaces	41	316
gzip	gzip reading and writing	32	61
csv	CSV reading and writing	15	113
Total		5,372	14,177

Contributions. The contributions of this study are twofold. First, we propose an empirical study to explore the frequency of exception-raising at runtime from the test perspective. Second, we provide implications for researchers and practitioners.

II. Study Design

A. Case Studies

In this study, we aim to study real-world software systems. Thus, we select Python systems that are largely adopted. We focus on Python because it is among the most popular programming languages nowadays,[5] and it has a rich software ecosystem. Table I presents the 25 selected systems. For a larger diversity of projects, we select two types of systems: (1) popular systems and (2) Python libraries.

Popular systems. The 10 selected systems presented at the top of Table I have thousands of GitHub stars, millions of clients, and, in some cases, billions of downloads, highlighting their relevance.[6] For example, DateUtil is a date library used by

[5]https://www.tiobe.com/tiobe-index

[6]Download values are based on PePy: https://pepy.tech.

close to 1M GitHub projects. Pylint is the most popular code analyzer in Python, with over 400M downloads. Rich is a text library with 40K stars and around 200M downloads. Requests is an HTTP library with 50K stars and 6,4B downloads. Flask is a popular web framework, with 1,3M users on GitHub and around 2B downloads. Cookiecutter is a template handler with 20K stars. Six is a compatibility library used by 1,5M projects with 6,5B downloads. Lastly, Jupyter Client provides Jupyter protocol client APIs, having around 500M downloads.
Python libraries. We also analyze the 15 libraries presented at the bottom of Table I, which belong to the Python Standard Library and are hosted on the CPython repository.

B. Monitoring Methods Executed by Tests

To extract information about exceptions raised at runtime, we need to run an instrumented version of the test suite, monitoring the method execution and collecting their raised exceptions. For this purpose, we rely on SpotFlow [29], a tool that performs dynamic analysis in Python and collects runtime data, such as argument values, return values, and raised exceptions at the method level. In short, this tool is implemented with the support of the standard system's trace function `sys.settrace` [30], which is the basis for performing runtime analysis in Python [31], [32]. The trace function registers a hook that gets called at every executed line of code and function call, allowing the recording of method-level runtime data, such as calls and raised exceptions.

Therefore, with the support of SpotFlow, we run an instrumented version of the 25 selected test suites and detect that 5,372 application methods are executed, as detailed in Table I.

C. Collecting Data from Executed Methods and Calls

While monitoring the tests, we gather data on methods and calls that raise exceptions at runtime.

Exception-raising methods: For each method, we record whether it raises any exception at runtime. Methods that do not raise exceptions are *exception-free methods*, while methods that raise at least one exception are *exception-raising methods*.

Exception-raising calls: For each method call, we record whether it results in an exception being raised, referring to such calls as *exception-raising calls*. For each method, we compute both the absolute and relative number of exception-raising calls. The relative number is the ratio of exception-raising calls to the total number of calls.

In total, the 5,372 executed methods receive 17.9M calls and raise 1.4M exceptions at runtime. On the median, each executed method receives 33 calls. This data is further explored in our research questions. Our dataset is publicly available at: https://doi.org/10.5281/zenodo.14187323.

D. Research Questions

We propose three research questions to explore exception-raising frequency at runtime. In RQ1, we analyze how many methods raise exceptions. In RQ2, we assess how frequently calls lead to exceptions. RQ3 explores the variation of exception-raising methods and calls per system. **Rationale:** Each research question addresses the frequency of the raised exceptions at a distinct granularity level. RQ1 focuses on the method level, while RQ2 on the call level. RQ3 analyzes methods and calls but on the system level. A better understanding of these aspects could provide insights into how developers handle exceptional behaviors, potentially guiding the development of novel testing tools.

TABLE II: Summary of methods raising exceptions at runtime.

Categories	**#**	**%**
Exception-free methods	4,222	78.6%
Exception-raising methods	1,150	21.4%
All	5,372	100%

TABLE III: Distribution of exception types.

	Total	**Methods**			
		1	2–3	4–9	10+
Types of raised exceptions	200	73	62	40	25

III. Results

A. RQ1: How many methods raise exceptions at runtime?

Considering the 25 selected Python systems, 5,372 application methods are directly or indirectly executed by their test suites. Among these methods, 21.4% (1,150) do raise exceptions at runtime, while 78.6% (4,222) do not raise any exceptions, as detailed in Table II.

Table III presents the distribution of the exception types of raised exceptions. In total, we find that 200 distinct exception types are raised at runtime. Most exceptions are raised by a single method (73 in 200). Moreover, 62 exceptions are raised by 2–3 methods, 40 exceptions by 4–9 methods, and only 25 exceptions by 10 or more methods. The most raised exception types are the generic ones: `ValueError`, `GeneratorExit`, `TypeError`, `KeyError`, and `StopIteration`. On the other hand, exception types raised by single methods are very specific, such as `SMTPSenderRefused`, `InvalidJSONError`, `EmptyHeaderError`, `IllegalWeekdayError`, and `NoEmoji`.

Figure 2 presents the distribution of method calls in exception-free and exception-raising methods. On the median, the methods that do not raise exceptions have 24 calls, while those that do raise have 105 calls. The difference is statistically significant (Mann-Whitney test, *p-value* < 0.05). Figure 3 details the distribution of executed paths in both groups. On the median, the methods that do not raise exceptions execute 1 path, while those that do raise execute 3 paths. The difference is statistically significant (*p-value* < 0.05).

Observation 1: 21.4% of the executed methods raise exceptions at runtime. On the median, exception-raising methods receive 4x more calls and execute 3x more paths than exception-free methods

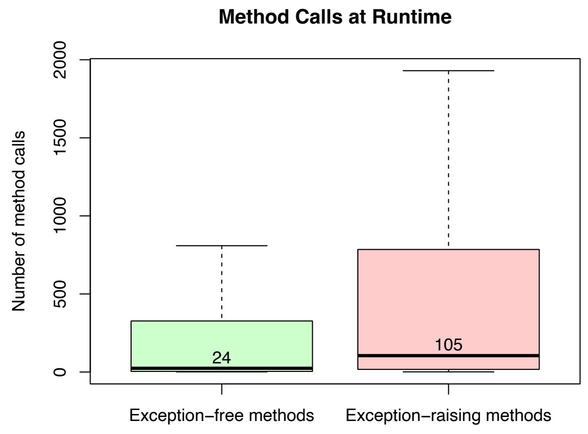

Fig. 2: Distribution of method calls at runtime.

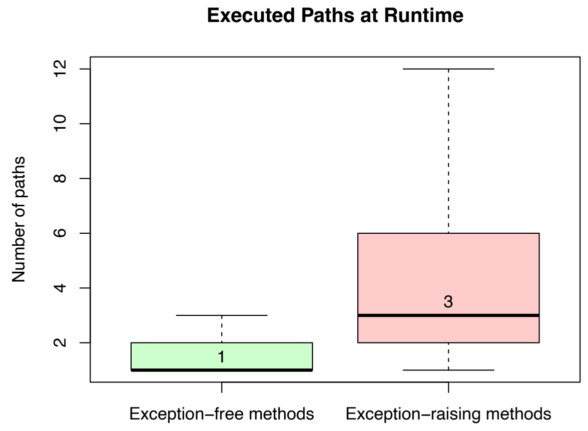

Fig. 3: Distribution of executed paths at runtime.

B. RQ2: How frequently do calls on exception-raising methods actually lead to exceptions?

In this research question, we analyze the 1,150 methods that raise exceptions at runtime. In the previous RQ, we saw that these methods received a median of 105 calls. However, naturally, not necessarily all these calls raise exceptions at runtime. Therefore, here, we focus on the method calls that actually lead to an exception being raised.

Figure 4 presents the distribution of method calls raising exceptions at runtime in both absolute (left-side) and relative (right-side) values. On the median, exception-raising methods receive 4 calls that raise exceptions (the first quartile is 2 calls, and the third quartile is 18 calls). In such methods, on the median, 10% of the calls are exception-raising. This means that 1 in 10 calls cover exceptional behaviors. In this case, the first quartile is 1%, while the third quartile is 48%. The first quartile in 1% states that in 25% of the exception-raising

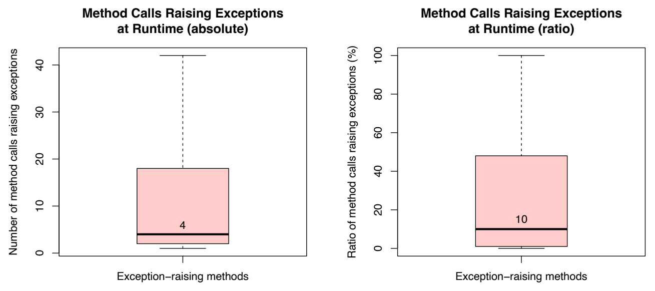

Fig. 4: Distribution of the method calls raising exceptions at runtime (absolute and relative values).

TABLE IV: Frequency of exception-raising calls.

Frequency	**Exception-Raising Calls**	**#Methods**	**%**
Rare	$\leq$ 10%	576	50.0%
Occasional	$>$ 10% and $\leq$ 50%	327	28.4%
Common	$>$ 50% and $<$ 90%	111	9.6%
Almost always	$\geq$ 90%	136	11.8%
All	-	1,150	100%

methods, at most 1 in 100 calls raise an exception.

Observation 2: In methods that raise exceptions, on the median, 1 in 10 calls exercise exceptional behaviors. Each exception-raising method receives a median of 4 calls that cover exceptions.

Table IV groups the exception-raising methods into four categories according to their frequency of exception-raising calls: rare, occasional, common, and almost always. Methods that raise exceptions in at most 10% of their calls are categorized as *rarely* raising exceptions. Most exception-raising methods fall into this category, which includes 50% (576) of the analyzed methods. Methods that raise exceptions in more than 10% and up to 50% of their calls are classified as *occasionally* raising exceptions. We find that 28.4% (327) of the methods raise exceptions occasionally. Methods that raise exceptions in more than 50% but less than 90% of their calls are categorized as *commonly* raising exceptions, while methods that raise exceptions in 90% or more are classified as *almost always* raising exceptions. Only 9.6% (111) of the methods commonly raise exceptions, while 11.8% (136) almost always raise exceptions.

Observation 3: Close to 80% of the methods that raise exceptions at runtime do so infrequently, while only about 20% raise exceptions more frequently.

Table V presents multiple examples of exception-raising methods and their calls in each category. Next, we briefly discuss some interesting examples.

Rarely raising exceptions: This category includes meth-

TABLE V: Examples of methods by frequency of exception-raising calls.

Frequency	Method	System	#Calls	Exception-Raising Calls	
				#	%
Rare	pylint.checkers.base_checker.BaseChecker.create_message_definition_from_tuple	pylint	1,279,848	1	<1%
	email._header_value_parser.get_ttext	email	9,188	1	<1%
	tarfile.TarInfo._decode_pax_field	tarfile	25,092	194	1%
	requests.utils._validate_header_part	requests	1,838	15	1%
	dateutil.parser._parser.parser.parse	dateutil	1,160	36	3%
	rich.live.Live.start	rich	26	1	4%
	rich.console.Console.rule	rich	20	1	5%
	flask.templating.render_template	flask	32	2	6%
	requests.sessions.Session.get	requests	34	3	9%
	cookiecutter.repository.determine_repo_dir	cookiecutter	50	5	10%
Occasional	cookiecutter.generate.apply_overwrites_to_context	cookiecutter	19	2	11%
	flask.scaffold.Scaffold.get	flask	9	1	11%
	pathlib.Path.touch	pathlib	17	2	12%
	flask.app.Flask.url_for	flask	42	7	17%
	collections.namedtuple	collections	55	11	20%
	logging.config._install_handlers	logging	17	4	24%
	requests.utils.is_ipv4_address	requests	52	15	29%
	argparse.ArgumentParser.parse_known_args	argparse	4,432	1,347	30%
	pylint.checkers.classes.class_checker.ClassChecker.visit_functiondef	pylint	3,521	1,247	35%
	pathlib.PurePath.relative_to	pathlib	303	140	46%
Common	pathlib.PurePath.__hash__	pathlib	12,414	6,384	51%
	cookiecutter.zipfile.unzip	cookiecutter	16	9	56%
	pylint.testutils._run._add_rcfile_default_pylintrc	pylint	425	246	58%
	configparser.RawConfigParser._get_conv	configparser	1,274	765	60%
	pylint.checkers.stdlib.StdlibChecker._check_open_call	pylint	135	84	62%
	pathlib.Path.mkdir	pathlib	490	316	64%
	pathlib.PurePath.with_suffix	pathlib	91	63	69%
	cookiecutter.replay.load	cookiecutter	4	3	75%
	pathlib.Path.samefile	pathlib	20	16	80%
	dateutil.parser._parser.parser._parse	dateutil	1,160	1,013	87%
Almost Always	pathlib.Path.rglob	pathlib	219	200	91%
	email._header_value_parser.get_address	email	334	306	92%
	pylint.utils.file_state.FileState.set_msg_status	pylint	3,564	3,384	95%
	requests.utils.get_auth_from_url	requests	273	264	97%
	six.MovedModule.__getattr__	six	41	40	98%
	email.message.Message._get_params_preserve	email	958	945	99%
	pathlib._RecursiveWildcardSelector._iterate_directories	pathlib	6,085	6,025	99%
	argparse._ActionsContainer._handle_conflict_error	argparse	4	4	100%
	email.header._ValueFormatter._maxlengths	email	56	56	100%
	pylint.utils.utils.get_module_and_frameid	pylint	3,618	3,618	100%

ods that raise exceptions in at most 10% of their calls. The rarest exception found in our dataset happens in the Pylint method `create_message_definition_from_tuple`.[7] This method receives over 1,2 million calls, but only one call raises the `InvalidMessageError` due to a malformed message. Method `get_ttext`[8] of the email library in CPython is another interesting example (as presented in Figure 5). This method receives 9,188 calls, but only one call raises the exception `HeaderParseError` due to an empty string passed to the parameter `value`. Overall, this category includes methods in which raising the exception is a rare phenomenon.

Occasionally raising exceptions: This category includes methods that raise exceptions in more than 10% and up to 50% of their calls. As an example, we present the Requests method `is_ipv4_address`[9] (Figure 6). This method receives 52 calls, from which 15 (29%) raise the exception `OSError`, indicating that the parameter `string_ip` is an invalid IPv4. Differently from the previous two examples (in which a single and rare call covered the exception), here, we see multiple exception-raising calls. In this case, the 15 raised exceptions happened due to the following inputs: *"localhost.localdomain"* (7 inputs), *"www.requests.com"* (4 inputs), *"google.com"* (3 inputs), and *"8.8.8.8.8"* (1 input). It is interesting to note that 2 out of the 15 exception-raising calls come directly from test `TestIsIPv4Address.test_invalid`.[10] In contrast, the other 13 exception-raising calls come indirectly from other tests, making them harder to track from the test perspective.

Commonly raising exceptions: This category includes

[7] https://github.com/pylint-dev/pylint/blob/c25923f3/pylint/checkers/base_checker.py#L182

[8] https://github.com/python/cpython/blob/e6dd71da/Lib/email/_header_value_parser.py#L2258

[9] https://github.com/psf/requests/blob/7335bbf4/src/requests/utils.py#L711

[10] https://github.com/psf/requests/blob/7335bbf4/tests/test_utils.py#L261

```python
def get_ttext(value):
    """ttext = <matches _ttext_matcher>

    We allow any non-TOKEN_ENDS in ttext, but add defects to the token's
    defects list if we find non-ttext characters.  We also register defects for
    *any* non-printables even though the RFC doesn't exclude all of them,
    because we follow the spirit of RFC 5322.

    """
    m = _non_token_end_matcher(value)
    if not m:
        raise errors.HeaderParseError(
            "expected ttext but found '{}'".format(value))
    ttext = m.group()
    value = value[len(ttext):]
    ttext = ValueTerminal(ttext, 'ttext')
    _validate_xtext(ttext)
    return ttext, value
```

Fig. 5: Method get_ttext of the email library in CPython. Calls: 9,188; exception-raising calls: 1 (<1%).

```python
def is_ipv4_address(string_ip):
    """
    :rtype: bool
    """
    try:
        socket.inet_aton(string_ip)
    except OSError:
        return False
    return True
```

Fig. 6: Method is_ipv4_address of Requests. Calls: 52; exception-raising calls: 15 (29%).

methods that raise exceptions in more than 50% but less than 90% of their calls. Method `with_suffix`[11] of the pathlib library in CPython presents an example in this category (Figure 7). This method returns a new path with the file suffix changed, performing several validations in the parameter `suffix`. Method `with_suffix` receives 91 calls, from which 63 (69%) raise the exception `ValueError`, indicating a problem in the suffix. Unlike the previous two categories, here, raising the exception is more common than not raising it. In fact, the presented method in Figure 7 is more "defensive" in its implementation, with three `raise` statements.

Almost always raising exceptions: This category includes methods that raise exceptions in 90% or more of their calls. We present two Pylint methods: `set_msg_status`[12] and `get_module_and_frameid`.[13] In method `set_msg_status` (Figure 8), 3,384 in 3,564 (95%) calls raise the exception `KeyError`. Figure 9 presents an even more strict case: 100% (all 3,618) of the calls to method `get_module_and_frameid` raise the exception `AttributeError`. In this case, raising the exception is a

[11]https://github.com/python/cpython/blob/0c5fc272/Lib/pathlib.py#L770

[12]https://github.com/pylint-dev/pylint/blob/c25923f3/pylint/utils/file_state.py#L184

[13]https://github.com/pylint-dev/pylint/blob/c25923f3/pylint/utils/utils.py#L103

```python
def with_suffix(self, suffix):
    """Return a new path with the file suffix changed.  If the path
    has no suffix, add given suffix.  If the given suffix is an empty
    string, remove the suffix from the path.
    """
    f = self._flavour
    if f.sep in suffix or f.altsep and f.altsep in suffix:
        raise ValueError("Invalid suffix %r" % (suffix,))
    if suffix and not suffix.startswith('.') or suffix == '.':
        raise ValueError("Invalid suffix %r" % (suffix))
    name = self.name
    if not name:
        raise ValueError("%r has an empty name" % (self,))
    old_suffix = self.suffix
    if not old_suffix:
        name = name + suffix
    else:
        name = name[:-len(old_suffix)] + suffix
    return self._from_parsed_parts(self._drv, self._root,
                                   self._parts[:-1] + [name])
```

Fig. 7: Method with_suffix of the pathlib library in CPython. Calls: 91; exception-raising calls: 63 (69%).

terminating condition for the `while` loop. It is interesting to note that in such methods, the "normal" behavior is actually to raise the exception. That is, the methods are implemented in such a way that raising is part of the expected behavior.

```python
def set_msg_status(
    self,
    msg: MessageDefinition,
    line: int,
    status: bool,
    scope: str = "package",
) -> None:
    """Set status (enabled/disable) for a given message at a given line."""
    assert line > 0
    if scope != "line":
        # Expand the status to cover all relevant block lines
        self._set_state_on_block_lines(
            self._msgs_store, self._module, msg, {line: status}
        )
    else:
        self._set_message_state_on_line(msg, line, status, line)

    # Store the raw value
    try:
        self._raw_module_msgs_state[msg.msgid][line] = status
    except KeyError:
        self._raw_module_msgs_state[msg.msgid] = {line: status}
```

Fig. 8: Method set_msg_status of Pylint. Calls: 3,564; exception-raising calls: 3,384 (95%).

Observation 4: Typically, exception-raising at runtime is a rare phenomenon and indicates "abnormal" behaviors. However, in some cases, exception-raising is frequent and represents "normal" behaviors.

C. RQ3: How do exception-raising methods and calls vary by system?

In this last RQ, we explore the variation of exception-raising methods, and calls vary per system. Table VI de-

```python
def get_module_and_frameid(node: nodes.NodeNG) -> tuple[str, str]:
    """Return the module name and the frame id in the module."""
    frame = node.frame()
    module, obj = "", []
    while frame:
        if isinstance(frame, Module):
            module = frame.name
        else:
            obj.append(getattr(frame, "name", "<lambda>"))
        try:
            frame = frame.parent.frame()
        except AttributeError:
            break
    obj.reverse()
    return module, ".".join(obj)
```

Fig. 9: Method get_module_and_frameid of Pylint. Calls: 3,618; exception-raising calls: 3,618 (100%).

tails the exception-raising methods and calls per system. Column *"Exception-Raising Methods"* presents the absolute (#) and relative (%) number of exception-raising methods. Column *"Exception-Raising Calls"* presents the distribution of exception-raising calls for the exception-raising methods, detailing the first quartile (Q1), second quartile (Q2), and third quartile (Q3). For example, Table VI shows that Pylint's tests execute 1,537 methods, of which 273 (17.8%) are exception-raising. Among these 273 exception-raising methods, the median proportion of exception-raising calls is 7%, with a first quartile of 1% and a third quartile of 50%.

Exception-raising methods: Overall, we notice a large variation in the ratio of exception-raising methods per system, ranging from 5.6% to 53.3%. The system with the smallest proportion of exception-raising methods is Error Corrector, with 5.6% (31 out of 550 methods), while the system with the largest proportion is csv, with 53.3% (8 out of 15 methods).

It is worth noting that only 3 out of 25 systems (Cookiecutter, configparser, and csv) have 50% or more exception-raising methods. In contrast, the remaining 22 systems have less than 50% exception-raising methods. Among those, two systems have less than 10%.

Observation 5: The majority of systems (22 out of 25) contain more exception-free methods than exception-raising methods.

Exception-raising calls: We also find a large variation in the proportion of exception-raising calls. The median of exception-raising calls per method varies from 1% (calendar) to 100% (Jupyter Client), the first quartile varies from <1% (logging, argparse, and tarfile) to 21% (Jupyter Client), and the third quartile varies from 3% (gzip) to 100% (Jupyter Client). Among the 25 systems, 19 (76%) have a median value of less than 30%, while only 6 systems (24%) have a median value greater than 30%. Notice that not all systems have an equivalent number of exception-raising methods. For instance, Pylint has 273 exception-raising methods, while calendar has only 4. Consider only the top-5 systems with the most exception-raising methods: Pylint (273), email (120), Flask (78), Rich (73), and Dateutil (63). In this case, the median of exception-raising calls per method is much smaller, ranging from 4% (email) to 15% (Flask).

Observation 6: Most systems (19 out of 25) have a median proportion of exception-raising calls per method below 30%.

IV. Discussion and Implications

A. *Most Exceptional Behaviors Are Rarely Exercised*

The literature reports that developers are more likely to test expected behaviors than unexpected ones, such as exceptional behaviors [8]–[18]. In this study, we contribute to this research line by showing that most methods that raise exceptions at runtime do so infrequently. In other words, given a method that raises an exception at runtime under specific conditions, it is likely that such an exception will rarely be triggered by the existing test suite. For example, we find that 50% of the exception-raising methods actually raise exceptions in at most 10% of their calls. This can be problematic because the fact that error handling code such as exceptions are not often executed makes them a natural place to hide bugs [18], [33].

Implication 1: We envision the development of novel tools to support exercising exceptional behaviors more effectively. For example, such tools could identify the tests that cover exceptional cases, including exceptions not propagated to tests. Such tools could alert developers whether exceptional cases are adequately tested or missing in the test suite.

B. *Some Exceptional Behaviors Are Frequently Exercised*

The majority of methods that raise exceptions at runtime do so infrequently. However, some methods actually present the opposite behavior: they raise exceptions more frequently. That is, given a method that raises an exception at runtime, in rare cases, the exception is frequently triggered by the existing test suite. For example, we find that 11.8% of the exception-raising methods raise exceptions in 90% or more of their calls. For such methods, the "normal" behavior is to raise the exception.

One factor that may explain such a variation is the origin of the exception. For example, exceptions can be explicitly raised by the system under test (as in Figure 1a), or implicitly raised by external dependencies or standard libraries (as in Figure 1b) [2], [34]. An exception raised directly by the system seems more relevant and testable behavior, whereas an exception raised somewhere deeper in the call stack may indicate less direct relevance.

Although we did not deeply explore the origin of the exceptions, RQ1 provides initial insights in this direction. Table III shows that most exceptions are raised by up to three methods, suggesting they are potentially specific, like the ones originating from the system under test (*e.g.,* `SMTPSenderRefused`). In contrast, a few exceptions are raised by ten

TABLE VI: Exception-raising methods and calls by system. Q1: first quartile; Q2: second quartile (median); Q3: third quartile

System	Executed Methods	Exception-Raising Methods #	Exception-Raising Methods %	Exception-Raising Calls (%) Q1	Q2	Q3
Pylint	1,537	273	17.8%	1%	7%	50%
Rich	589	73	12.4%	1%	7%	29%
Error Corrector	550	31	5.6%	18.5%	50%	80%
BentoML	319	56	17.6%	8%	27%	72%
Flask	284	78	27.5%	4%	15%	70%
Dateutil	241	63	26.1%	2%	9%	22%
Requests	174	44	25.3%	2%	12.5%	34%
Jupyter Client	138	31	22.5%	21%	100%	100%
Cookiecutter	66	35	53%	6%	12%	31%
Six	32	7	21.9%	1%	60%	99%
email	381	120	31.5%	1%	4%	25%
logging	216	49	22.7%	<1%	5%	24%
argparse	126	38	30.2%	<1%	2%	31%
collections	112	26	23.2%	9%	58%	100%
pathlib	97	42	43.3%	7%	41.5%	62%
tarfile	89	33	37.1%	<1%	5%	19%
configparser	82	41	50%	2%	8%	23%
calendar	63	4	6.3%	1%	1%	11%
ftplib	51	21	41.2%	1%	4%	33%
difflib	47	9	19.1%	16%	22%	44%
imaplib	47	18	38.3%	3%	4%	18%
smtplib	43	19	44.2%	4%	8%	31%
os	41	20	48.8%	10%	45%	75%
gzip	32	11	34.4%	1%	2%	3%
csv	15	8	53.3%	2%	9%	71%
All	5,372	1,150	21.4%	1%	10%	48%

or more methods, indicating they are generic, like the ones originating from standard libraries (*e.g.,* `ValueError`).

Implication 2: We call attention to the fact that exception-raising behaviors are not necessarily "abnormal" or rare. The fact that a test exercises a method that raises exceptions does not necessarily indicate it is testing an "abnormal" behavior. Raising an exception may simply be part of the method's "normal" behavior. Researchers working on exceptional behavior testing [1], [2], [20]–[22] should be aware of such methods to avoid failing to detect abnormal behaviors. In such cases, it would be important to consider other factors, such as the origin of the raised exceptions (*i.e.,* SUT or external).

C. Refactoring Expensive `try/except` Blocks

Our results show that different exception-raising methods may have distinct frequencies of exception-raising calls, ranging from rare to almost always. One explanation may be the Python coding style EAFP (easier to ask for forgiveness than permission), which assumes the existence of valid keys or attributes, catching the exceptions `KeyError` or `AttributesError` if the assumption proves false.[14] According to the Python documentation, this coding style is characterized by the presence of many `try/except` blocks. However, note that `try/except` blocks are very efficient when no exceptions are raised, but catching an exception is actually expensive.[15] Thus, a `try/except` block that checks the existence of a key should expect the dictionary to have the key almost all the time and rarely raise the exception. This way, one possible refactoring is replacing such `try/except` blocks with more efficient solutions, *e.g.,* checking the key existence with the `in` keyword in `if/else` blocks.

Interestingly, we find that many methods commonly raise the exceptions `KeyError` and `AttributesError`, indicating they are candidates to be refactored. For example, Figure 10 shows four methods of project DateUtil that frequently raise `KeyError`: `ampm` (it raises exceptions in 96% of the calls), `hms` (96%), `weekday` (83%), and `month` (75%).[16] Another extreme case happens in method `_infer_dunder_doc_attribute`[17] of project Pylint. In this method, the exception `KeyError` happens in 99.9% (2,729 of 2,731) of the calls. Therefore, such methods could be refactored to check the existence of the key using the `in` keyword and `if/else` blocks to increase efficiency.

Implication 3: `try/except` blocks that frequently raise exceptions at runtime when checking the existence of keys or attributes are strong candidates for refactoring. Our study can spot those less efficient `try/except` blocks that can be replaced by more efficient `if/else` blocks checking the key existence. In this context, we envision that future research in the context of refactoring [23]–[28] could leverage the execution frequency of language constructors (*e.g.,* `try/except` blocks) to detect novel refactoring opportunities.

[14]https://docs.python.org/3/glossary.html#term-EAFP
[15]https://docs.python.org/3/faq/design.html#how-fast-are-exceptions
[16]https://github.com/dateutil/dateutil/blob/9eaa5de584f9f374/src/dateutil/parser/_parser.py#L322-L346
[17]https://github.com/pylint-dev/pylint/blob/5c59b48acb5e0c8e/pylint/checkers/base/docstring_checker.py#L32-L35

```python
def weekday(self, name):
    try:
        return self._weekdays[name.lower()]
    except KeyError:
        pass
    return None

def month(self, name):
    try:
        return self._months[name.lower()] + 1
    except KeyError:
        pass
    return None

def hms(self, name):
    try:
        return self._hms[name.lower()]
    except KeyError:
        return None

def ampm(self, name):
    try:
        return self._ampm[name.lower()]
    except KeyError:
        return None
```

Fig. 10: Methods that frequently raise the exception `KeyError`. The `try/except` blocks can be replaced by `if/else` blocks to increase efficiency.

V. Threats to Validity

Origin of the exceptions. One factor that may affect the variation of the exception-raising methods and calls is their origin. For example, exceptions directly raised by the SUT seem more relevant and testable than those raised by third-party libraries and propagated to the SUT [2], [34]. Although we provided initial insights into the type of exceptions, further studies are needed to better understand the effects of the origin of the exceptions.

Public APIs and implementation details. In this study, we do not distinguish between public APIs (*e.g.,* public methods) and implementation details (*e.g.,* private or internal methods) when exploring the exceptions raised at runtime. Implementation details exist to support public APIs [35]. While public APIs should be well-tested, implementation details can sometimes be indirectly tested by public APIs [35]. However, complex implementation details can be directly tested in test suites [35]. Further studies are needed to explore the differences between exception-raising in public APIs and implementation details.

Generalization of the results. In this study, we analyzed exception-raising from 25 popular and real-world Python test suites. Despite these observations, our findings – as usual in empirical software engineering – cannot be directly generalized to other systems or implemented in other programming languages. Further studies should be performed on other software ecosystems and programming languages.

VI. Related Work

Overall, the literature agrees that developers are more likely to test normal behaviors than abnormal ones [8]–[19]. This happens because the expected behavior of the program is often simpler to test. Another factor is that developers may lack test expertise, focusing on only testing the "happy cases" [12]. In an experiment with developers, Teasley *et al.* [9] found evidence of using a positive test strategy (*i.e.,* testing the expected behavior), which was partially mitigated by increasing the expertise of the developers.

Many studies explore exceptional behaviors from the test perceptive [1], [2], [18], [20]–[22], [36]. Goffi *et al.* created test oracles for exceptional behaviors from Javadoc comments [18]. The authors also reported that exceptional behaviors are poorly covered by tests. Lima *et al.* explored exception handling testing practices in Java libraries and found that `catch` blocks are less covered [20]. Marcilio and Furia provided a large-scale empirical study of exceptional tests in Java, that is, tests that may trigger exceptional behaviors [2]. The authors analyze multiple patterns Java developers can use to write exceptional tests and detect several characteristics of such tests. For example, exceptional tests represent 13% of all tests, tend to be larger, and are mostly written using `try/catch` blocks. In this context, Dalton *et al.* [1] analyzed exceptional behavior testing in Java, that is, tests that expect exceptions to be raised. The results showed that 60.9% of the projects have at least one test dedicated to verifying exceptional behavior, concluding that exceptional behavior testing is a rare phenomenon. In common, both studies [1], [2] explore the tests that expect exceptions to be raised.

Despite the various studies on exceptional behavior testing, they are mainly concentrated on analyzing raised exceptions that propagated to tests. Our study provides a complementary perspective: we analyzed all raised exceptions at runtime, not only the ones that propagate to tests. Furthermore, we deeply explored the frequency of raised exceptions at runtime, which is not the focus of any prior study.

VII. Conclusion

We provided an empirical study to explore how frequently exceptional behaviors are tested in real-world systems. We analyzed the test suites of 25 Python systems, covering 5,372 executed methods, 17.9M calls, and 1.4M raised exceptions. Our main findings can be summarized as follows: (1) 21.4% of the executed methods do raise exceptions at runtime; (2) in methods that raise exceptions, on the median, 1 in 10 calls exercise the exceptional behaviors; and (3) close to 80% of the methods that raise exceptions at runtime do so infrequently, while only about 20% raise exceptions more frequently; and (4) most systems contain more exception-free methods than exception-raising methods. Based on our results, we discussed practical implications for researchers and practitioners, including the development of novel tools to more effectively support exercising exceptional behaviors and the refactoring of expensive `try/except` blocks.

In future work, we plan to perform more qualitative analysis on the analyzed methods, calls, and exceptions, for example, by exploring the origin of the raised exceptions (*i.e.,* SUT or external) [2], [34] and whether they come from public APIs or implementation details [35]. We also intend to develop tools to identify the tests that cover exceptional cases, including exceptions not propagated to tests. Finally, we plan to provide an empirical study to quantify and qualify the refactoring to replace expensive `try/except` blocks.

Acknowledgments

This research is supported by CAPES, CNPq, and FAPEMIG.

References

[1] F. Dalton, M. Ribeiro, G. Pinto, L. Fernandes, R. Gheyi, and B. Fonseca, "Is exceptional behavior testing an exception? An empirical assessment using Java automated tests," in *International Conference on Evaluation and Assessment in Software Engineering*, 2020, pp. 170–179.

[2] D. Marcilio and C. A. Furia, "How java programmers test exceptional behavior," in *International Conference on Mining Software Repositories*. IEEE, 2021, pp. 207–218.

[3] S. C. Reid, "An empirical analysis of equivalence partitioning, boundary value analysis and random testing," in *International Software Metrics Symposium*. IEEE, 1997, pp. 64–73.

[4] C. Kaner, S. Padmanabhan, and D. Hoffman, *The Domain Testing Workbook*. Context Driven Press, 2013.

[5] M. Aniche, *Effective Software Testing: A developer's guide*. Simon and Schuster, 2022.

[6] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[7] T. Winters, H. Wright, and T. Manshreck, "Software Engineering at Google: Lessons Learned from Programming over Time," 2020.

[8] M. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4925–4946, 2021.

[9] B. E. Teasley, L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman, "Why software testing is sometimes ineffective: Two applied studies of positive test strategy," *Journal of Applied Psychology*, vol. 79, no. 1, p. 142, 1994.

[10] I. Salman, B. Turhan, and S. Vegas, "A controlled experiment on time pressure and confirmation bias in functional software testing," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1727–1761, 2019.

[11] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, "Effects of negative testing on TDD: An industrial experiment," in *International Conference on Agile Processes in Software Engineering and Extreme Programming*. Springer, 2013, pp. 91–105.

[12] S. H. Edwards and Z. Shams, "Do student programmers all tend to write the same software tests?" in *Conference on Innovation & technology in Computer Science Education*, 2014, pp. 171–176.

[13] R. Mohanani, I. Salman, B. Turhan, P. Rodríguez, and P. Ralph, "Cognitive biases in software engineering: a systematic mapping study," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1318–1339, 2018.

[14] L. M. Leventhal, B. M. Teasley, D. S. Rohlman, and K. Instone, "Positive test bias in software testing among professionals: A review," in *International Conference on Human-Computer Interaction*. Springer, 1993, pp. 210–218.

[15] L. Bijlsma, N. Doorn, H. Passier, H. Pootjes, and S. Stuurman, "How do students test software units?" in *International Conference on Software Engineering: Software Engineering Education and Training*. IEEE, 2021, pp. 189–198.

[16] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.

[17] G. R. Bai, J. Smith, and K. T. Stolee, "How students unit test: Perceptions, practices, and pitfalls," in *ACM Conference on Innovation and Technology in Computer Science Education*, 2021, pp. 248–254.

[18] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *International Symposium on Software Testing and Analysis*, 2016, pp. 213–224.

[19] A. Hora, "Monitoring the execution of 14k tests: Methods tend to have one path that is significantly more executed," in *International Conference on the Foundations of Software Engineering*, 2024, pp. 532–536.

[20] L. P. Lima, L. S. Rocha, C. I. Bezerra, and M. Paixao, "Assessing exception handling testing practices in open-source libraries," *Empirical Software Engineering*, vol. 26, no. 5, p. 85, 2021.

[21] J. Zhang, Y. Liu, P. Nie, J. J. Li, and M. Gligoric, "Generating exceptional behavior tests with reasoning augmented large language models," *arXiv preprint arXiv:2405.14619*, 2024.

[22] H. Yoshioka, Y. Higo, S. Matsumoto, S. Kusumoto, S. Itoh, and P. T. T. Huyen, "Do exceptional behavior tests matter on spectrum-based fault localization?" in *International Conference on Product-Focused Software Process Improvement*. Springer, 2023, pp. 399–414.

[23] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[24] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and software Technology*, vol. 58, pp. 231–249, 2015.

[25] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. Newman, and A. Ouni, "How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations," in *International Conference on Mining Software Repositories*, 2024, pp. 202–206.

[26] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *International Conference on Software Engineering: Future of Software Engineering*. IEEE, 2023, pp. 31–53.

[27] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, T. Bryksin, and D. Dig, "Together We Go Further: LLMs and IDE Static Analysis for Extract Method Refactoring," *arXiv preprint arXiv:2401.15298*, 2024.

[28] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using large language models with few-shot examples," in *Asia-Pacific Software Engineering Conference*. IEEE, 2023, pp. 151–160.

[29] A. Hora, "SpotFlow: Tracking Method Calls and States at Runtime," in *International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 35–39.

[30] sys.settrace, https://docs.python.org/3/library/sys.html#sys.settrace, November, 2024.

[31] A. Hora, "What code is deliberately excluded from test coverage and why?" in *International Conference on Mining Software Repositories*, 2021, pp. 392–402.

[32] ——, "Excluding code from test coverage: practices, motivations, and impact," *Empirical Software Engineering*, vol. 28, no. 1, p. 16, 2023.

[33] Fix error handling first, https://nedbatchelder.com/text/fix-err-hand.html, November, 2024.

[34] S. McConnell, *Code Complete*. Pearson Education, 2004.

[35] Prefer Testing Public APIs Over Implementation-Detail Classes, https://testing.googleblog.com/2015/01/testing-on-toilet-prefer-testing-public.html, November, 2024.

[36] C. Artho, A. Biere, and S. Honiden, "Enforcer–efficient failure injection," in *International Symposium on Formal Methods*. Springer, 2006, pp. 412–427.