Property-Based Testing in Practice using Hypothesis

In-depth study on how developers use Property-Based Testing in Python using Hypothesis

David de Koning

Supervisor(s): Andreea Costea, Sára Juhošová

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2025

Name of the student: David de Koning
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Sára Juhošová, Marco Zuñiga Zamalloa

Abstract

Property-based testing (PBT) allows developers to specify high-level properties that should hold for a range of inputs, which are then automatically generated by the testing framework. Despite its theoretical appeal, PBT is not widely used in the real world. To better understand how PBT is used in practice, we present a qualitative and quantitative study of 87 property-based tests written with the Hypothesis framework in seven widely used Python projects, including `cpython`, `pandas`, and `jax`.

Our analysis reveals that while PBTs are relatively rare in these repositories, they are often simple in structure and highly effective at expressing functional properties. The most common patterns include round-trip checks and comparisons against test oracles, which account for a significant portion of the test suite. We also observed a high rate of custom generator usage (39.1%), but no use of custom shrinking, suggesting strong defaults in Hypothesis. The dataset was variable in stylistic and structural choices, ranging from single-assertion tests to complex hardware-dependent cases.

This study provides new insights into the practical use of PBT in Python, expands on prior quantitative work with qualitative findings, and identifies concrete implications for improving testing frameworks and educational resources. We conclude with recommendations for supporting broader adoption of PBT and outline directions for future research, including cross-language comparison, automated annotation, and temporal analysis of test evolution.

Note on the use of Generative AI in this Research Paper. OpenAI's GPT-4[1] Large Language Model (LLM) was used in a supporting role, both in the research and in the writing of this paper. All LLM factual answers that played a significant role in the research have been checked. No LLM answers have been directly copied, they were used only as inspiration, or to help solve small problems.
An AI-assisted spellchecker based on GPT-4 has been used in writing this paper.
More information on the use of Generative AI can be found in Section 4.5.

1 Introduction

Property-based testing (PBT) is an approach to software testing in which general properties – expected relations or invariants about a program's behavior – are specified and automatically tested with large amounts of input data. In this way, developers get better guarantees about their code without the need for resource-demanding tools like formal proofs. Although this seems like the ideal way to do software testing, practice shows that property-based testing is not widely adopted within the programming community. For example, Hypothesis, the largest PBT framework for Python, is used by just 4% of the Python user base, while pytest, the largest general testing framework for Python, is used by 50% of Python users [1].

```python
# Test that the elements in the sorted list the same as the original list
@given(st.lists(st.integers(), min_size=1, max_size=10)
def test_merge_sort_same_elements(xs: list):
    sorted_list = merge_sort(xs)
    for elem in xs:
        assert elem in sorted_list
    assert len(sorted_list) == len(xs)
```

Figure 1: Example of a property-based test that checks whether the elements in a sorted list are the same as in the original list.

Unlike traditional unit tests that test specific input-output combinations, property-based tests focus on general rules or invariants that should be true for all inputs and outputs in a certain domain. A PBT framework generates these inputs automatically, based on some domain specification given by the programmer.

In Python, the most widely used framework for property-based testing is Hypothesis[2]. This framework allows developers to write property-based tests using decorators such as `@given`, which automatically generate inputs based on an input domain provided by the developer.

Figure 1 is an example of a simple Hypothesis test that checks whether a custom sorting function retains all original elements. The most important features of a property-based test illustrated by this example are the following:

- a **generator** that generates lists of numbers,
- a **filter** that limits the size of these lists between 1 and 10,
- a **system under test (SUT)** for which a certain property should hold, marked by `merge_sort(xs)`, and
- **assertions** that check whether the given propositions hold.

1.1 Terminology

Given the prior explanation of a basic example of a property-based test, the formal definitions of the important components of PBT can be understood:

Property-Based Testing. Given a system under test S, a domain of test data D, and a set of properties $P = \{p_0, ..., p_{m-1}\}$, property-based testing is the process of automatically generating test data $d_0, d_1, ..., d_{n-1} \in D$ using a generator and running a set of assertions A_{p_j} to check that $p_j(x)$ or $p_j(x, y)$ holds, for either:

1. $S(f(d_i))$, where f is some function that transforms the generated data to an input for S, or
2. $S(b)$, where d_i is used to set up the environment for S and b is some constant (possibly $\emptyset$),

such that x and y are some environmental variables or (some transformations of) the generated input and the output of S, for all $(i, j) \in \{0, ..., n-1\} \times 0, ..., m-1$.

[1] https://openai.com/index/gpt-4-research/

[2] https://hypothesis.readthedocs.io/en/latest/

Property. A property p is a rule in the form of one of two types:

1. An invariant $p\,(x)$: a proposition on some data, x – usually the input or the output data of some system under test – that should always be true.
2. A relation $p(x, y)$: a proposition that relates two data, x and y – usually the input data and the output data of some system under test.

Assertion. Given some proposition $q\,(x)$, an assertion a_p is a function called from within a property-based test that checks whether some proposition $q(x)$ is true or false. Upon calling the assertion $a_q(x)$ from a property-based test case, if $q(x)$ holds, the execution of the test case is continued. If $q(x)$ does not hold, the execution of the test case is stopped and marked as failing.

System Under Test (SUT). A system under test S is a code entity for which some functionality is tested. It can be a function, a module, or a system. An SUT is usually called from a test with some input x, such as $S(x)$. If S is called without any input, this can be denoted as $S(\emptyset)$.

Strategy/Generator. Given a domain of test data D, a strategy or generator is a programmatical construct that provides n test data $d_0, d_1, \ldots, d_{n-1} \in D$, that are passed onto a property-based test to create n test cases that test a set of properties.

1.2 Research Context

Understanding how PBT is used in practice is essential to improving its adoption. We need to know what its strengths and limitations are and in what way it can fit the needs of the average programmer. A good starting point is to look at the small portion of programs that do employ PBT: how is PBT currently used?

While Corgozinho et al. [2] found that some PBT patterns are used much more often than others in popular Python projects using Hypothesis, these findings are based on a mostly quantitative analysis of public repositories, though a qualitative analysis leads to "results [that] are richer and more informative," [3]. Corgozinho et al. acknowledge the limitations of their dataset and call for further research into both Hypothesis and alternative PBT frameworks.

In this study we respond to that call by employing qualitative analysis techniques in combination with quantitative analysis, to investigate how PBT is used in large Python projects. This approach allows us to uncover nuanced patterns in how developers implement property-based tests, beyond frequency counts. In addition, we compare our insights to similar studies in other programming languages and PBT frameworks.

We have chosen to study Hypothesis because it is a popular framework for property-based testing in Python, with millions of weekly downloads[3]. In this research paper, we will answer the following research question:

> **How is property-based testing used in real-world Python projects with Hypothesis?**

[3] https://pypistats.org/packages/hypothesis

Corgozinho et al. [2] studied 86 property-based tests randomly sampled from 30 different projects. In this study, we investigate 87 property-based tests. The tests have not been randomly sampled from the chosen projects, but rather all property-based tests from the chosen projects are included. In this way, we aim to obtain a deeper understanding of the role of PBT in each individual project.

To answer the main research question, we have divided it into four concrete sub-questions:

RQ1: *For which purposes and in which parts of the code base is Hypothesis used in real-world Python projects?*
Contribution: The goal of this question is to list the main roles that property-based tests have, especially in comparison to other testing paradigms that are used within the project. This information can be used to better understand why property-based testing is used in general, which can pave the way for improvements in property-based testing frameworks.

RQ2: *What high-level property categories are typically tested using property-based testing in Python projects?*
Contribution: This question is closely related to RQ1, in the sense that it will reveal why developers use PBT. However, it focuses on how individual tests are implemented, instead of the meaning of the tests in the context of the projects.

RQ3: *How are property-based tests structurally and stylistically expressed?*
Contribution: To answer this research question, we look at the specific syntax used to express property-based tests, as well as the structure of the tests. This will show how individual tests are structurally designed and how complex they are. This knowledge could form the basis for research on how complex PBTs can be split into simpler ones to more easily find bugs.

RQ4: *How and why do developers utilize advanced Hypothesis features, such as custom generation strategies and explicit shrinkers, in real-world Python test-suites?*
Contribution: Here, we aim to find out if Hypothesis is mainly used in the 'basic' form – i.e. using existing generators and shrinkers for basic data types (e.g. booleans, numbers, strings, and lists) – or whether developers implement support for more advanced data types. Together with RQ3, it answers the question of how complex property-based tests typically are.

Similar research questions are answered by our colleagues for different combinations of programming language and frameworks. Their findings can be compared to ours to get a deeper understanding of how property-based testing is used in real-world software projects. The four other frameworks researched are:

- `proptest` in `Rust` [4],
- `quickcheck` in `Rust` [5],
- `jqwik` in `Java` [6], and
- `QuickCheck` in `Haskell` [7].

The remainder of this paper is organized as follows. Section 2 introduces key concepts in property-based testing. Section 3 details the research methodology. Section 4 presents the results, followed by a discussion of ethical considerations

and challenges in Section 5. Section 6 provides an interpretation of the findings, and Section 7 concludes with answers to the research questions and directions for future work.

2 Methodology

In exploring the use of property-based testing in Python projects, we aim to go further than counting how often PBT is used, and instead to see how it is used. This approach involves constructing a dataset of real-world repositories (Section 2.1) and, using the open coding approach to analyze the collected tests (Section 2.2).

2.1 Dataset Collection

To understand how property-based testing is used in real-world Python projects, we constructed a dataset of open-source repositories that depend on Hypothesis. We used the `github-dependents-info` tool[4] to retrieve all GitHub repositories that list Hypothesis as a dependency, selecting only those with at least 100 GitHub stars to ensure a baseline of popularity and quality. This initial step, run on May 1, 2025, yielded 494 repositories.

Next, we filtered these repositories to keep only those that appeared to contain at least two property-based tests. This was determined by shallow-cloning each repository and counting occurrences of the `@given` decorator in the source code using a custom Python script[5]. While this heuristic may find some false positives (e.g. code comments), it was a practical way to find PBT usage.

From the resulting 273 repositories, we manually selected a smaller subset for qualitative analysis. Our selection consists of seven repositories with 25 or fewer PBTs, which allows us to completely examine all tests of the repositories within the time frame of this project. We chose this approach to gain better insights into how PBT is used in each individual repository. The final set of repositories can be found in Figure 2.

The number of PBTs identified in each project varies substantially. Similarly, PBT density – defined as the number of PBTs relative to the total number of tests – ranges from 0.073% in `cpython` to 0.500% in `jax`. Thus, the importance of property-based testing seems to differ between repositories, giving us an interesting sample of Python repositories that use PBT.

2.2 Data Analysis

To analyze the property-based tests, we used an open coding strategy [8]. This method emphasizes iterative exploration of the data to derive concepts and themes from the content itself.

We opted for a fully manual analysis of every collected PBT as opposed to a fully automated or a hybrid approach where the tests are scanned for pre-determined features. This is because of the explorative nature of the research; we did not know exactly what we were looking for in the tests before looking at the tests. Complete manual analysis allowed us to notice patterns in the dataset without being confined to the constrictions of a pre-established structure. Furthermore, features such as custom assertions can be expressed in many ways in code, making automated analysis needlessly complex and time-consuming.

As such, we printed out all collected property-based tests on paper. Then, we reviewed them manually, annotating features and writing memos [8] for syntax and patterns that stood out. Initial codes were written down for observable features such as input generation strategies, test structure, types of properties asserted, and relationships to other software elements.

As recurring concepts emerged, we revisited the tests to validate and refine the codes and concepts. The concepts were then structured into a spreadsheet where each row represents an individual property-based test and each column captures a specific feature, as illustrated in Figure 3. This tabular representation allowed us not only to easily summarize the most important findings but also to compare the use of PBT between testing frameworks. A full list of features and their explanations can be found in Appendix A.

3 Results

In this section, we present the results of our qualitative analysis of property-based tests found in seven popular open-source Python repositories. Due to space constraints, we cannot discuss every detail here. A complete overview of how we found and collected the property-based tests can be found online[7]. The data set resulting from our analysis of all 87 property-based tests is also available [9]. We present the results of our qualitative analysis below, organized by research question.

RQ1: For which purposes and in which parts of the code base is Hypothesis used in real-world Python projects?

The purpose of the use of Hypothesis can largely be derived from a few specific characteristics of the analyzed property-based tests:

Presence of PBT. Hypothesis was used in small amounts across the repositories, as indicated by the PBT density, as shown in Figure 2. When it was used, this was always in a file with a name that clearly indicated that it contained tests. For 86 out of the 87 tests, the test name started with `test_` (the exception being `run` from jax, which was the only test in the class `HypothesisTest`, making its function clear).

Generalizing Unit Tests. Several repositories featured PBTs that mirrored existing unit tests, sometimes appearing side-by-side in the same file. This suggests an attempt to generalize preexisting examples with generated data.

Input Generation. As shown in Figure 4, the generated inputs ranged from simple primitives to deeply nested or project-specific structures, where integer values were by far the most generated. One extreme example of complex input generation was `test_splash_attention` in `jax`, which involved generating multiple integers, tuples, data types, and custom objects such as `splash.SegmentIds`, `Mask`, and

[4] https://github.com/nvuillam/github-dependents-info

[5] https://github.com/DaKoning/hypothesis-dataset/blob/main/filter_repos.py

[6] https://github.com/XAMPPRocky/tokei

[7] https://github.com/DaKoning/hypothesis-dataset

name	description	# GitHub stars	LOC	# tests	# PBTs	PBT density
cpython	Python language reference implementation	66,662	811,325	28,790	21	0.073%
pandas	Data analysis and manipulation	45,288	508,034	18,452	15	0.081%
streamlit	Creating UIs for data apps	39,121	120,890	3,567	6	0.168%
gradio	Creating web UIs for ML models	37,778	78,413	762	2	0.262%
jax	High-performance ML and computations	32,089	357,886	4,797	25	0.512%
spaCy	Advanced natural-language processing	31,491	121,209	1,814	4	0.221%
numpy	Array and numerical computing	29,386	265,442	6,905	14	0.203%

Figure 2: Summary of the collected Python repositories that make use of Hypothesis. LOC: number of lines of Python code in the repositories as counted with `tokei`[6]. # tests: total number of tests in the repositories. # PBTs: number of property-based tests in the repositories. PBT density: number of PBTs relative to the total number of tests. The number of tests and property-based tests are found using a custom script that identifies tests by occurrences of `def test_`, and PBTs by occurrences of `@(*.)given`.

name	assertions	input	category
`test_base64`	4	binary	roundtrip
`test_count`	1	float	testOracle
`test_pickle`	1	binary	objectCached

Figure 3: Example of how a spreadsheet is used to summarize findings of PBT analysis.

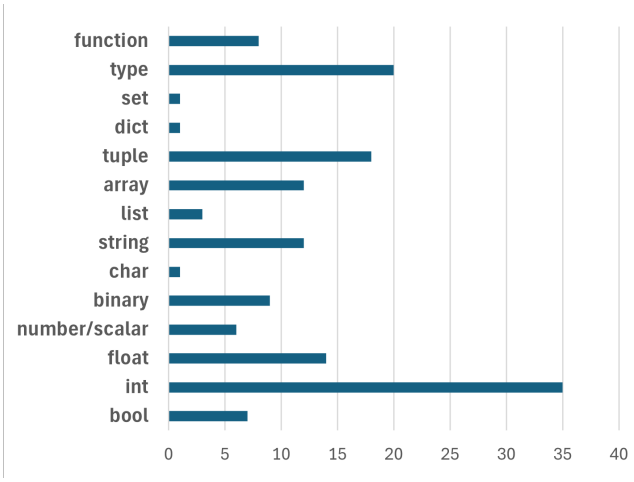

Figure 4: Frequencies of the types of inputs that are generated by Hypothesis in the 87 analyzed property-based tests. Each of the 87 tests may have multiple generated input types, as the framework can generated multiple inputs and input types that are a subtype of others are included.

`splash.BlockSizes`. In total, 62 out of 87 tests generated at least one input that was not a primitive.

As shown in Figure 6, 14 tests used Hypothesis to generate the system under test, usually by sampling from a list of functions. Figure 6 also shows that in 35 tests the generated inputs were used directly, which means that some transformation of the inputs took place in 52 tests.

Non-functional Properties. In most PBTs, the property that was asserted was a functional one. However, in 9 cases, PBTs checked non-functional properties of the SUT. This means that the properties defined certain quality characteristics that the system should have. For example, `numpy`'s `test_operator_object_left`[8] tests the performance of binary operators by checking whether they can return a result within a recursion depth of 200:

```
@given(sampled_from(objecty_things),
       sampled_from(
    binary_operators_for_scalar_ints),
       sampled_from(types + [rational]))
def test_operator_object_left(o, op, type_):
    try:
        with recursionlimit(200):
            op(o, type_(1))
    except TypeError:
        pass
```

Exceptions. 5 out of 87 property-based tests asserted that the system under test throws an exception. In all of these cases, asserting that an exception is thrown was only one of the possible execution flows of the test. That is, we found no tests that solely focused on asserting that an exception is thrown.

We also found that an exception could be thrown directly from the test function for 7 of the property-based tests. This way, the PBT can fail without an assertion, with the additional effect that its failure will be more obvious to the developer.

Tests in Development. We encountered different indicators that certain tests or systems under tests were in development. Such indicators included `TODO` markings or GitHub issue references in test comments, tests that were marked as expected to fail, and tests that were skipped.

One specific test that appeard to indicate that its SUT was in development, was `gradio`'s `test_is_in_or_equal_fuzzer`. In this PBT, the developer utilized a test oracle – used for generating the result expected from the SUT – which was almost identical to the SUT itself. It seemed that the developer expected the implementation of the SUT to change in the future, and wanted to ensure that specific behavior would remain functionally the same.

RQ2: What high-level property categories are typically tested using property-based testing in Python projects?

We identified 20 different categories that describe the properties that were tested by the PBT, as shown in Figure 5. Exist-

[8] ©2025 NumPy Developers; All Rights Reserved

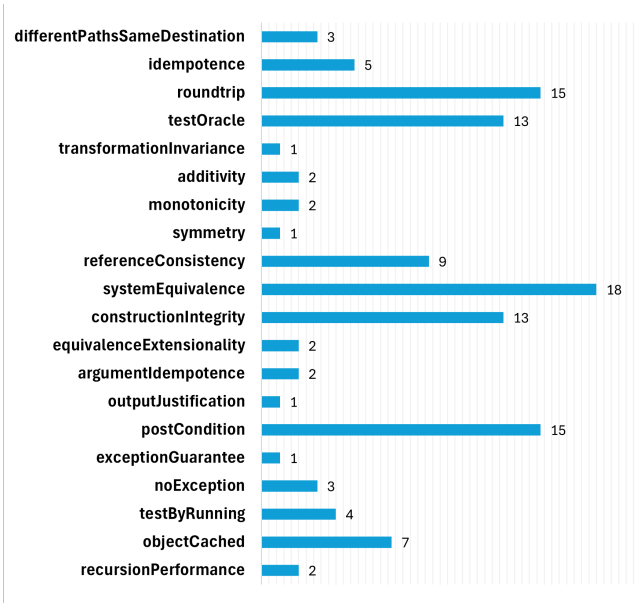

Figure 5: Frequencies of the PBT categories identified in analysis of 87 property-based tests. Each of the 87 tests may belong to multiple categories.

ing literature[9] provided us with the first 5 of these categories, while we devised the others ourselves based on recurring patterns and outstanding outliers. Many property-based tests fell into multiple different categories, usually because they tested multiple different properties or because the properties were complex.

While the full list of categories can be found in Appendix B, the following are the categories that were most prominent in our analysis:

- **Round-trip property**: Found in 15 tests. These tests test whether converting data back and forth (e.g. encoding then decoding) preserves its original form.
- **Oracle-based checking**: Seen in a large number of tests, with three main subcategories:
  - *testOracle* (13 tests): A dedicated function provides the expected result.
  - *systemEquivalence* (18 tests): A different implementation of the SUT is used for comparison.
  - *referenceConsistency* (9 tests): The SUT is a performance-optimized version of a reference implementation, often relying on specialized hardware.
- **Post-condition**: Identified in 15 tests. These tests assert a simple condition that should always be true after running the system under test. For example, we found the test `test_map_set_del`[10] from `streamlit`:

```
@given(m=stst.session_state(), key=stst.
    USER_KEY, value1=hst.integers())
def test_map_set_del(m, key, value1):
    m[key] = value1
    l1 = len(m)
    del m[key]
    assert key not in m
    assert len(m) == l1 - 1
```

This PBT asserts that the length of a map decreases by 1 after deleting an element.

- **Construction Integrity**: This property category, found in 13 tests, is about checking that the data gathered from an object after constructing it are consistent with the data provided for its construction. As an example, take `test_dow_parametric` from `pandas`[11]

```
def test_dow_parametric(self, ts, sign):
    # GH 53738
    ts = (
        f"{sign}{str(ts.year).zfill(4)}"
        f"-{str(ts.month).zfill(2)}"
        f"-{str(ts.day).zfill(2)}"
    )
    result = Timestamp(ts).weekday()
    expected = (
        (np.datetime64(ts) - np.
    datetime64("1970-01-01")).astype("
    int64") - 4
    ) % 7
    assert result == expected
```

The test asserts that the weekday number gathered from the construction of a `Timestamp` instance corresponds to the weekday number calculated in the test based on the input data.

RQ3: How are property-based tests structurally and stylistically expressed?

The implementation of the property-based testing paradigm differed between the repositories. Some tests were highly minimalistic, while others involved long chains of logic, numerous assertions, or interaction with complex infrastructure (e.g. hardware-dependent behavior or parameterized configurations). Here, we present some important design choices that stood out to us throughout the dataset. The frequencies of these choices are summarized in Figure 6.

Assertions. The average number of assertions per test ranged from 4.4 to 16.4, for the execution flow with the minimum and maximum number of assertions, respectively. 46 of the 87 tests used custom assertions, defined as asserting mechanisms that do not originate in a library.

Test Input Origin. While all tests used Hypothesis to randomly generated inputs, another origin of inputs are Hypothesis examples. This method allows the programmer to specify inputs that must always be tested on, in addition to the generated values. We saw Hypothesis examples implemented by 19 PBTs, which specified about 7 examples on average.

In addition, 22 tests were parameterized. Typically, some inputs were fixed via parameterization, while others were generated. This was common when all values of a certain input type (e.g. booleans or enums) needed to be tested.

[9] https://fsharpforfunandprofit.com/posts/property-based-testing-2/

[10] ©2025 Snowflake Inc.

[11] ©2025 Open source contributors.

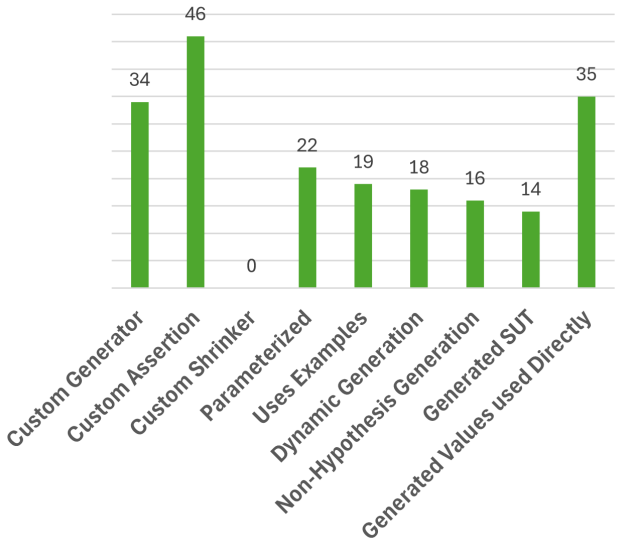

Figure 6: Distribution of 87 analyzed property-based tests across key features: whether they use custom input generation, whether they use custom assertions (not from a library), whether they use custom shrinking, whether they are parameterized, whether they use predefined examples (sets of inputs on which the tests should always be run), whether they use dynamic generation (generation of inputs within the test function), whether they use non-Hypothesis generation, whether the system under test is generated by Hypothesis, and whether the generated values are directly passed to the system under test.

Test Filtering and Skipping. We found the following mechanisms to limit or skip the execution of tests. Their frequencies can be found in Figure 7.

- **Input filtering**: Limiting the domain of input gen-reatino, performed inside the generator itself, e.g., `integers(min_value=0)`.
- **Assumptions**: These are runtime guards that abort a test run early if a condition is not met.
- **Explicit skipping**: Used when tests are unfinished or dependent on specific hardware. This was often used in `jax` tests, to skip execution on GPUs due to known issues.
- **Preconditions**: Applied as checks before running the test, sometimes based on system state or external data availability. Example from `cpython`'s `test_same_str`[12]:

```
VALID_KEYS = _valid_keys()
if not VALID_KEYS:
    raise unittest.SkipTest("No time zone
     data available")
```

[12]©2025 Python Software Foundation; All Rights Reserved

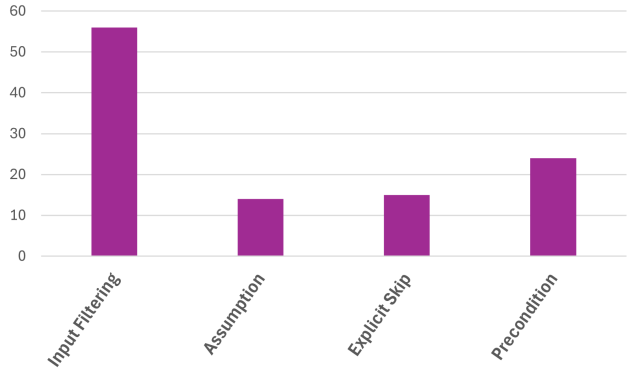

Figure 7: Frequencies of different methods of limiting or skipping test execution found in analysis of 87 property-based tests. Each test may have zero or more methods of limiting or skipping execution.

RQ4: How and why do developers utilize advanced Hypothesis features, such as custom generation strategies and explicit shrinkers, in real-world Python test-suites?

Input Generation. 34 out of 87 tests used a custom generator as opposed to a built-in generator. Custom generators were often to construct domain-specific objects.

Additionally, we saw that 16 tests used some method other than Hypothesis strategies to generate random values. Often, these values were created by a `random` method, not by a method created for testing purposes.

Custom Shrinking. While shrinking – minimizing a failing test input to simplify debugging – is an important part of property-based testing, we did not see any tests that implemented a custom version of a shrinker.

Settings. Other advanced Hypothesis features, accessed through the `settings` method, were not used much. They were used, however, when the developer wanted to change the amount of generated test examples or the time that the PBT is allowed to take.

4 Discussion

This section discusses the key insights drawn from our analysis of property-based tests (PBTs) across seven Python repositories. We interpret the observed patterns, compare them to existing work, reflect on technical challenges encountered during the study, address potential limitations and biases in our methodology, and discuss how this research was done responsibly.

4.1 Analysis of Results

A clear pattern emerging from our study is the dominance of two test patterns: *round-trip* and *test oracle* tests. These patterns are likely prevalent because they are conceptually straightforward and relatively easy to implement. Round-trip properties require a function and its inverse (or a way to undo its effect), while test oracles offer a logical way of comparing by defining or extracting expected outcomes. These patterns reduce the burden on developers to come up with abstract invariants, making them a simple start for property-based testing.

We also observed two distinct styles of using Hypothesis' input generation methods. In many tests, strategies were provided directly via decorators (e.g., `@given(integers())`). However, several tests made use of dynamic generation strategies from within the test body. This second approach allows developers to generate inputs conditionally or derive inputs based on computations or other generated values, thereby offering greater flexibility. This style seemed to be particularly useful when test inputs are interdependent or when constructing domain-specific structures.

Interestingly, we found that custom shrinking was not used in any of the 87 tests. Hypothesis provides automatic shrinking, which simplifies failing inputs for easier debugging. This might because the difficulty of writing custom shrinkers in Hypothesis is too high compared to the benefitis. Alternatively, it is likely that the default shrinking is sufficient for the needs of the developers, as the default shrinker (or "internal test-case reduction") is generic enough to work on any generator that a developer may write [10].

4.2 Answers to the Research Questions

Our subquestions were answered as follows:

RQ1: For which purposes and in which parts of the code base is Hypothesis used in real-world Python projects? Hypothesis is most frequently used for validating functional components, such as encoders/decoders, data format checks, and system improvement validators. It is rarely used for integration or system-level tests, but is often used to generalize unit tests.

RQ2: What high-level property categories are typically tested using property-based testing in Python projects? The dominant categories were *round-trip*, *test oracle*, and *system equivalence*. These are intuitive and low-effort to define, explaining their frequent use.

RQ3: How are property-based tests structurally and stylistically expressed? Most tests used a Hypothesis decorator with fixed strategies, though dynamic generation within the test body also occurred frequently. Assertion counts varied, and custom assertions were as common as standard assertions. Custom examples, parameterization and execution-flow restriction were also common.

RQ4: How and why do developers utilize advanced Hypothesis features, such as custom generation strategies and explicit shrinkers? Custom generation strategies were used in more than a third of the tests, indicating that Hypothesis makes generator design accessible. These generators were often used to generate complex data structures, or even the systems under test themselves. For shrinking, on the other hand, we found no custom implementations, suggesting that developers found the default shrinkers sufficient.

4.3 Comparison to Other Work

Our findings are in line with prior quantitative studies on PBT in Python. In particular, Goldstein et al. and Corgozinho et al. identified a low overall adoption of PBT and a skewed distribution of test styles, which we also observed. However, our study contributes additional qualitative insights into how Hypothesis is used in practice, which may make way for improvements to PBT frameworks and support future research.

We also compared our results with concurrent research into other property-based testing frameworks in different programming languages [4] [5] [6] [7]. For example, in Rust projects using `quickcheck` and `proptest`, a similar distribution of assertion counts per test was observed -— slightly more than half of all tests used a single assertion, mirroring our findings. The same dominant test patterns – *roundtrip* and *test oracle* -— also appeared frequently in Rust, although less so in Java projects using `jqwik`, where other patterns were more prominent.

Interestingly, custom generators were used in 39.1% of the Python tests we analyzed – significantly more than in Rust `quickcheck` (29.1%) or Java `jqwik` (36%). We think this is in part thanks to the relative usability and expressiveness of Hypothesis strategies, which make custom generation accessible even for more complex inputs.

In contrast, custom shrinking was used far more frequently in Rust `quickcheck` (20.9%) than in Python (0%), and was also absent in Java `jqwik`. This suggests that custom shrinking may be more accessible or necessary in languages and frameworks where default shrinking is limited or less effective, or where the debugging workflow benefits more from customized minimal inputs.

4.4 Threats to Validity

During our data collection, we encountered a number of technical challenges. First, our script for collecting PBTs relied on the presence of the `@given` decorator, and thus may have missed tests where Hypothesis was used in less conventional ways. In addition, decorator expressions that spanned several lines above the test definition were not reliably captured. Although we manually included such tests in our analysis, this limitation could be addressed in the future.

Secondly, we focused primarily on the content of the test functions themselves, and not on attributes of the containing class or module. As a result, some contextual information – such as class-level parameters, assumptions, or test-skips – may not have been fully incorporated into our analysis.

There are also several limitations to the methodology used in this study. One notable challenge involved identifying whether a test could be decomposed into multiple independent properties. We attempted to label tests as decomposable if they included logically independent assertions. However, this classification may be imperfect: assertions that appear independent may be conceptually intertwined. Conversely, even when assertions are dependent, decomposing them could improve clarity or maintainability. Additionally, we observed more complex decomposition opportunities, such as when a test generates multiple SUTs from a list, which could be split into multiple focused tests, or when the assertions themselves can be broken down into smaller assertions.

The open coding process itself may have introduced some ambiguity. Assigning short codes to aspects of tests inevitably involves some personal interpretation. This can lead to under- or over-generalization, especially when concepts overlap. To mitigate this, we revised our annotations iter-

atively across multiple passes, which helped ensure greater consistency.

Furthermore, subjectivity is inherent to qualitative research. Although we did our best to make our procedures transparent and reproducible, different researchers may interpret the same tests differently based on experience, familiarity with the domain, or personal coding practices. While we believe our results are robust, replication by other researchers would be valuable.

The set of repositories we studied was also limited in scope. We selected seven prominent open-source Python projects with a manageable number of PBTs to allow in-depth qualitative analysis. However, the conclusions drawn may not generalize to all Python codebases, particularly those that are smaller, less tested, or have different development practices.

Finally, our workflow – where we first coded all tests on paper and then structured the findings in a spreadsheet – proved to be somewhat inefficient. We found that the most insightful observations often arose not during a subsequent pass of a single repository, but during comparative analysis across repositories. Future studies could streamline the workflow by integrating annotating and digitally structuring into a single step, or by belaying reiteration until more cross-repository context is gathered.

4.5 Responsible Research

We are committed to conducting our research responsibly and ethically. We have taken the following steps to ensure that our research is conducted in a responsible manner.

Licenses. Written software can be the intellectual property of an organization or a person. As such, we have taken care to make sure that all analyzed property-based tests fall under a license that allows us to use and redistribute the source code. All accessed source code was open source and available on GitHub.

Each of the analyzed repositories falls under one of the following licenses:

- Python Software Foundation License Version 2 (`cpython`[13]),
- BSD 3-Clause License (`pandas`[14]),
- Apache License Version 2.0 (`streamlit`[15], `gradio`[16], `jax`[17]),
- The MIT License (`spaCy`[18]), and
- NumPy Copyright license (`numpy`[19]).

All of these licenses allow the source code to be freely distributed, as long as the copyright notices in the code are retained. Since this research paper utilizes code snippets from the repositories, we need to include the appropriate licenses. For this reason, the respective license can be found in a footnote whenever a code example is shown.

To make our research reproducible, the scripts that we have written and used to collect property-based tests from the repositories can be found online[20]. The `README` file also contains a note about where the licenses for the collected property-based tests can be found when the scripts are run.

Generative AI. In this research, we saw Generative AI as a tool that could make us more confident about our findings and in our writing. That is why we used OpenAI's GPT-4 Large Language Model (LLM) in some situations to

- search for definitions (e.g. "What do you call the distance between the two closest rounded values?"),
- explain domain-specific concepts (e.g. "What is 'splash attention' in jax?"),
- find inspiration in writing (e.g. "What do you think of the following first sentence of a research paper: 'Property-based testing (PBT) is an approach to software testing in which the functionality of (a part of) a program is automatically tested with large amounts of input data.'"), or
- help solve technical issues (e.g. "powerpoint add python code with syntax highlighting and line numbers").

We never copied any answers generated by an LLM, or presented its findings as our own. We double-checked all facts that were of importance to our research.

We also used an AI-assisted spelling and grammar checker based on GPT-4 in writing this paper.

All GPT-4 prompts written for this research can be found in Appendix C.

5 Conclusions and Future Work

This section summarizes the key findings of our study, reflects on their implications for the use and development of property-based testing (PBT) in Python, and outlines potential directions for future research.

5.1 Summary of Key Findings

Our study offers a detailed qualitative and quantitative analysis of 87 property-based tests across seven widely used Python repositories that use the Hypothesis framework. The tests varied in complexity, purpose, and design, but several themes emerged.

We observed an askew distribution of property categories: the majority of tests fell into well-understood and relatively easy-to-construct patterns, most notably *round-trip* properties and comparisons to *oracles*. These tests, while often simpler in structure than other tests, seemed capable of capturing meaningful functional properties and identifying subtle bugs in their systems under test.

At the same time, we identified a subset of more complex property-based tests that targeted specific edge cases or tested complicated behavior – often involving custom generators or

[13] https://github.com/python/cpython
[14] https://github.com/pandas-dev/pandas
[15] https://github.com/streamlit/streamlit
[16] https://github.com/gradio-app/gradio
[17] https://github.com/jax-ml/jax
[18] https://github.com/explosion/spaCy
[19] https://github.com/numpy/numpy
[20] https://github.com/DaKoning/hypothesis-dataset

hardware-specific execution paths. These demonstrate the expressive power of PBT when used for testing more than basic invariants. This suggests that there is a spectrum of PBT usage, from lightweight property checks to through example-based specification.

This analysis enables us to address our central research question:

> **How is property-based testing used in real-world Python projects with Hypothesis?**

We found that PBT is primarily used to test functional correctness, particularly in mathematical modules, systems that implement hardware-driven optimizations, and functions with complex input patterns. To achieve this, simple patterns, such as the *roundtrip* property and oracle-based testing were often used, in addition to the custom generation of values.

5.2 Implications

Our findings suggest several opportunities for improving the practical use of property-based testing. Firstly, better tooling and educational resources could help lower the barrier for writing more expressive and domain-specific properties. Developers may benefit from templates that illustrate more complex and effective PBT patterns.

Secondly, our results imply that some unit tests – especially those with many similar test cases – could be replaced or augmented with PBTs. The structure and coverage offered by property-based tests may reveal more edge cases with less code.

Thirdly, our findings indicate that better support and educational resources on custom shrinking may not be necessary. It seems that the default shrinker in Hypothesis is sufficient for most projects.

Finally, our observations on generator usage suggest a promising area of research for test decomposition. In tests where the generator also determines the system under test (e.g., selecting from a list of possible implementations), these could be split into separate tests per SUT, thereby improving modularity and debuggability. Test decomposition may also be relevant in tests where assertions are run based on a conditional, or where assertions form a con- or disjunction that can be broken up into simpler assertions.

5.3 Future Work

Based on this study, several directions for future research are available:

- **Repository Coverage.** Extending the dataset to include a broader and more diverse set of repositories would improve the generalizability of our findings. This would also help identify underrepresented usage patterns.
- **Cross-Framework Comparison.** Future work could systematically compare PBT usage across programming languages and frameworks, especially where different abstractions or features influence developer choices.
- **Automated Analysis.** Using machine learning or natural language processing techniques to automate annotation and classification could enable the study of much larger data sets, which could lead to useful statistical insights.
- **Temporal Analysis.** Studying the evolution of property-based tests over time by analyzing Git commit histories could reveal when and why PBTs are introduced, maintained, or abandoned, offering insights into their long-term value in software projects.
- Decompose when multiple SUT functions

Overall, this study demonstrates that property-based testing, even when used in small amounts, can provide high leverage for asserting program correctness. With further support from the research and software testing community, its adoption could grow significantly in Python and other programming languages.

References

[1] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head, "Property-based testing in practice," in *ICSE '24: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, New York, New York, 2024, pp. 1–13.

[2] A. L. Corgozinho, M. T. Valente, and H. Rocha, "How developers implement property-based tests," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Bogotá, Colombia, 2023, pp. 380–384.

[3] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999, doi: 10.1109/32.799955.

[4] A. Barotsis, "Property-Based Testing in Open-Source Rust Projects: A Case Study of the proptest Crate," Bachelor Thesis, Delft University of Technology, 2025.

[5] M. Derbenwick, "Property-Based Testing in Rust, How is it Used?: A case study of the quickcheck crate used in open source repositories," Bachelor Thesis, Delft University of Technology, 2025.

[6] H. Toth, "Property-Based Testing in the Wild!: Exploring Property-Based Testing in Java: An Analysis of jqwik Usage in Open-Source Repositories," Bachelor Thesis, Delft University of Technology, 2025.

[7] Y. Zhao, "Property-Based Testing in the Wild!: A Study of QuickCheck Usage in Open-Source Haskell Repositories," Bachelor Thesis, Delft University of Technology, 2025.

[8] R. Hoda, *Qualitative Research with Socio-Technical Grounded Theory: A Practical Guide to Qualitative Data Analysis and Theory Development in the Digital World*. Springer, 2024, ch. 10.

[9] M. Derbenwick, H. Toth, D. de Koning, A. Barotsis, Y. Zhao, A. Costea, and S. Juhošová, "Property-based testing in the wild!" 4TU.ResearchData, 2025, doi: 10.4121/368f63ab-10fc-4603-a15a-bde25e72e778.

[10] D. R. MacIver and A. F. Donaldson, "Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer," in *34th European Conference on Object-Oriented Programming (ECOOP*

2020), ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, pp. 13:1–13:27, doi: 10.4230/LIPIcs.ECOOP.2020.13.

A Analyzed Characteristics

We analyzed the collected property-based tests using a set of characteristics that we derived from the tests themselves. The full list of characteristics is shown in Figure 1. Each characteristic is explained in the table, and we used these characteristics to summarize our findings about the property-based tests.

Table 1: Full list of characteristics used to summarize findings about property-based tests.

characteristic	explanation
`link`	URL to the first line of the PBT in the analyzed commit of the repository.
`repo`	Name of the repository where the PBT is located.
`name`	Name of the PBT.
`summary`	Description of what the PBT tests for.
`min_assertions`	Minimum number of assertions ran if test completes.
`max_assertions`	Maximum number of assertions ran if test completes.
`amt_assrts_dpndt_inp_size`	Whether the amount of assertions is dependent on the size of the input.
`assertions_independent`	Whether all assertions are independent.
`can_decompose_assertions`	Whether any pair of assertions is independent.
`amt_sut_calls`	Maximum number of calls made to the system under test.
`is_local`	Whether the system under test is local (as opposed to a library).
`test_level`	Whether the PBT tests functionality, integration, or environment.
`is_nested`	Whether the assertions are in a different function than the test.
`f_or_nf`	Whether the tested property is functional, non-functional, or both.
`tests_state_modification`	Whether the tested property involves modification of state.
`category`	The category that the tested property falls under.
`has_assumptions`	Whether the test uses Hypothesis assumptions.
`generated_input_type`	A list of inputs that are generated by Hypothesis.
`is_input_filtered`	Whether any generated input is filtered (i.e. the domain of the input is restricted by some rule other than its type).
`is_input_sampled`	Whether any generated input is sampled from a list of possibilities.
`uses_custom_generator`	Whether the PBT uses a custom generator to create any of the inputs, a custom generator is considered to be a generator that is decorated with `@hypothesis.composite` or makes use of non-Hypothesis functions to process the generated values.
`uses_dynamic_generation`	Whether inputs are generated using Hypothesis from within the test, as opposed to from decorator
`uses_non_hypothesis_generation`	Whether some other module is used to generate random data.
`amt_example_inputs`	How many non-generated test cases are specified (if any).
`is_parameterized`	Whether some inputs to the test come from a parameterization decorator.
`generated_input_used_directly`	Whether the input is passed directly onto the SUT/environment (if not, it is processed first).
`uses_custom_shrinker`	Whether the PBT makes use of a custom shrinker.
`asserts_exceptions`	Whether the test asserts that a certain exception is thrown by the SUT.
`can_throw_exception`	Whether the test includes an exception-throwing clause.
`uses_assert_close`	Whether there is an assertion with specified tolerances (i.e. two values are asserted to be close, but not necessarily equal).
`uses_custom_assertion`	Whether the test uses an assertion function that does not originate in a library.
`uses_hypothesis_note`	Whether the PBT uses Hypothesis note to display a message when the test fails.
`test_precondition`	What condition must be met for the PBT to be run (if any).

explicitly_functionally_system_dependent	Whether the results of the test are explicitly shown to be affected by hardware.
deterministic_generation	Whether data are deterministically generated by Hypothesis.
member_test	Whether the PBT is a member of a class.
generated_SUT	Whether the system under test is generated by Hypothesis.
xfail	In which cases the test is expected to fail (if any).
number_of_testcases	Maximum number of test cases that may be generated by Hypothesis.
deadline (ms)	Maximum time in milliseconds that Hypothesis is allowed to run the test.
skip_test	In which cases the test is skipped (if any).
has_test_helper	Whether the he test uses another function that was defined for testing purposes (but not a custom assertions or custom generator).

B Property-Based Test Categories

In this appendix, we present the categories of property-based tests (PBTs) that we identified in our analysis. These categories are based on the characteristics of the tests and their intended purposes.

Some categories are taken from a blog post that is cited in the Hypothesis documentation[21] (marked with *), while others are devised by analyzing the tests and comparing them. Since not all categories described in the blog post are widely used [2], we chose to only include the four categories that could be identified in the dataset.

The categories are explained in Table 2. Each category is designed to capture a specific aspect of the property-based tests, such as their behavior under certain conditions or their relationship to the system being tested. The count column indicates how many tests fall into each category. It is possible for one property-based test to belong to multiple categories, which is why the count column in the table below does not add up to the total number of tests (87).

Table 2: Full list of categories used to classify property-based tests based on their properties.

name	explanation	count	frequency
differentPathsSameDestination*	Running operations in different orders yields the same result.	3	3.4%
idempotence*	The result of an operation does not change if it is ran multiple times.	5	5.7%
roundtrip*	The result of applying an operation followed by its inverse to some data, results in the original data.	15	17.2%
testOracle*	A function defined for testing purposes provides the correct result for the system under test to give.	13	14.9%
transformationInvariance*	Some characteristic of the data does not change when the data is tranformed.	1	1.1%
additivity	The sum of the results of an operation applied to two values is equal to the result of that operation applied to the sum of the values, i.e. $op(a) + op(b) = op(a + b)$.	2	2.3%
monotonicity	The results of the system under test do not change direction as its argument solely increases or solely decreases, i.e. the system under test is a non-decreasing or non-increasing function.	2	2.3%
symmetry	Applying a function to some arguments has the same result as applying it to the symmetric counterpart of those arguments.	1	1.1%
referenceConsistency	A reference implementation for the system under test provides the correct result for the system under test to give. The system under test is some optimization for the reference function and requires specialized hardware or some extra setup.	9	10.3%
systemEquivalence	An alternative implementation of the system under test provides the correct result for the system under test to give.	18	20.7%
constructionIntegrity	The construction of an object based on some data results in an object that is consistent with the provided data.	13	14.9%
equivalenceExtensionality	Two objects are equal if and only if they have a specific characteristic in common, i.e. $\text{obj}_1 = \text{obj}_2 \iff \text{obj}_1.x = \text{obj}_2.x$.	2	2.3%
argumentIdempotence	Applying a function to a value has the same result as applying the function to that value as multiple arguments, i.e. $f(x) = f(x, x) = f(x, x, x) = ...$	2	2.3%
outputJustification	The output of the system under test is consistent with the generated justification.	1	1.1%
postCondition	The output of the system under test always has a certain characteristic.	15	17.2 %
exceptionGuarantee	The system under test always throws an exception in the provided circumstances (input or environment).	1	1.1%
noException	The system under test never throws an exception in the provided circumstances (input or environment).	3	3.4%
testByRunning	The system under test has its own direct means of passing or failing a test (e.g. through exceptions or assertions).	4	4.6%
objectCached	The system under test caches equal objects, such they have the same memory location.	7	8.0%
recursionPerformance	The system under test provides a satisfactory output within a certain recursion depth.	2	2.3%

[21] https://fsharpforfunandprofit.com/posts/property-based-testing-2/

C Generative AI Prompts

In this appendix, we present every prompt that we wrote to a Large Language Model (LLM). The LLM that we used was OpenAI's GPT-4. In favor of brevity, we have left out some code inputs and outputs (replaced with description between <brackets>) and we have removed line breaks.

- "hypothesis strategies sampled from vs one of"
- "Is there a name for functions that are inside a class (as opposed to not being part of a class)?"
- "What's going wrong in this overleaf table? <latex code of table>"
- "overleaf does not know what \keepXColumns is"
- "I would like to rewrite the sentence below without using the term "Swiss army knife", such that it sounds like it is the perfect tool for software testing. "Although this sounds like the Swiss army knife of software testing, practice shows that property-based testing is not widely adopted within the programming community.""
- "What is the best tool for qualitative coding of open source software projects?"
- "I'm doing research into how the property-based testing is used in practice. In particular, I will be looking into open-source Python projects that use the Hypothesis framework. I want to use open coding for qualitatively analyzing the .py files. Do you know of any software/tools that I can use to help me in this endeavor?"
- "While running an Ubuntu executable, I'm getting the following error. Can you help me solve it? [13617] Error loading Python lib '/tmp/_MEI7wkn5P/libpython3.12.so.1.0': dlopen: /lib/x86_64-linux-gnu/libm.so.6: version GLIBC_2.38' not found (required by /tmp/_MEI7wkn5P/libpython3.12.so.1.0)"
- "For my paper about PBT in Python using Hypothesis, I've written the following introduction. What do you think about it? <first draft of Introduction section>"
- "If sampling is defined as selecting n possibly equal elements from a finite domain, what would be a term for selecting n distinct elements from an infinite domain?"
- "Can you translate the following scopus query into ieee xplore? (TITLE-ABS-KEY ("property-based test*") OR TITLE-ABS-KEY ("property based test*") OR TITLE-ABS-KEY ("property-test*") OR TITLE-ABS-KEY ("property test*")) AND (TITLE-ABS-KEY ("in practice") OR TITLE-ABS-KEY ("real*world") OR TITLE-ABS-KEY ("in the wild"))"
- "latexpdf underscore rendered as ¨"
- "powerpoint text add line numbers"
- "powerpoint add python code with syntax highlighting and line numbers"
- "What do you call the rule that tick(n) + tick(m) == tick(n+m)"
- "What do you call the distance between the two closest rounded values? For example, this would be 100 if you round to the nearest 100, or it would be 1 if you round to the nearest integer"
- "Is there a better way to do the following in overleaf: <latex multicols list>"
- "What does this do in python? scalar1 = arr1[()]"
- "In Excel, I have a column of lists of categories, and I want to find how many of each category I have. How do I do this?"
- "overleaf escape @ sign in texttt"