A Python Decorator for Programmatic and Deterministic Unit Test Code Generation

James Hatfield
Glen Burnie, MD, USA
jahatfi@gmail.com

***Abstract*—Automated software testing is an area of active research, particularly automated generation of unit tests. Current and previous work such as generative artificial intelligence, search-based heuristics, and randomization often lead to incomplete coverage and non-deterministic results. In addition, some of the previous work lacks a natural way for a human expert to provide input to the automated test generator outside the source code itself. The author presents a Python decorator to generate deterministic Python unit tests from existing higher level tests via metaprogramming. This Python decorator hooks function calls in order to record the metadata (input, output, relevant global variables, and code coverage) of each hooked function. The recorded metadata is then used to automatically generate unit tests for each such function. The author's initial testing confirmed that it can be a force multiplier for the developer, as one passing integration test of the overall system can produce many unit tests. This work leverages expert human insight by deriving unit tests from as little as one human-designed test. Further, many test cases may be generated for each test. Python programmers can save significant time and effort using the generated test cases as boilerplate if more test cases are required to increase coverage.**

***Index Terms*—Python, Automated Test Generation, Metaprogramming**

I. Introduction

A. Software Testing Paradigms

1) Test Driven Development: Advocates of Test Driven Development (TDD) such a Harry Percival [1] argue that software developers should follow the TDD process in which they develop software by first writing tests, then the bare minimum amount of code to make those tests pass, before beginning the cycle anew with more tests. On page 22 [1] Percival advocates both functional and unit tests, making the following distinction: "functional tests test the application from the outside, from the point of view of the user. Unit tests test the application from the inside, from the point of view of the programmer." Later, on page 470 [1] Percival defines an "integration test" as one that depends on and interacts with some external system. This paper will use both terms interchangably, in addition to the term "ad hoc test" in reference to informal tests used by the developer. The key asssumption for this paper is that there exists at least one passing test above the unit level that calls at least one internal function. This paper will describe how the author's work takes advantage of the comprehensive nature of higher-level tests in which multiple functions are executed, potentially with various inputs. Previous academic research [2], [3] observed that industry TDD adoption is lower than desired, reporting longer development cycles, skill issues, and legacy code as a few root factors, among others.

2) Non-TDD Testing Paradigms: In contrast to the TDD methodology in which only the developers write the tests [4], others [5]–[7] propose a more customer-focused testing approaches such as Behavior-Driven Development (BDD) or Acceptance Test Driven Development (ATDD) focused on the software's behavior from the user's perspective. In these paradigms, non-developers play a role in creating the tests, such as a business analyst (in BDD) [8] or a cross-functional team of developers, business analysts, and testers [9]. These advocates argue that such feature-first approaches are optimal for accelerating development and reducing a product's time to market due to the time-consuming nature of manually developing unit tests [6], [10]. Though Brown *et. al.* acknowledge that integration and unit testing may go hand-in-hand, they ultimately argue in favor of the former before the latter [5]. Although implied by Brown *et. al.*, Shahabuddin *et. al.* [6] go so far as to explicitly argue that unit testing can be performed *after* initial product delivery to the customer. This research corresponds with the author's personal observations that immature organizations (or solo developers on small or personal projects) often don't adhere to the TDD approach, and end up writing the tests after the development of the code or fail to write tests at all. In such cases, the hobbyist may focus primarily on function or integration tests and opt not to develop unit tests, perhaps because the pros of unit tests do not outweigh the cons compared to a higher level test. Table I briefly summarizes a few pros and cons of high level (e.g. functional) compared to units tests.

TABLE I: Pros and Cons of High Level and Unit Tests

	High Level Test(s)	**Unit Test(s)**
Pros	• More relevant for the customer • Efficiently tests modules together • Less dependent on the internal unit implementations	• Verifies individual components • Efficient when testing single components
Cons	• Waste of time when testing small unit changes • May not accurately identify root cause of failure	• Time consuming to develop • Easily broken by refactoring code

Despite the time-consuming nature of developing unit tests

manually, developers of all paradigms recognize value in *having* unit tests. By definition, any reduction in the cost of unit test creation drives up their return on investment.

3) Test Coverage: The term "coverage" refers to the measure of the completeness of a test. Coverage can be measured in various ways, two prominent metrics are branch coverage (what percent of if/else branches executed) and line coverage (what percent of lines in the function executed during the test) [11]. The *coverage* Python module can be used to gather this information directly; in fact, it is often paired with *pytest* to generate coverage tests for existing tests. Without loss of generality, this paper will focus use the line coverage metric.

B. Previous work on automated test creation

Given the cost of creating valuable tests, the body of academic work on generating them automatically via metaprogramming is extensive. Classic algorithms like search and randomization feature prominently in earlier work [12], with generative artificial intelligence (AI) significantly increasing in popularity most recently [10], [11], [13], [14]. These AI methods generate tests based on a variety of inputs, typically the code itself and some other input such as human prompts [15], natural language requirements [11], or entire projects [16].

While the results of such AI-generated tests are promising, their coverage is not perfect [10] and still require significant developer review and correction [17].

Lemieux *et. al.* propose a hybrid method, *CodaMosa*, that combines search and mutation (classical approaches) with generative AI [18]. Their work is impressive but still relies on heuristic techniques that the work here does not. This author is also unclear what, if any, specialized hardware was required to use *CodaMosa* given it's reliance on a large language model (LLM).

C. Initial motivation

The author began exploring this concept of automated test case generation while refactoring a personal project, a static analyzer he'd written for C source code. As the author did this for fun and personal education, he had not written any unit tests. Instead, the author carefully developed a sample input that caused large portions of the code to execute (e.g. created high code coverage). This guided development until the code eventually produced a "good" output, i.e. a functional test.

Unit tests would have greatly eased the refactoring process and the author hypothesized that the metaprogramming abilities of Python might empower creation of unit tests from that sole existing test. This paper describes the author's effort to prove this hypothesis.

D. Organization of this paper

Section 2 of this paper describes the overall approach to building a Python decorator to generate unit tests from existing tests. The subsequent section enumerates both technical and philosophical assumptions made by the author, as well as alternative approaches when those assumptions may not hold. Section 4 describes the three different approaches taken by the author to evaluate the generated tests, followed by a section dedicated to further discussion of this work compared and contrasted to more related work. A short conclusion follows, capped off by Acknowledgements.

II. Assumptions

An initial assumption of the author's hypothesis is that all code executing under the functional test is mostly correct, as the inputs and outputs from each function called during the functional test would become data for the unit tests.

That said, the author acknowledges that software engineers may not wish to assume that an apparently working functional test indicates accuracy of all internally executed components. Regardless, the creation of such unit tests would still provide a syntactically accurate unit test file for the software engineer to start from, rather than create each unit test from scratch.

In addition to saving time from creating the test boilerplate, the true expense saved is that of manually defining the desired inputs and correct outputs. This author believes there is value in this automated approach even if the generated tests must still be verified for accuracy, as in the author's experience it was faster in some cases to manually inspect the resulting unit test code for accuracy than develop and verify unit tests from scratch.

Another assumption required for successful execution is that the *repr()* method of each object generates valid Python code that can be used to re-create that object, or that such a function can be developed and used to temporarily overwrite a *repr* method not meeting this requirement. The author demonstrates this in the repository code by overwriting the *repr* method of the Pandas DataFrame class as shown in Listing 1.

```
import pandas as pd
def _pandas_df_repr(df: pd.DataFrame)->str:
    '''
    The 'repr' method for Pandas DataFrames does
    not work the same as 'repr' for built-in types.
    StackOverflow user Silveri's answer at
    https://stackoverflow.com/questions/67845199.
    provides a solution. The code overwrites the
    native Pandas DataFrame __repr__  method
    directly,
    indirectly affecting the repr() method as well.

    Note: For most use cases overwriting a class's
    method like this is considered bad practice.
    '''
    return f"DataFrame.from_dict({df.to_dict()})"

pd.DataFrame.__repr__ = _pandas_df_repr
```

Listing 1: Overwriting incompatible *repr* method

III. Recording the Execution of a Python Function

A. Defining the required components to create a unit test

The author set out to record the execution of Python functions in such a way as to enable exact reproduction of that function call in a standalone unit test. The following components must be recorded in order to do so:

1) The function itself and it's arguments, including *kwargs*
2) Any relevant global state (e.g. variables)
3) Exceptions raised
4) Test coverage
5) Files/databases read from and/or written to
6) Data sent/received via a socket

The last two are left for future work, but this paper demonstrates how to capture the first five. The subsections that follow discuss exactly how to access or determine this info and cache it for subsequent creation of unit tests with this information.

B. Accessing the function and its arguments

Python enables trivial access to a function and its arguments via the concept of decorator functions. Not to be confused with the decorator pattern, a Python decorator is a function that calls another, thereby permitting the developer to place new code before and/or after calling the original "decorated" function *f*. The decorator function has full access to both the decorated (or "wrapped" function) *f* as well as all the arguments passed to *f*, both *args* and *kwargs*. In other languages this kind of wrapping is often referred to as "function hooking" or "function call interception" [19]. Any number of decorators can be applied to a function in Python, creating a figurative Russian nesting doll of functions calling functions, each with the ability to access the arguments of the function it wraps and modify the return value before return. A Python decorator is applied with the '@'symbol as shown on line 20 of Listing 2[1]:

```
1  import copy
2  from typing import Callable
3  from ...src.unit_test_generator import
       unit_test_generator_decorator
4
5  def outermost_decorator_with_parameters(my_int: int)
       :
6      def my_outer_decorator_with_args(f:Callable):
7          def innermost_decorator(*args, **kwargs):
8              print(f"Before call to '{f.__name__}'")
9              print(f"{args=}; {my_int=}")
10             args_copy = list(copy.deepcopy(args))
11             args_copy[0] = args_copy[0] + 1
12             args = tuple(args_copy)
13             print(f"Modified args: {args=}")
14             result = f(*args, **kwargs)
15             print(f"In decorator after call: {result
       =}")
16             return result
17         return innermost_decorator
18     return my_outer_decorator_with_args
19
20 @outermost_decorator_with_parameters(5)
21 def add_ints(a: int, b:int)->int:
22     return a+b+c[0]
23
24 c = [5] # global variable
25 x = 2
26 y = 7
27 print(f"Adding {x},{y},{c}...")
28 final_result = add_ints(x, y)
29 print(f"{final_result=}")
```

[1]The author optimized this and other code snippets for display in this paper; therefore they may differ slightly from their original source found in the associated repository

Listing 2: decorator.py: A sample decorator that takes one argument.

Running the code in Listing 2 yields the output shown in Listing 3. Note the apparently erroneous math (line 1,6 of Listing 3), as the *add_ints* function is unaware that its first argument was modified by the inner_most_decorator.

```
1 Adding 2,7,5...
2 Before call to 'add_ints'
3 args=(2, 7); my_int=5
4 Modified args: args=(3, 7)
5 After call: result=15
6 The result is 15
```

Listing 3: Output of decorator.py

As shown in Listing 3, not only can the Python developer access the function via the variable *f* (e.g. line 14), the developer also has full access to the variables passed to *f*, and can make arbitrary changes to the arguments in a transparent way (line 11), i.e. the calling function would never know the arguments were modified before being passed to the callee function.

In addition to access to the function and its arguments, a decorator provides the power to insert code immediately before and after the function (lines 8-13, 15, respectively), including leveraging arbitrary arguments (e.g. *my_int*) passed to the decorator itself.

A decorator like the one on line 5 of Listing 2 permits the passing of separate arguments to the decorator itself. If no extraneous argument is needed, that *outermost* decorator is unnecessary.

The author uses such a decorator (specifically named *unit_test_generator_decorator* in the referenced repository) to take a "before" and "after" snapshot of the arguments before and after the function is called.

C. Accessing relevant global state

In addition to the arguments passed directly to the function, any relevant global state must also be captured. "Relevant" here refers only to those global values read from and/or written to by the function. The author's code focuses on variables (e.g. the int *c* in the example above), detecting import modules such as *re, os, etc* in a separate parsing step (see) Access to the global values is non-trivial compared to accessing the function and its arguments, but still possible by first using the *dis* module to disassemble the decorated function ato determine which global variables are ever accessed by the function. This disassembly is only required on initial execution of the decoratee as subsequent executions, if any, benefit from cached results of the disassembly. Disassembly of *add_ints* function during execution is shown in Listing 4:

```
1  29           0 RESUME                   0
2
3  31           2 LOAD_FAST                0 (a)
4               4 LOAD_FAST                1 (b)
5               6 BINARY_OP                0 (+)
6              10 LOAD_GLOBAL              0 (c)
```

```
             22 LOAD_CONST               1 (0)
             24 BINARY_SUBSCR
             34 BINARY_OP                0 (+)
             38 RETURN_VALUE
```

Listing 4: Result of Disassembling *add_ints*

Note the LOAD_GLOBAL command on line 6 of Listing 4 to load the value of *c*. Such global names *read from* are saved for later reference. Likewise, names and values *written to* via the STORE_GLOBAL commands are also cached for later use. Of note, the disassembly shown above differs from stand-alone disassembly of the same function in the Python interpreter (compare the listing above to the *examples2/disassemble_decorator_with_decorators.txt* file in the accompanying repository.) During actual execution the author's code disassembles only the *add_ints* function, after the decorators have already executed, leaving only the original *add_ints* function for disassembly. In contrast, disassembling the function in the static, non-executing context of the Python interpreter reveals the code of all applied decorators. Values read from the global state must be recorded in order to monkeypatch them in the resulting unit test. The decorated function must also record modified global values in order to assert that the state was correctly changed by the function.

D. Detecting exceptions

Detecting exceptions is easiest of the three information capture steps. Any exception can be detected with the code shown in Listing 5.

```
  try:
    # call the decorated function, e.g.
    f(args, kwargs)
  exception Exception as e:
    # Save the exception type and exception message

```

Listing 5: Catching and recording exceptions

E. Determining Test Coverage

The Python *coverage* module provides support for capturing line (and branch)-based coverage as shown in Listing 6 [20].

```
import coverage
cov = coverage.Coverage(None)
with cov.collect():
    try:
        if kwargs:
            this_coverage_info.kwargs = kwargs
            start_time = time.perf_counter()
            result = func(*args, **kwargs)
        else:
            start_time = time.perf_counter()
            result = func(*args)

        logger.debug("No exception :)")
    # There's no way to know ahead of time
    # what kind of exception func might raise,
    # so catch any of them.
    # pylint: disable-next=broad-exception-caught
    except Exception as e:
        caught_exception = e
    finally:
        end_time = time.perf_counter()
```

Listing 6: Calling a function and capturing the test coverage with the *coverage* module.

The documented approach to using the *coverage* module saves results to a persistent database on the filesystem as specified with the *data_file* option. However, due to the already high overhead of the author's work, he opted to specify *None* for this value to use the in-memory option, thereby eliminating such file I/O time [20].

F. Summary of the approach

The author uses all the methods discussed above to take a "before" and "after" snapshot of the arguments and relevant global state of each execution.

Note that the state of mutable *arguments* must also be captured *after* the function executes as called functions may change mutable arguments that persist upon return to the caller. The return value or exception type and exception message are also captured, in addition to line test coverage. For each execution of a given function, an instance of the *CoverageInfo* class is created and the fields populated, see Listing 7.

```
"""See accompanying repo for full code"""
from dataclasses import field
from dataclasses import dataclass
from typing import Any

@dataclass
# pylint: disable-next=too-many-instance-attributes
class CoverageInfo:
    """
    Holds all data gathered from single recording
    of a function or class method execution
    """
    args_before: list[str] = field(default_factory=
    list)
    args_after: dict[str, Any] = field(
    default_factory=dict)
    kwargs: dict[str, Any] = field(default_factory=
    dict)
    globals_before: dict[str, Any] = field(
    default_factory=dict)
    globals_after: dict[str, Any] = field(
    default_factory=dict)
    expected_result: str  = ""
    expected_type: str = ""
    coverage: list[int] = field(default_factory=list
    )
    exception_type: str = ""
    exception_message: str = ""
    constructor: str = ""
    cost:float = 0.0
    testable:bool = True
```

Listing 7: The CoverageInfo class caches all metadata associated with a single execution.

In addition, one instance of the FunctionMetaData classes shown in Listing 8 is populated for each decorated function.

```
"""See accompanying repo for full code"""
from pathlib import Path
from typing import Optional, List

class FunctionMetaData(Jsonable):
    """
```

```
    Class to track metadata when testing
    functions and methods

    "Jsonable" is another class defined elsewhere
    that permits easy creation of the string
    representation of any class.
    """
    # pylint: disable-next=too-many-arguments
    def __init__(
        self,
        name:str,
        parameter_names:List[str],
        is_method:bool,
        source_file:Path,
        lines:Optional[List[int]] = None,
        non_code_lines:Optional[List[int]] = None,
        global_vars_read_from:Optional[set] = None,
        global_vars_written_to:Optional[set] = None,
        coverage_io:Optional[dict[str, CoverageInfo
    ]] = None,
        coverage_percentage:float=0.0,
        types_in_use:Optional[set] = None,
        unified_test_coverage:Optional[set] = None,
        needs_pytest:bool = False,
        exceptions_raised:Optional[set] = None,
        callable_files: Optional[dict[str, str]] =
    None
        ):
        pass
        # See accompanying repo for full code
```

Listing 8: The FunctionMetaData class caches all metadata associated with a single function, to include all associated CoverageInfo classes (line 25)

IV. Programmatically Tuning the Decorator

A. Selectively bypassing the decorator

As the reader may imagine, the decorator described above adds significant overhead to the execution time of the overall test, as the function call is intercepted, arguments inspected and copied, etc. To mitigate this overhead the author provides a variety of methods to "bypass" the decorator, i.e. calling the decorated function and simply returning the result without further action.

The first and perhaps most obvious approach is to set a coverage threshold such that the decorator is effectively disabled once a desired level of coverage (e.g. 80%) is achieved.

One could also define a specific number of executions to capture, after which the decorator would be disabled. The author supports either option by setting either the *percent_coverage* or *sample_count* option to the decorator to the desired value. Setting *percent_coverage* to a value greater than 100 will capture all executions of the decorated function. Listing 9 shows this code.

```
if percent_coverage and \
   percent_coverage <= this_metadata.
    get_percent_covered():
   # Desired percent coverage already achieved: skip
   # logging statements here
   # Since this decorator is effectively nullified
    now,
   # do NOT try/catch/raise any exceptions.
   return func(*args, **kwargs)

if sample_count and \
   sample_count <= len(this_metadata.coverage_io):
   # Desired number of samples already achieved:
    skip
   # logging statements here
   # Since this decorator is effectively nullified
    now,
   # do NOT try/catch/raise any exceptions.
   return func(*args, **kwargs)
```

Listing 9: Programmatically "bypassing" the decorator by immediately returning the function results when desired coverage met

The careful reader may wonder if the *sample_count* option may result in duplicate tests cases, e.g. five test cases are captured but three are duplicates, yielding only three unique test cases. This would in fact be the case, however, the author addressed this by hashing the inputs (global values, args and kwargs) and caching the hash in a set named *hashed_inputs*. If a function has already been called with the exact same inputs, the decorator immediately returns the result in the same fashion as shown above. Thus, all test cases ultimately generated are unique. This represents the last mitigation currently in the code. In future work the author hopes to return the cached result/raise the same exception of this execution to save even more overhead.

It is also important to note that the decorator need only be applied until the test cases are generated and the developer is satisfied that the created unit tests will suffice. At that point the decorator can be completely removed from the functional or integration test[2], thereby restoring the runtime of the original test back to its original, faster runtime.

B. Selectively keeping specific test cases

Ideally, to maximize coverage, the same function will be executed multiple times during a functional test or similarly high level test. That said, the tester may wish to preserve a minimal number of test cases to avoid too much redundancy. One such approach is to drop the current execution record if the current execution didn't add any new coverage. Conversely, one can drop previous records if the current coverage supersedes previous coverage. There are a few ways to minimize duplicate coverage as explored below, but the reader should bear in mind:

1) that *fully* orthogonal test cases, as measured by coverage, are not feasible (all test cases will cover the initial lines until the first conditional statement)
2) test cases with redundant coverage can still hold value
3) some code such as function calls and regular expressions may have one line in source code, but multiple branches in the underlying library or machine code.

Nevertheless, the value added by redundant tests often plateaus [18], hence the following sections describe how to reduce such redundancy. The first section describes the technique employed by the author to balance memory usage and performance but with intermediate complexity. The paragraph that follows describes a much simpler and faster

[2]Equivalently, *sample_count* and *percent_coverage* can both be set to 0.

approach but offers poorer performance in that it rarely selects optimal test cases. For the reader desiring the truly ideal solution, i.e. keep the fewest test cases that achieve maximum coverage, a note on the proper algorithm to pursue such a goal completes this section. This method was not selected due to its higher memory requirements and higher complexity of implementation.

1) Selected Approach: Individual Subsets: One can maintain coverage information for previous executions individually and check the current set of covered lines against all previous coverage sets. If the current coverage is a subset of all existing records, it is discarded. Else, all previous records whose coverage is a proper subset of the current record are deleted (or *pruned*) and the current record kept. This is more effort than the next approach but retains fewer records with the same total coverage. A simple sample of the individual subset approach is shown in Table 2.

Execution #	Coverage (line #s)	Action
1	1-5	Keep this record, since it is the first one.
2	1-7	Keep this and drop record #1, as that coverage is only a subset of this coverage.
3	1-2	Don't keep this record; it is a subset of an existing record (#2)

TABLE II: Hypothetical scenario #1 demonstrating record pruning by checking for subsets

Unfortunately, this subset approach still leaves room for redundant tests. Consider a new scenario:

Execution #	Coverage (line #s)	Action
1	1-5	Keep this record, since it is the first one.
2	6-10	Keep this one, it covers new code.
3	5-6	Keep this record, it's not a subset of any previous record.

TABLE III: Hypothetical scenario #2 demonstrating record pruning by checking for subsets

Since the subset approach does not aggregate all the coverage into a single unified set as in the next approach, the record for execution #3 would be kept, but this would be redundant, as executions 1 and 2 already executed lines 5 and 6, respectively. The author implemented this approach such that setting the *keep_subsets* options to True would not drop redundant records in case the user desired to keep test cases that produce redundant coverage. The author did maintain a unified set of coverage as discussed next, but it played no role here.

2) Check for new coverage via unified set: Another approach is to maintain a unified set U_t that is the union of all lines covered by all recorded tests from S_0 to S_n:

$$U_t = S_0 \cup S_1 ... \cup ... S_n$$

then only keeping the current record S_i if any currently covered lines are not in the unified set:

$$S_i \not\subset U_t$$

then updating the unified set with the new coverage information like so:

$$U_t \Leftarrow U_t + S_i$$

With this approach the individual coverage information S_i is not maintained (U_t is updated, then S_i is discarded), making it impossible to drop superseded coverage records over time. 100% coverage would mean that:

$$U_f = U_t$$

This approach is the simplest of the three described here but suffers from a lack of granularity: it's impossible to remove old records if new records have better coverage, as specific coverage per test is not recorded.

3) Optimal Result by Solving the Minimum Set Cover Problem: If the developer desires the best solution, e.g. to keep the bare minimum number of test cases, all records must be cached initially. From there, the tester must solve the Minimum Set Cover Problem [21] to select the minimum test cases. This classic NP-Hard computer science problem involves finding the smallest number of sets whose union is equal to or greater than ("covers") some universe U of elements. In this context, the coverage (set of line number executed) of each test is one set S_i, and the set of all line numbers of a function is the universe U_f to cover. As noted, this requires caching the unique coverage of each test, which may be prohibitive from a memory standpoint for large applications. If memory is not a concern, this method may reduce the runtime compared to the alternate approaches described above. [3] As the pruning only takes place after the tests are complete, the tests are not slowed as much as the previous methods that prune during the tests. That said, the sole pruning operation will take comparatively longer as all the records must be analyzed all at once. The reader should be aware that due to the NP-Hard nature of this problem, typical solutions are heuristic-based.

The author did not implement the Minimum Set Cover problem or its weighted variants, but encourages the interested reader to do so. The author records the execution time of each execution as the 'cost'. This value could be applied in a weighted variant of the Minimum Set Cover problem, i.e. select the "best" test cases such that coverage is maximized and run time is minimized.

V. Generating the Unit Tests via Metaprogramming

After capturing the function metadata as documented in Listings 7 and 8, the next step was to programmatically generate the unit tests with all their captured test cases and place them

[3] Validating this hypothesis could be an area for future work.

in a file named *test_$function_name.py* via a call to *generate_all_tests_and_metadata* at the end of the functional test. Calling this function eventually executes *auto_generate_tests* that uses metaprogramming to:

1) Build a list of imports
2) Define global variables constant across all test cases
3) Convert values to valid (i.e. canonical) Python code
4) Build a comment specifying the level of coverage achieved[4]
5) Collect parameters to create a parameterized pytest test
6) Monkeypatch all non-constant relevant globals read from
7) Assert result is correct
8) Assert modified globals and mutable arguments are correct
9) Assert expected exceptions and their messages are correct
10) Write all of the above to a syntactically correct Python file
11) Use subprocess to lint and format the result, e.g. with *black and/or ruff*

To aid the developer's troubleshooting efforts, the contents of each FunctionMetaData class are also dumped to their own .json file, one per decorated function (hence the *and_metadata* suffix noted above) [5]. This file permits the developer to easily inspect the inputs, outputs, and other data associated with the recording without the distraction of the test code.

Thus, if the function *foo* is decorated with *unit_test_decorator*, the generated unit test code can be found in *test_foo.py*, and the metadata can be found in a JSON file name named *foo.json*

In addition to all of the above, significant development efforts went into serializing and deserializing Python code to and from strings. Though certainly not a panacea, the work here advances that done by Lemueux *et. al.* [18] not only in terms of automated testing but also in terms of serialization capabilities.

VI. Tester, Test Thyself

The decorator was evaluated informally with three separate approaches. The author began by crafting simple but demonstrative test cases (found in the *tests/* folder in the associated repository.) Next, the author applied the concept of programmatically generated unit tests to the code itself with positive results. With two key logical checks it was possible to apply the decorator to its very own support code. Those checks were to:

1) Immediately return if the pytest module was loaded, indicating a unit (not integration) test is currently being executed
2) Immediately return if the call stack contained a loop, e.g. A→B→A

These checks are shown in Listing 10.

```
if "pytest" in sys.modules:
    logger.debug("pytest is loaded; don't decorate
    when under a test")
    result = func(*args, **kwargs)
    return result

function_calls: dict[str, int] = defaultdict(int)

# The code blocks below (before the call to
    do_the_decorator_thing)
# prevent application of this decorator in cyclical
    calls, e.g.
# A -> B -> A # Does not apply to decorator in 2nd
    call to A
for f in inspect.stack()[::-1]:
    this_frame = inspect.getframeinfo(f[0])
    function_calls[this_frame.function] += 1

max_func_call_vals = max(function_calls.values())
if function_name not in
    recursion_depth_per_decoratee:
    recursion_depth_per_decoratee[function_name] =
    max_func_call_vals

elif 1 < max_func_call_vals >=
    recursion_depth_per_decoratee[function_name]:
    try:
        return func(*args, **kwargs)
    except Exception as e:
        logger.critical(e)
        raise e

# At this point we know we aren't in a function call
    graph cycle.
```

Listing 10: limit_recursion.py

Careful examination of the call stack was required to limit recursive decorators to just one level. Finally, as previously mentioned, the author applied his work to another project involving parsing C code. These three approaches are described detail below:

A. Manually Created Tests

The author created a variety of tests to ensure the unit test generation code functioned properly. The Procedural Division example is explained in depth below and the others are briefly summarized.

The *tests/example_procedural_division* tests procedural (as opposed to object-oriented or functional) code that:

1) Returned a string given two ints
2) Wrote to a global variable
3) Raised two different types of exceptions

The author wrote a *divide_ints* function as shown in Listing 11.

```
def divide_ints(a: int, b: int):
    """
    Divide two numbers, raising TypeError or
    ValueError
    if not ints or denominator is 0, respectively.
    Set a global_error code on error, else return a
    string
    representing the operation.
    """
    global error_code # pylint: disable=global-
    statement
    logger.info("error_code=%d", error_code)
    if not isinstance(a, int):
        error_code = -1
        raise TypeError(f"TypeError: {a=} not an int
    !")
    if not isinstance(b, int):
```

[4]e.g. line 17 in Listing 13

[5]This includes "untestable" test cases, i.e. those that could not be represented as canonical strings but may be valuable to the developer regardless

```
        error_code = -2
        raise TypeError(f"TypeError: {b=} not an int!")
    if b == 0:
        error_code = -3
        raise ValueError("ValueError: Cannot divide by 0!")
    return f"{a}/{b}={a/b}"
```

Listing 11: divide_ints.py

The author then wrote an ad-hoc ”test” that calls *divide_ints* with a variety of inputs, but no assertions as shown in List 14. Note the use of erroneous inputs to cover the error handling code.

```
# Use a global variable to test that unit_test_generator_decorator
# considers global variables when generating unit tests
error_code = 0

def main():
    """
    Call division function
    """
    start = time.perf_counter()
    a_list = [6, 3,"10", 8, [], {}, 4]
    b_list = [2, 0, 2, [], 2, 3, set()]
    global error_code  # pylint: disable=global-statement
    for a, b in zip(a_list, b_list):
        try:
            logger.info("-"*80)
            error_code = 0
            try:
                logger.info("Trying divide_ints(%s, %s)", repr(a), repr(b))
            except TypeError as e:
                logger.info(e)
            print(f"divide_ints({a}, {b})={divide_ints(a, b)}")
        except (ValueError, TypeError) as e:
            logger.error("%s: %s", type(e),str(e))
        logger.info("error_code=%d", error_code)

    generate_all_tests_and_metadata(Path('.'), Path('.'))
    print(f"Took {time.perf_counter()-start} seconds")

divide_ints = unit_test_generator_decorator(percent_coverage=110)(divide_ints)

main()
```

Listing 12: test divide_ints()

Running this code with pytest as displayed in Listing 6

```
    $ python divide_ints.py
```

Listing 13: Executing example to create unit tezst

produces the following *test_divide_ints.py* containing the following unit test (modified only slightly below to reduce line breaks):

```
"""
Programmatically generated test
function for divide_ints()
"""

import re
import pytest
import divide_ints
from _pytest.monkeypatch import MonkeyPatch

# Now import modules specific to divide_ints:

ERROR_CODE = 0


# In sum, these tests covered
# 100.0% of divide_ints's lines
@pytest.mark.parametrize(
    "a, b, exception_type, exception_message, \
    expected_result, expected_type, args_after, \
    globals_before, globals_after",
    [
        (
            "10",
            2,
            TypeError,
            "TypeError: Variable a='10' is not an int!",
            "None",
            "N/A",
            {},
            {},
            {"error_code": -1},
        ),
        (
            8,
            [],
            TypeError,
            "TypeError: Variable b=[] is not an int!",
            "None",
            "N/A",
            {"b": "[]"},
            {},
            {"error_code": -2},
        ),
        (6, 2, "N/A", "N/A", "6/2=3.0", str, {}, {}, {"error_code": 0}),
        (
            3,
            0,
            ValueError,
            "ValueError: Cannot divide by zero!",
            "None",
            "N/A",
            {},
            {},
            {"error_code": -3},
        ),
    ],
)
def test_divide_ints(
    a,
    b,
    exception_type,
    exception_message,
    expected_result,
    expected_type,
    args_after,
    globals_before,
    globals_after,
):
    """
    Programmatically generated test
    function for divide_ints()
    """
    monkeypatch = MonkeyPatch()
    monkeypatch.setattr(divide_ints, "error_code", ERROR_CODE)
```

```
77      if exception_type != "N/A":
78          with pytest.raises(exception_type,
79                             match=re.escape(
    exception_message)):
80              divide_ints.divide_ints(a, b)
81      else:
82          result = divide_ints.divide_ints(a, b)
83          assert result == expected_result \
84              or result == eval(expected_result)
85      for global_var_written_to in ["error_code"]:
86          if global_var_written_to in ["None", "[]", "
    {}"]:
87              assert not divide_ints.__dict__.get(
    global_var_written_to)
88          else:
89              assert (
90                  divide_ints.__dict__.get(
    global_var_written_to)
91                  == globals_after[
    global_var_written_to]
92              )
```

Listing 14: test divide_ints

Lines 18-20 set up the parameterization decorator, defining the inputs to each test, and the inputs to each are defined in a list of tuples spanning lines 23-33, 34-44, 45-57. The actual test function follows starting on line 60, monkeypatches the required ERROR_CODE global variable on lines 75-76, tests for expected exceptions on lines 77-80, and only calls the function on line 82 if it expects no exception. The result is verified on lines 82-83 [6]. An unexpected exception (or incorrect (exception, exception message) pair) would cause the unit test to fail. Finally, global variables expected to be modified are checked for correct values on lines 85-92. Note that not all test cases were retained due to the deduplication logic discussed in Section IV-B.

Executing the unit test is simple as shown in Listing 15[7].

```
1  pytest -s -v test_divide_ints.py
2  ========== test session starts ==========
3  platform win32 -- Python 3.11.7, pytest-7.4.4,
     pluggy-1.4.0 -- PATH
     REDACTED\.venv\Scripts\python.exe
4  cachedir: .pytest_cache
5  rootdir: PATH
     REDACTED\tests\example_procedural_division
6  plugins: cov-5.0.0
7  collected 4 items
8
9  test_divide_ints.py::test_divide_ints[Test #1
     arguments SNIPPED] PASSED
10 test_divide_ints.py::test_divide_ints[Test #2
     arguments SNIPPED] PASSED
11 test_divide_ints.py::test_divide_ints[Test #3
     arguments SNIPPED] PASSED
12 test_divide_ints.py::test_divide_ints[Test #4
     arguments SNIPPED] PASSED
13
```

Listing 15: Running Pytest

[6]Due to the complexity of converting from Python object to string (and sometimes back again), the author was forced due to time constraints to rely on the *eval* function to convert strings to valid Python. The author acknowleges this is a bad practice and hopes to fix it in future revisions.

[7]Due to the verbose way pytest prints the all the parameters of parameterized tests such as these, the author removed them for the sake of simpler display. The reader is encouraged to run the code as shown on their own machine

The reader is encouraged to study the other examples in the repository. All examples can be executed via the *tests/test_all** scripts. The reader can inspect those scripts to determine how to run each test individually. They are summarized very briefly below:

The *tests/example_fizzbuzz* tests I/O to a simple function and modification of a global value.

The *tests/example_all_types* tests the code against a variety of built-in Python types, such as ints, strings, sets, lists, tuples, and dictionaries.

The *tests/example_pass_by_assignment* tests functions that modify mutable arguments such as lists. Not a comprehensive test, but demonstrates that the concept works on one such example.

The *tests/example_oo_car* tests object-oriented code.

B. Application of this code to itself

The author naturally sought to determine the effectiveness of automatic test code generation against itself. This attempt at self-testing code created an intractable problem for this author due to the recursion in the call stack combined with lack of proper *repr* functions. That said, the author did successfully decorate three of the support functions for "self-testing", resulting in automatically generated, successfully passing unit tests for this portions of this very code. Using the previous Procedural Division example (and others) the reader should see the following test files created in addition to *test_divide_ints.py*

1) *test_coverage_str_helper.py*
2) *test_normalize_args.py*
3) *test_update_global.py*

These tests pass when run with this command in *tests/example_procedural_division/* directory:

```
1  pytest -s -v .
```

Listing 16: Running all generated unit tests for the division example

C. Application of this code to an external project

To validate this work on an external codebase, the author applied this project to a single function (due to time constraints) of his C code parser that inspired it. The author applied this decorator to a Python function designed to extract the name of any C functions from a provided line of code, then ran the sole function test that called the decorated function extensively with various and redundant inputs. The applied decorator worked as desired, including deduplicating results as previously discussed, rapidly capturing over three dozen unique test cases. Upon visually inspecting the inputs and outputs, the author nearly instantly identified two bugs in the decorated function. After patching both bugs revealed by the unit tests, he re-ran the function test, and inspected the newly created unit tests. At that point he was confident that the generated unit test and dozens of tests cases sufficed for unit testing apart from the functional test.

Satisfied, he removed the decorator, restoring the functional test to its original runtime. Thus, under ideal situations, the

tester only pays the decoration penalty as few as two times per function. Of course, multiple functions can be decorated at once.

VII. Discussion

In this paper and accompanying code the author has proved his hypothesis that the metaprogramming abilities of Python can be used to programmatically create multiple unit tests from even a single existing test.
The decorator and supporting code discussed above (and provided in the accompanying repository) were written in Python 3.11 and tested on a Windows 10 laptop with 16GB of memory and 6 core i7-9750H @2.60Ghz CPU. The author did not test it on *nix environments but given the cross-platform support of Python, the author believes it should run in *nix environments with no more than minor modifications. The author releases it under the GNU LGPL open-source license.

A. Areas for future work

As partly noted above, future work should:

1) Reduce reliance on *eval*
2) Add support for file and socket I/O
3) Compare runtime performance of various algorithms for keeping minimal subsets of tests
4) Add multi-thread and multi-process testing and support
5) Investigate applicability of these ideas to other OOP languages such as C#, Go, and Zig
6) Verify cross-platform compatibility
7) Determine backwards compatibility with older Python versions

B. Limitations

As noted in Section II, in order to generate unit tests as described in this paper, the process must be boot-strapped with a initial passing test.
Additionally, for very large applications, or large functions with complex inputs, memory constraints may limit the number of functions that can be decorated at once, which would slow down the process of creating a large number of unit tests.

C. Related Work

This work is perhaps most similar to [15], though differs in that rather than prompt users interactively for sample inputs and outputs, it automatically detects such inputs, outputs, and changes in global state by monitoring actual execution of the code. It is also worth noting that the work presented here could easily be applied to AI or otherwise automatically generated tests. Although in one paper the AI-generated end-to-end tests were written in Java [22], analogously created Python tests could be used with the work presented here to generate previously non-existent tests at the unit level.
In the current research this author did not discover a deterministic unit test generation paradigm that monitored other types of tests from which to create unit tests.

VIII. Conclusion

In this paper the author presented a Python decorator leveraging previously designed test(s) to directly create unit tests at minimal development cost. The trade-off is that this approach increases execution time of the function tests and, like AI-generated tests, such tests would still require review by the developer. The latter is true of any automatically generated test and the author proposed and demonstrated a variety of approaches to mitigate the increased runtime for existing tests. The author proved his initial hypothesis that that unit tests could be created programmatically by monitoring and capturing metadata from existing tests.
Although much work remains, the author maintains an optimistic outlook that the testing paradigm presented here could prove valuable in the world of automated test generation due to the variety of successful tests noted above. By generating unit tests from functional tests, not only are the unit tests created essentially for free, but the quality (i.e. coverage) of other tests generated via BDD or ATDD processes can be assessed. This could enable rapid comparison of the relative test coverage of various testing paradigms.
The code is available in a single Python file to permit easy integration into any supported Python project. The author wishes to give back to the computing community and therefore offers this project free and open source in the hopes that it will be found useful and lay the ground work for future research and application. The author hopes the concepts described herein will be applied in other languages with strong metaprogramming support such as C#, Go, Zig, etc. The full repo containing both the Python code and LaTeX files for this report can be found in the ”ddt” repo at:

https://github.com/jahatfi/ddt

IX. Acknowledgements

The author wishes to thank Drs. Dan and Mary Lou Midgett for their assistance in technical writing, Lt Col Solomon Sonya for his input on the abstract, and Mr. Brad Stinson for sharing his pytest expertise.

REFERENCES

[1] H. Percival, *Test-driven development with Python: obey the testing goat: using Django, Selenium, and JavaScript*. " O'Reilly Media, Inc.", 2014.

[2] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors limiting industrial adoption of test driven development: A systematic review," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 337–346.

[3] H. A. Ramzan, S. Ramzan, and T. Kalsum, "Test-driven development (tdd) in small software development teams: Advantages and challenges," in *2024 5th International Conference on Advancements in Computational Sciences (ICACS)*. IEEE, 2024, pp. 1–5.

[4] A. Axelrod and A. Axelrod, "Unit tests and tdd," *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*, pp. 395–424, 2018.

[5] A. W. Brown, S. Ambler, and W. Royce, "Agility at scale: economic governance, measured improvement, and disciplined delivery," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 873–881.

[6] S. M. Shahabuddin and Y. Prasanth, "Integration testing prior to unit testing: A paradigm shift in object oriented software testing of agile software engineering," *Indian Journal of Science and Technology*, vol. 9, no. 20, pp. 1–10, 2016.

[7] M. M. Moe, "Comparative study of test-driven development tdd, behavior-driven development bdd and acceptance test–driven development atdd," *International Journal of Trend in Scientific Research and Development*, vol. 3, no. 4, pp. 231–234, 2019.

[8] A. C. Barus, "The implementation of atdd and bdd from testing perspectives," in *Journal of Physics: Conference Series*, vol. 1175, no. 1. IOP Publishing, 2019, p. 012112.

[9] K. Pugh, *Lean-Agile acceptance test-driven-development*. Pearson Education, 2010.

[10] K. Kahur and J. Su, "Java unit testing with ai: An ai-driven prototype for unit test generation," 2023.

[11] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, April 2024.

[12] S. Lukasczyk, F. Kroiß, G. Fraser, and P. Contributors, "se2p/pynguin: Pynguin v0.17.0," Feb. 2022.

[13] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative ai: A comparative performance analysis of autogeneration tools," *arXiv preprint arXiv:2312.10622*, 2023.

[14] W. Takerngsaksiri, R. Charakorn, C. Tantithamthavorn, and Y.-F. Li, "Tdd without tears: Towards test case generation from requirements through deep reinforcement learning," *arXiv preprint arXiv:2401.07576*, 2024.

[15] S. K. Lahiri, S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, P. Choudhury, C. von Veh, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," 2023. [Online]. Available: https://arxiv.org/abs/2208.05950

[16] N. Rao, K. Jain, U. Alon, C. L. Goues, and V. J. Hellendoorn, "Cat-lm training language models on aligned code and tests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2023, pp. 409–420.

[17] E. Sundqvist, "Ai-assisted unit testing: Empirical insights into github copilot chat's effectiveness and collaborative benefits," 2024.

[18] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[19] P. Kang, "Function call interception techniques," *Software: Practice and Experience*, vol. 48, no. 3, pp. 385–401, 2018.

[20] N. Batchelder and C. to Coverage.py, "Coverage.py: The code coverage tool for python," Aug. 2024.

[21] R. Hassin and A. Levin, "A better-than-greedy approximation algorithm for the minimum set cover problem," *SIAM Journal on Computing*, vol. 35, no. 1, pp. 189–200, 2005.

[22] M. Leotta, H. Z. Yousaf, F. Ricca, and B. Garcia, "Ai-generated test scripts for web e2e testing with chatgpt and copilot: A preliminary study," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 339–344.