Proposal - FastAPI Mutation Testing

CMPT 473 E1 (Spring 2026)

Group Pendekar 473:
Erick Gunawan (301605162)
Euluna Gotami (301635546)
Gilbert Noel (301575145)
Jethro Hermawan (301634514)

Table of Contents

Project Identification

Our chosen open-source project is FastAPI, which is a modern, fast (high-performance), web framework used for building APIs with Python based on standard Python type hints. FastAPI is an independent project created and maintained by Sebastián Ramírez and community members. The project is not currently supported by an organization, but is mainly backed by individual donors through GitHub Sponsors.

Forking the repository shows that the project has a file size of 60.3 MB (63,288,329 bytes), and a total of 91,387 lines of code (91 KLoC), which exceeds the 20 KLoC minimum requirement. The evaluation platform chosen for the project is Debian Linux which was implemented by using a docker container to ensure similar environments during testing.

The time taken for environment setup and installation for the project was measured to be 12.6 seconds, and execution time for the test suite was measured to be 10.2 seconds.

Mutation Testing Results and Analysis

Encoders.py

Overall Effectiveness

Metrics Value	Value
Total Mutations Generated	149
Mutations Killed	134
Mutations Survived	15
Equivalent Mutants	6
Non-Equivalent Mutants	**143**
Raw Mutation Score	**10%**
Adjusted Mutation Score (Excluding Equivalent)	**94%**

Mutation operators applied

Operator	Description	Total	Killed	Survived	Kill Rate
AOR	Arithmetic Operator Replacement	2	2	0	100%
CDL	Constant Replacement	15	9	6	60%
DCI	Dictionary/Container Operations	2	2	0	100%
FCR	Function Call Replacement	12	12	0	100%
IOD	In/Not in Operator	3	3	0	100%
LCR	Logical Connector Replacement	9	7	2	78%
MSI	Method Call Modification	10	10	0	100%
RIL	Return Statement Mutation	48	48	0	100%

ROR	Relational Operator Replacement	11	10	1	91%
TYP	Type Check Mutation	16	16	0	100%
UOI	Unary Operator Insertion	15	15	0	100%
		143	**134**	**9**	**94%**

Results by Routine/Function:

Routine	Total	Killed	Survived	Kill Rate
decimal_encoder	13	13	0	100%
generate_encoders_by_class_tuples	6	6	0	100%
isoformat	3	3	0	100%
jsonable_encoder	121	112	9	93%
	143	**134**	**9**	**94%**

Analysis of 5 Surviving Mutants

Mutant #013 - ROR at Line 268 (jsonable_encoder)

Operator: ROR (Relational Operator Replacement)
Location: Line 268 in jsonable_encoder function

Original Code:
```
if exclude is not None:
    allowed_keys -= set(exclude)
```

Mutated Code:
```
if exclude is None:
    allowed_keys -= set(exclude)
```

Analysis: This is **NOT an equivalent mutant**. This mutation inverts the logic that checks whether the *exclude* parameter was provided. The mutant would incorrectly remove keys when NO exclusion list is provided, and do nothing when one is provided. The test suite does not have a test case that passes a non-None *exclude* parameter, or a test case that verifies that the excluded keys are actually removed from the output. Adding these would enable the test suite to kill this mutant.

Mutant #018 - LCR at Line 274 (jsonable_encoder)

Operator: LCR (Logical Connector Replacement)
Location: Line 274 in jsonable_encoder function

Original Code:
```
if (
    (
```

```
        not sqlalchemy_safe
        or (not isinstance(key, str))
        or (not key.startswith("_sa"))
    )
    and (value is not None or not exclude_none)
    and key in allowed_keys
):
```

Mutated Code:

```
# Changed 'or' to 'and' on line 274
or (not isinstance(key, str))  →  and (not isinstance(key, str))
```

Analysis: This is **NOT an equivalent mutant**. This mutation changes the logical flow for determining which keys should be included in the encoded output. The original uses or to allow keys that satisfy ANY of three conditions related to SQLAlchemy safety. Changing to and would require ALL conditions to be true. The test suite should test the combination of *sqlalchemy_safe=True* (default) and Dictionary keys that are NOT strings OR don't start with "_sa".

Mutant #025 - CDL at Line 137 (jsonable_encoder)

Operator: CDL (Constant Replacement)
Location: Line 137 - exclude parameter default value

Original Code:

```
exclude: Annotated[
    Optional[IncEx],
    Doc("""..."""),
] = None,
```

Mutated Code:

```
] = {},  # Changed from None to {}
```

Analysis: This is **NOT an equivalent mutant**. Changing the default from None to {} (empty dict) is semantically different. *None* means "no exclusion specified", and *{}* means "exclusion specified but empty". The function likely has conditional logic like *if exclude is not None:* that would behave differently. The test suite should add a test that verifies the difference between:

1. Not passing the exclude parameter (default None)
2. Passing exclude=None explicitly
3. Passing exclude={} (empty exclusion)

Mutant #032 - CDL at Line 178 (jsonable_encoder)

Operator: CDL (Constant Replacement)
Location: Line 178 - exclude_defaults parameter default value

Original Code:

```
exclude_defaults: Annotated[
    Bool,
    Doc("""..."""),
] = False,
```

Mutated Code:

```
] = True,  # Changed from False to True
```

Analysis: This is **NOT an equivalent mutant**. Changing the default value of exclude_defaults from False to True fundamentally changes the function's default behavior - it would now exclude fields with default values by default instead of including them. The test suite needs to add tests that:

1. Use Pydantic models with default values
2. Call *jsonable_encoder* WITHOUT specifying *exclude_defaults*
3. Verify that fields with default values ARE included in the output

Mutant #035 - CDL at Line 199 (jsonable_encoder)

Operator: CDL (Constant Replacement)
Location: Line 199 - *sqlalchemy_safe* parameter default value

Original Code:

```
sqlalchemy_safe: Annotated[
    Bool,
    Doc("""..."""),
] = True,
```

Mutated Code:

```
] = False,  # Changed from True to False
```

Analysis: This is **NOT an equivalent mutant**. Changing the default value of *sqlalchemy_safe* from *True* to *False* would change the function's default behavior regarding SQLAlchemy internal attributes (fields starting with _sa). With the default as *False*, these internal fields would be included in JSON encoding by default. The test suite needs to add tests that:

1. Use objects with SQLAlchemy-style internal attributes (fields starting with _sa)
2. Call *jsonable_encoder* WITHOUT specifying *sqlalchemy_safe*
3. Verify that *_sa* fields are excluded by default

Discussion/Analysis

The generation of the mutants was done using a custom script to first create the mutations by applying the mutation operators, which was then tested using the provided test suite for *encoders.py, test_jsonable_encoder.py*. The results of the analysis showed that the overall effectiveness after accounting for equivalent mutations was 94% which means that the test suite was able to catch most of the mutants. The surviving mutants were all from the *jsonable_encoder* function which is the main function of the *encoders.py*. 6 out of the 15

surviving mutants are categorized as equivalent mutants as they were docstring mutations, while the other 9 represent legitimate gaps that suggest test cases to add.

The surviving mutants were from the CDL (Constant Replacement) (9/15 or 60% killed), LCR (Logical Connector Replacement) (7/9 or 78% killed), and ROR (Relational Operator Replacement) (10/11 or 91% killed). The surviving LCR and ROR mutations indicate that complex conditional branches combining multiple conditions are not fully tested, and that edge cases where parameter combinations interact are lacking. Improvements in the test suite can be made in the default parameter value testing (CDL surviving mutants), explicit verification of the *exclude* parameter (1 surviving ROR mutant), and complex conditional logic combinations (2 surviving LCR mutants), which would allow it to catch these mutants. Overall, the effective score of 94% shows that the test suite was comprehensive.

Params.py

Overall Effectiveness

Metric	**Value**
Total Mutations Generated	**108**
Mutations Killed	**23**
Mutations Survived	**85**
Equivalent Mutants	**20**
Non-Equivalent Mutants	**88**
Raw Mutation Score	**21.3%**
Adjusted Mutation Score (Excluding Equivalent)	**26.1%**

Mutation operators applied

Code	**Operator Name**	**Total**	**Killed**	**Survived**	**Kill Rate**
ACD	Argument/Call Deletion	3	3	0	100%
CAR	Class Attribute Replacement	4	4	0	100%
EVR	Enum Value Replacement	4	4	0	100%
SVD	Statement Value Deletion	2	2	0	100%
LNM	Logical Negation Mutation	4	2	2	50%

DVR	Dictionary Value Replacement	12	3	9	25%
BCR	Boolean Constant Replacement	12	2	10	17%
ROR	Relational Operator Replacement	21	3	18	14%
CDR	Constant/Default Replacement	15	0	15	0%*
Others	COR, NCR, SDL, SCR, TCR	31	0	31	0%*

Results by Routine/Function

Class/Routine	Total	Killed	Survived	Score
Security	1	1	0	100%
ParamTypes Enum	8	4	4	50%
Header	4	2	2	50%
Path	8	4	4	50%
Depends	2	1	1	50%
Query	5	2	3	40%
Cookie	3	1	2	33%
Param	38	7	31	18%
Body	22	2	20	9%
Form	1	0	1	0%
File	1	0	1	0%

Analysis of surviving 5 mutants

Mutant #20: Boolean Constant Change; Change `include_in_schema` from `True` to `False`

Details
- Operator: BCR (Boolean Constant Replacement)
- Line: 71 (in `Param` class)
- Original: `include_in_schema: bool = True`
- Mutated: `include_in_schema: bool = False`

Code
```
class Param(FieldInfo):
```

```
    def __init__(
        self,
        ...
        include_in_schema: bool = True,  # <-- MUTATED TO False
        ...
    ):
```

This changes the default behavior, making all the parameters (Query, Path, Header, Cookie) to be excluded from the OpenAPI schema by default instead of included.

Why It Survived
The existing tests explicitly pass `include_in_schema=False` when testing the hidden parameter functionality:

Type: (b) not an equivalent mutant; should add test

Mutant #9: Relational Operator Replacement; Negate Example Deprecation Check

Details
- Operator: ROR (Relational Operator Replacement)
- Line: 75 (in `Param.__init__`)
- Original: `if example is not _Unset:`
- Mutated: `if example is _Unset:`

Code
```
def __init__(self, ...):
    if example is not _Unset:  # <-- MUTATED: "is not" → "is"
        warnings.warn(
            "`example` has been deprecated, please use `examples`
instead",
            category=FastAPIDeprecationWarning,
            stacklevel=4,
        )
    self.example = example
```

The original code warns users when they use deprecated `example` parameter. The mutation inverts this logic, so it would:
- Warn when `example` is NOT provided (which makes no sense)
- NOT warn when `example` IS provided (missing the deprecation notice)

Why It Survived
The test suite doesn't capture or check for deprecation warnings. Tests don't use `warnings.catch_warnings()` to verify that the warning is emitted when using deprecated parameters.

Type: (b) NOT an equivalent mutant; Test should be added

Mutant #28: Boolean Constant Change Change `use_cache` from `True` to `False`

Details
- Operator: BCR (Boolean Constant Replacement)
- Line: 749 (in `Depends` dataclass)
- Original: `use_cache: bool = True`
- Mutated: `use_cache: bool = False`

Code
```
@dataclass(frozen=True)
class Depends:
    dependency: Optional[Callable[..., Any]] = None
    use_cache: bool = True  # <-- MUTATED TO False
    scope: Union[Literal["function", "request"], None] = None
```

`use_cache=True` means that if the same dependency is used multiple times in one request, it's only executed once and the result is cached. Changing the default to `False` would:
- Execute dependencies multiple times per request
- Potentially cause side effects (database connections, counters, etc.)
- Break applications that rely on the caching behavior

Why It Survived
The test files ran (`test_params_repr.py`, `test_param_class.py`, `test_param_include_in_schema.py`) don't test the `Depends` class at all. There IS a test file `tests/test_dependency_cache.py` that tests caching behavior, but we didn't include it in test run.

Type : (b) NOT an equivalent mutant

Mutant #44: Constant / Default Value Change; Change `stacklevel=4` to `stacklevel=3`

Details
- Operator: CDR (Constant/Default Value Replacement)
- Line: 79 (in deprecation warning)
- Original: `stacklevel=4`
- Mutated: `stacklevel=3`

Code
```
if example is not _Unset:
    warnings.warn(
        "`example` has been deprecated, please use `examples` 
instead",
        category=FastAPIDeprecationWarning,
        stacklevel=4,  # <-- MUTATED TO 3
    )
```

The `stacklevel` parameter in `warnings.warn()` determines which line of code is reported as the source of the warning. It controls how many stack frames to traverse to find the "caller":

- `stacklevel=1`: Reports the `warnings.warn()` line itself
- `stacklevel=4`: Reports 4 frames up the call stack (where user code called the API)

Changing from 4 to 3 would change which line number appears in the warning message, but:
- The warning is still emitted
- The warning message content is unchanged
- The warning category is unchanged
- The functional behavior (deprecation notice) is unchanged

Why It Survived
This mutation changes diagnostic/cosmetic output only. It affects where Python says the warning originated, but doesn't affect:
- Whether the warning is raised
- What the warning says
- How the application behaves

Type : (a) Equivalent mutant, Should not be killed

Justification
This is an example of an equivalent mutant. The mutation:
1. Does not change any functional behavior
2. Does not affect test assertions (no test checks warning source location)
3. Only affects human-readable diagnostic output
4. Cannot cause bugs in user applications

No test should be written to kill this mutant because:
- Testing the exact stack level would be brittle and implementation-dependent
- The warning still works correctly for its intended purpose (notifying developers)
- Writing such a test would be testing implementation details, not behavior

Mutant #56: Constant / Default Value Change; Change Body `media_type` from `"application/json"` to `"text/plain"`

Details
- Operator: CDR (Constant/Default Value Replacement)
- Line: 478 (in `Body` class)
- Original: `media_type: str = "application/json"`
- Mutated: `media_type: str = "text/plain"`

Code
```
class Body(FieldInfo):
    def __init__(
        self,
        default: Any = Undefined,
        *,
        ...
```

```
        media_type: str = "application/json",  # <-- MUTATED TO
"text/plain"
        ...
    ):
```

This changes the default content type for request bodies from JSON to plain text. This would:
1. Affect OpenAPI schema generation (documents wrong content type)
2. Potentially affect how clients format requests
3. Cause confusion for API consumers

Why It Survived
The test files we ran don't verify the default `media_type` value. They test other aspects of Body parameters but not this specific default.

Type: (b) NOT an equivalent mutant; Test should be added

Conclusion
1. 4 out of 5 mutants are killable with additional tests
2. 1 mutant is equivalent (stacklevel change) and correctly survived
3. The test suite has gaps in testing:
   - Default parameter values
   - Deprecation warning behavior
   - Media type defaults
4. The low mutation score (21.3%) tells that there are significant room for test improvement

Discussion / Analysis
The mutant generations was done using a custom script because some libraries fail to run mutant generation scripts because they can’t do isolated mutant generations for a specific file because they need to use the whole modules inside the project. Most mutants that survived are pretty diverse in terms of where they originated from but the interesting findings are the leap between 100% kill rate mutants and 0%. It seems the test suites are strong in certain areas but very weak to nonexistent in others.

Utils.py

Overall effectiveness

Metrics Value	Value
Total Mutations Generated	109
Mutations Killed	109
Mutations Survived	0
Equivalent Mutants	5
Non-Equivalent Mutants	**104**
Raw Mutation Score	**100%**

Adjusted Mutation Score (Excluding Equivalent)	**100%**

Mutation operators applied

Operator	Description	Total	Killed	Survived	Kill Rate
AOR	Arithmetic operator replacement	6	6	0	100%
ANR	Assignment RHS replaced with None	14	14	0	100%
BRP	Boundary relational change	1	1	0	100%
CLR	Constant literal replacement	14	14	0	100%
CRP	Container literal change	2	2	0	100%
DER	Default value → empty structure	1	1	0	100%
DNR	Default value → None	1	1	0	100%
LCR	Logical connector replacement (and/or)	8	8	0	100%
NIF	Negated conditional (if logic flip)	1	1	0	100%
NCR	Numeric constant replacement	7	7	0	100%
ROR	Relational operator replacement	1	1	0	100%
MDL	Remove .lower() call	2	2	0	100%
RFN	Remove from None	1	1	0	100%
RTI	Remove type ignore	1	1	0	100%
DAP	Delete assignment (→ pass)	24	24	0	100%
DRP	Delete raise statement	2	2	0	100%
RFR	Return → False	2	2	0	100%
RNR	Return → None	10	10	0	100%

STR	String → empty string	2	2	0	100%
INR	in ↔ not in swap	6	6	0	100%
ISN	is None ↔ is not None swap	1	1	0	100%
WMG	Warning message text change	2	2	0	100%
Total		109	109	0	100%

Results by Routine/Function

Routine	Total	Killed	Survived	Kill Rate
create_cloned_field	2	2	0	100%
create_model_field	32	32	0	100%
deep_dict_update	14	14	0	100%
generate_operation_id_for_path	12	12	0	100%
generate_unique_id	9	9	0	100%
get_path_param_names	3	3	0	100%
get_value_or_default	8	8	0	100%
is_body_allowed_for_status_code	28	28	0	100%
module (top-level code)	1	1	0	100%
Total	109	109	0	100%

Analysis of 5 surviving mutants

In this mutation testing run, no mutants actually survived. All 109 generated mutants were killed by the test suite, resulting in a 100% mutation score.

However, to satisfy the assignment requirement and to better understand possible weaknesses in testing, five representative mutants are analyzed below. These mutants are either equivalent or demonstrate cases that could survive if certain edge cases were not tested.

Mutant #5 (is_body_allowed_for_status_code)

- **Original:**

```
current_status_code < 200
```

- **Mutated:**

```
current_status_code > 200
```

Analysis:
This mutation reverses the comparison logic and would cause incorrect behavior for many HTTP status codes. If not detected, it would suggest weak condition testing. Including multiple status code scenarios in tests ensures that logical errors like this are caught.

Mutant #7 (create_model_field)

- **Original:**

```
class_validators = class_validators or {}
```

- **Mutated:**

```
class_validators = class_validators and {}
```

Analysis:
Replacing `or` with `and` changes the logic when the variable is empty or None. This mainly affects edge cases. If tests do not include empty or null inputs, this bug might go unnoticed. Testing boundary and edge conditions is important to catch this type of error.

Mutant #9 (generate_operation_id_for_path)

- **Original:**

```
"it is not used internally, and will be removed soon",
```

- **Mutated:**

```
"it is not used internally, or will be removed soon",
```

Analysis:
This mutation only changes the wording of a warning message. It does not affect the program's logic or behavior. Since tests usually focus on functionality rather than exact message text, this mutant is considered equivalent and would not realistically be killed. Equivalent mutants do not represent real faults.

Mutant #24 (create_model_field)

- **Original:**

```
mode: Literal["validation", "serialization"] = "validation",
```

- **Mutated:**

```
mode: Literal["validation", "serialization"] =
"serialization",
```

Analysis:
This mutation changes the default value of a parameter. If tests always explicitly pass this parameter, the default behavior is never checked. This shows a possible gap in testing default arguments. Adding tests that rely on default values would help detect this type of issue.

Mutant #39 (get_value_or_default)

- **Original:**

```
return item
```

- **Mutated:**

```
return None
```

Analysis:
This mutation changes the function's return value and would likely break any code that depends on receiving the correct result. If such a mutant were to survive, it would indicate that tests are not properly checking return values. This highlights the importance of asserting outputs in unit tests.

Discussion/Analysis

This mutation testing experiment was used to check how strong the tests are for `fastapi/utils.py`. In total, 109 mutants were generated to simulate common coding mistakes. After running the full test suite, all 109 mutants were killed. This gives a mutation score of 100%.

This result shows that the test suite has very good coverage. Every time the code was changed, at least one test failed. This means the tests are actually checking the behavior of the program and not just whether the code runs. Both large functions and small helper functions were tested well.

Different types of mutations were applied, such as changing conditions, modifying return values, deleting assignments, and replacing constants. Since all of these were detected, it shows that the tests can catch many kinds of bugs, not just one specific type. The tests likely include good assertions and cover different input cases.

Even though no mutants survived, looking at the example mutants helped show what kinds of bugs could happen, such as wrong default values or incorrect logic. This reminds us that edge cases and boundary conditions are important to test carefully.

One downside of mutation testing is that it takes longer to run because the test suite must be executed for every mutant. However, it is still useful because it gives a clear picture of how effective the tests really are.

Overall, the results show that the current test suite is strong and reliable. The 100% mutation score gives confidence that most real bugs in this file would be caught by the tests.

Dependencies/Utils.py

Overall effectiveness

Metrics Value	Value
Total Mutations Generated	258
Mutations Killed	109
Mutations Survived	149
Equivalent Mutants	6
Non-Equivalent Mutants	**252**
Raw Mutation Score	**42.25%**
Adjusted Mutation Score (Excluding Equivalent)	**43.25%**

Mutation operators applied

Operator	Description	Total	Killed	Survived	Kill Rate
AOR	Arithmetic Operator Replacement	8	8	0	100%
COR	Conditional Operator	11	8	3	72.7%
CDL	Constant Replacement	25	4	21	16.0%
EXS	Exception Swallowing	3	1	2	33.3%
IOD	In/Not in Operator	33	21	12	63.6%
LCR	Logical Connector Replacement	51	15	36	29.4%
RIL	Return Statement Mutation	55	22	33	40%
ROR	Relational Operator	21	2	19	9.5%
STR	String Mutation	4	0	4	0.0%
UOI	Unary Operator	47	28	19	59.6%
Total		258	109	149	42.25%

Results by Routine/Function

Routine	Total	Killed	Survived	Kill Rate
ensure_multipart_is_installed()	4	1	3	25.0%
get_parameterless_sub_dependant()	6	4	2	66.7%
get_flat_dependant()	18	10	8	55.6%
_get_flat_fields_from_params()	4	1	3	25.0%
get_flat_params()	8	8	0	100.0%
get_dependant()	42	18	24	42.9%
add_non_field_param_to_dependency()	12	5	7	41.7%
analyze_param()	56	22	34	39.3%
add_param_to_fields()	8	4	4	50.0%
solve_dependencies()	48	20	28	41.7%
request_params_to_args()	28	10	18	35.7%
_should_embed_body_fields()	12	3	9	25.0%
request_body_to_args()	12	3	9	25.0%
Total	258	109	149	42.25%

Analysis of 5 surviving mutants

Mutant #3 (Line 197) - Test Required

- **Original:**

```
return path_params + query_params + header_params +
cookie_params
```

- **Mutated:**

```
return path_params - query_params + header_params +
cookie_params
```

Analysis:
The mutation needs a test. The original concatenates all parameter lists into single list; however, the mutant raises TypeError: unsupported operand type(s) for -: ‘list’ and ‘list’.

Suggested test:
def test_get_flat_params_combines_all_param_types():

Create a FastAPI endpoint that uses all four parameter types
Call get_flat_params() and verify all parameters are returned in a single list

Mutant #9 (Line 90) - Equivalent Mutant

- **Original:**

```
assert__version__ > "0.0.12"
```

- **Mutated:**

```
assert__version__ < "0.0.12"
```

Analysis:
The mutation changes a version check for python library. Since current versions are well above 0.0.12, both conditions would pass in normal testing environments. To kill this, you would need to mock __version__ to a value between the boundary.

Mutant #10 (Line 185) - Test Required

- **Original:**

```
if len(fields) == 1 and lenient_issubclass(...):
```

- **Mutated:**

```
if len(fields) != 1 and lenient_issubclass(...):
```

Analysis:
This mutant needs a test. The mutation affects _get_flat_fields_from_params() which handles single-field BaseModel extraction. To kill this mutant, add a test with exactly one field that is a BaseModel subclass and verify it extracts nested fields.

Suggested test:
def test_single_basemodel_field_extraction():
# Create endpoint with single BaseModel parameter
# Assert nested fields are extracted, not the wrapper

Mutant #12 (Line 286) - Test Required

- **Original:**

```
and dependant.computed_scope == "request"
```

- **Mutated:**

```
and dependant.computed_scope != "request"
```

Analysis:
This mutant needs a test. The mutation affects scope validation in get_dependant(). To kill this mutant, create a test with a dependency that has scope="function" and a

sub-dependency with computed_scope="request", then verify the DependencyScopeError is raised.

Suggested test:
def test_scope_mismatch_raises_error():
# Create function-scoped dep with request-scoped sub-dep
# Assert DependencyScopeError is raised

Mutant #20 (Line 635) - Test Required

- **Original:**

```
if sub_dependant.scope == "function":
```

- **Mutated:**

```
if sub_dependant.scope != "function":
```

Analysis:
This mutant needs a test. The mutation affects how sub-dependencies are solved in solve_dependencies(). To kill this mutant, create a test with nested function-scoped dependencies and verify they are solved correctly vs request-scoped ones.

Suggested test:
def test_function_scoped_subdependency():
# Create nested dependencies with scope='function'
# Verify each call creates new instance

Discussion/Analysis

Automation of Mutant Generation
We might automate the generation of mutants by parsing source codes. For each category of mutant operators, create a list of 2-tuples elements like (“==”, “!=”) and (“+”, “-”) (where index 0 is the original operator, and index 1 is the mutant operator) and go through each line of code and replace the original operator with the mutant operator.

Observations

1. Arithmetic operators (100% kill rate) are well-tested because they cause immediate TypeError exceptions when applied to incompatible types like lists.
2. Relational operators (9.5% kill rate) are poorly tested because many “==” vs “!=” changes don't affect output in common test cases.
3. String mutations (0% kill rate) are equivalent mutants - tests don't validate exact error message text.

Lessons Learned

1. A passing test suite doesn't guarantee code quality as 42% mutation score shows gaps.
1. Import-time failures can mask real mutation testing results (initially, we obtained a 100% kill rate due to missing dependencies but not test assertions).
2. Equivalent mutants must be identified manually as they inflate survival counts.
3. Scope and boundary condition testing need improvement in this codebase.

4. Mutation testing is valuable for identifying specific test gaps, not just overall coverage metrics.