Exercise 2: Automated Testing & Coverage

-Aryan Ghosh

GitHub Repository

Table of Contents

PART 1: BASELINE COVERAGE 2

1. HumanEval/155 2
2. HumanEval/101 3
3. HumanEval/132 4
4. HumanEval/136 5
5. HumanEval/156 6
6. HumanEval/160 7
7. HumanEval/159 7
8. HumanEval/139 9
9. HumanEval/141 10
10. HumanEval/151 11

PART 2: LLM-ASSISTED TEST GENERATION & COVERAGE IMPROVEMENT 13

1. HumanEval/160 13
2. HumanEval/139 14

PART 3: FAULT DETECTION CHECK 15

1. HumanEval/160 15
2. HumanEval/139 16

Part 1: Baseline Coverage

All tests were run on GPT5-Prompt_1 generated code from Exercise 1.

1. **HumanEval/155**

Base prompt:

def even_odd_count(num):

Given an integer, return a tuple that has the number of even and odd digits respectively.

- **Number of tests Passed**
  1 test file run, with 5 true and prompt-based cases (all passed).

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 100%

```
Name                    Stmts   Miss  Cover
-------------------------------------------
even_odd_count.py          10      0   100%
test_even_odd_count.py      8      0   100%
-------------------------------------------
TOTAL                      18      0   100%
```

  - Branch Coverage: 100%

```
Name               Stmts   Miss Branch BrPart  Cover
----------------------------------------------------
even_odd_count.py     10      0      4      0   100%
----------------------------------------------------
TOTAL                 10      0      4      0   100%
```

- **Interpretation**
  All code paths, including the sign-handling and both even/odd branches of the digit evaluation, are exercised by the benchmark test suite. The output confirms comprehensive test coverage and correct counting logic for a variety of integer inputs.

2. HumanEval/101

Base prompt:

```
def words_string(s):
    """
    You will be given a string of words separated by commas or spaces. Your task is to split the
string into words and return an array of the words.
```

- **Number of tests Passed**
  1 test file run, with 5 true and prompt-based cases (all passed).

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 100%

```
Name                 Stmts   Miss  Cover
----------------------------------------
test_words_string.py      8      0   100%
words_string.py           3      0   100%
----------------------------------------
TOTAL                    11      0   100%
```

  - Branch Coverage: 100%

```
Name            Stmts   Miss Branch BrPart  Cover
-------------------------------------------------
words_string.py      3      0      0      0   100%
-------------------------------------------------
TOTAL                3      0      0      0   100%
```

- **Interpretation**
  All code lines and both branches (matches separator or not) are exercised by the benchmark test suite, confirming full coverage for both comma and space separators as well as edge formatting cases.you are a gasy mtf

3. **HumanEval/132**

def is_nested(string):

""Create a function that takes a string as input which contains only square brackets.

The function should return True if and only if there is a valid subsequence of brackets where at least one bracket in the subsequence is nested.

- **Number of tests Passed**
  2 tests passed (8 total, 4 true cases, 4 false cases)

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 100%

```
Name              Stmts   Miss  Cover
-------------------------------------
is_nested.py          9      0   100%
test_is_nested.py    12      0   100%
-------------------------------------
TOTAL                21      0   100%
```

  - Branch Coverage: 100%

```
Name          Stmts   Miss Branch BrPart  Cover
---------------------------------------------
is_nested.py      9      0      6      0   100%
---------------------------------------------
TOTAL             9      0      6      0   100%
```

- **Interpretation**
  All code lines and all branch paths for this function are exercised by the benchmark test suite, indicating complete coverage for all main logic, including edge and error cases

4. **HumanEval/136**

Base prompt:

def largest_smallest_integers(lst):

Create a function that returns a tuple (a, b), where 'a' is the largest of negative integers, and 'b' is the smallest of positive integers in a list.

If there is no negative or positive integers, return them as None.

- **Number of tests Passed**
  1 test file run, with 7 true and prompt-based cases (all passed).

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 100%

```
Name                                Stmts   Miss  Cover
-----------------------------------------------------
largest_smallest_integers.py          11      0   100%
test_largest_smallest_integers.py     10      0   100%
-----------------------------------------------------
TOTAL                                 21      0   100%
```

  - Branch Coverage: 100%

```
Name                          Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------------
largest_smallest_integers.py 11      0     10      0   100%
-----------------------------------------------------------
TOTAL                          11      0     10      0   100%
```

- **Interpretation**
  All code lines and logical branches (largest negative, smallest positive, and None-for-missing cases) are exercised by the benchmark tests. The coverage results confirm that all relevant edge cases, such as lists with only negatives, only positives, zeros, or duplicates are covered.

5. **HumanEval/156**

Base prompt:

def int_to_mini_roman(number):

Given a positive integer, obtain its roman numeral equivalent as a string, and return it in lowercase.

Restrictions: 1 <= num <= 1000

- **Number of tests Passed**
  1 test file run, with 6 true and prompt-based cases (all passed).

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 100%

```
Name                      Stmts   Miss  Cover
---------------------------------------------
int_to_mini_roman.py         11      0   100%
test_int_to_mini_roman.py     9      0   100%
---------------------------------------------
TOTAL                        20      0   100%
```

  - Branch Coverage: 100%

```
Name                  Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------
int_to_mini_roman.py     11      0      6      0   100%
-------------------------------------------------------
TOTAL                    11      0      6      0   100%
```

- **Interpretation**
  All code paths, including outer loop and Roman numeral conversion logic (with edge values like 4, 9, 944, 1000), are fully covered by the benchmark test suite, validating the conversion process for a diverse set of test cases.

6. **HumanEval/160**

Base prompt:

def do_algebra(operator, operand): Given two lists operator and operand. The first list has basic algebra operations, and the second list is a list of integers. Use the two given lists to build the algebraic expression and return the evaluation of this expression.

Example:
operator=['+', '*', '-']
array=[2, 3, 4, 5]
result = 2 + 3 * 4 – 5 => result = 9
Note: The length of operator list is equal to the length of operand list minus one. Operand is a list of non-negative integers. Operator list has at least one operator, and operand list has at least two operands.

- **Number of tests Passed**
  1 test file run, with 5 true and prompt-based cases (all passed).

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 85%

```
Name               Stmts   Miss  Cover
---------------------------------------
do_algebra.py         13      2    85%
test_do_algebra.py     8      0   100%
---------------------------------------
TOTAL                 21      2    90%
```

  - Branch Coverage: 81%

```
Name           Stmts   Miss Branch BrPart  Cover
------------------------------------------------
do_algebra.py     13      2      8      2    81%
------------------------------------------------
TOTAL             13      2      8      2    81%
```

- **Interpretation**
  Most paths and logical branches for expression construction and evaluation are covered; however, there are some error-handling code paths such as invalid operator checks or input validation exceptions that are not triggered by the current benchmark test suite. The missed coverage likely relates to exception handling for invalid input scenarios, indicating that tests for input validation and unsupported operator cases would increase coverage and robustness.

7. **HumanEval/159**

Base prompt:

def eat(number, need, remaining):

You're a hungry rabbit, and you already have eaten a certain number of carrots, but now you need to eat more carrots to complete the day's meals.

You should return an array of [total number of eaten carrots after your meals, the number of carrots left after your meals]

If there are not enough remaining carrots, you will eat all remaining carrots, but will still be hungry.

- **Number of tests Passed**
  1 test file run, with 4 true and prompt-based cases (all passed)

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 100%

```
Name         Stmts   Miss  Cover
--------------------------------
eat.py           4      0   100%
test_eat.py      7      0   100%
--------------------------------
TOTAL           11      0   100%
```

  - Branch Coverage: 100%

```
Name     Stmts   Miss Branch BrPart  Cover
----------------------------------------
eat.py       4      0      2      0   100%
----------------------------------------
TOTAL        4      0      2      0   100%
```

- **Interpretation**
  All code lines and both branches (need <= remaining and need > remaining) for this function are exercised by the benchmark test suite, indicating comprehensive coverage and correct result handling for available and insufficient carrot cases.

8. HumanEval/139

Base prompt:

```
def special_factorial(n):
    The Brazilian factorial is defined as:
    brazilian_factorial(n) = n! * (n-1)! * (n-2)! * ... * 1!
    where n > 0
    For example:
    >>> special_factorial(4) =  288
```

The function will receive an integer as input and should return the special factorial of this integer.

- **Number of tests Passed**
  1 test file run, with 5 benchmark cases (all passed)

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 89%

```
Name                    Stmts   Miss  Cover
---------------------------------------------
special_factorial.py        9      1    89%
test_special_factorial.py   8      0   100%
---------------------------------------------
TOTAL                      17      1    94%
```

  - Branch Coverage: 85%

```
Name                  Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------
special_factorial.py      9      1      4      1    85%
-------------------------------------------------------
TOTAL                     9      1      4      1    85%
```

- **Interpretation**
  Most of the function's execution paths are covered, particularly for valid inputs (n>=1) . However, coverage does not include the error-checking branch for invalid argument values (n<1), as the benchmark suite does not test for exceptions. This indicates robust correctness for the standard domain, but missing coverage for input validation and error handling.

9. **HumanEval/141**

Base prompt:

def file_name_check(file_name):Create a function which takes a string representing a file's name, and returns 'Yes' if the file's name is valid, and returns 'No' otherwise.

A file's name is considered to be valid if and only if all the following conditions are met:
- There should not be more than three digits ('0'-'9') in the file's name.
- The file's name contains exactly one dot '.'.
- The substring before the dot should not be empty, and it starts with a letter from the latin alphabet ('a'-'z' and 'A'-'Z').
- The substring after the dot should be one of these: ['txt', 'exe', 'dll']

Examples:
file_name_check("example.txt") # => 'Yes'
file_name_check("1example.dll") # => 'No'

- **Number of tests Passed**
  1 test file run, 8 benchmark cases (all passed).

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 100%

```
Name                    Stmts   Miss  Cover
-------------------------------------------
file_name_check.py         11      0   100%
test_file_name_check.py    11      0   100%
-------------------------------------------
TOTAL                      22      0   100%
```

  - Branch Coverage: 100%

```
Name                 Stmts   Miss Branch BrPart  Cover
------------------------------------------------------
file_name_check.py      11      0      8      0   100%
------------------------------------------------------
TOTAL                   11      0      8      0   100%
```

- **Interpretation**
  All logic branches, input validations, and error conditions for file name rules are exercised by the provided test suite. The tests confirm proper handling of edge cases like wrong extension, excess digits, missing separator, and empty names.

10. HumanEval/151

Base prompt:

def double_the_difference(lst):

Given a list of numbers, return the sum of squares of the numbers in the list that are odd. Ignore numbers that are negative or not integers.

double_the_difference([1, 3, 2, 0]) == 1 + 9 + 0 + 0 = 10
double_the_difference([-1, -2, 0]) == 0
double_the_difference([9, -2]) == 81
double_the_difference([0]) == 0
If the input list is empty, return 0.

- **Number of tests Passed**
  1 test file run, 7 benchmark cases (all passed).

- **Line and Branch Coverage** (Output copied from terminal)
  - Line Coverage: 100%

```
Name                           Stmts   Miss  Cover
--------------------------------------------------
double_the_difference.py           2      0   100%
test_double_the_difference.py     10      0   100%
--------------------------------------------------
TOTAL                             12      0   100%
```

  - Branch Coverage: 100%

```
Name                       Stmts   Miss Branch BrPart  Cover
------------------------------------------------------------
double_the_difference.py       2      0      0      0   100%
------------------------------------------------------------
TOTAL                          2      0      0      0   100%
```

- **Interpretation**
  All conditional paths including type checking, positive/odd checks, and rejection of invalid values are exercised by the benchmark test suite. The results confirm thorough coverage for positive, negative, float, zero, and boolean edge cases.

Results Summary Table

Problem	Line Cov (%)	Branch Cov(%)	Notes
HumanEval/155	100	100	All even/odd digit cases, negatives, and zero tested.
HumanEval/101	100	100	Handles commas/spaces and all split logic.
HumanEval/132	100	100	Handles nested brackets
HumanEval/136	100	100	Handles no negative/positive, boundary checks.
HumanEval/156	100	100	All roman numeral thresholds and edge cases.
HumanEval/160	85	81	Missed error checks for bad operator/invalid arg length.
HumanEval/159	100	100	All carrot remaining/hungry branches tested.
HumanEval/139	89	85	Missed check for $n < 1$ branch in input validation.
HumanEval/141	100	100	All file/digit/extension validation branches covered.
HumanEval/151	100	100	Filters positive odd ints, non-int/zero/float/bool.

Part 2: LLM-Assisted Test Generation & Coverage Improvement

1. **HumanEval/160**

- Prompt Used (GPT5):
  Please analyze the function do_algebra and its test file below. The coverage report shows that some error-handling branches are not tested. Specifically:
  - The function returns a ValueError if the operator list includes an unsupported operator
  - It also returns a ValueError if the length of operator does not match len(operand) - 1
  Generate new Python unit tests that cover these error cases.
  Return only the new test functions, ready to append to the test_do_algebra.py file and avoid duplicate tests.
- Coverage Before (Benchmark tests):
  - Line Coverage: 85%
  - Branch Coverage: 81%
  - Stmts Missed: 2 of 13; Branches Missed: 2 of 8
- Coverage After (New Tests):
  - Line Coverage: 100%
  - Branch Coverage: 100%
  - All statements and branches exercised

```
Name          Stmts   Miss Branch BrPart  Cover
-----------------------------------------------
do_algebra.py     13      0      8      0   100%
-----------------------------------------------
TOTAL             13      0      8      0   100%
```

- What Changed
  - Added new unit tests that trigger the error-handling branches for invalid operator and length mismatches.
  - These new tests successfully exercised the previously untested code paths, achieving full line and branch coverage.
  - All coverage improvement came from covering exception scenarios not represented in the original benchmark tests.
- Redundancy Note
  - The LLM produced distinct, non-duplicate tests directly targeting missing branches.
  - No duplicate or near-duplicate test cases were generated, each addition measurably increased coverage.

  Branch coverage reached 100% after the first round of LLM-generated tests. The convergence criterion (Coverage(i) - Coverage(i-2) ≤ 3%) is met. Therefore, further iterations were not required for this problem.

2. HumanEval/139

- Prompt Used (GPT5):
  Please analyze the function special_factorial and its test file below. The coverage report shows that some error-handling branches are not tested. Specifically:
  - The function raises a ValueError if n < 1.
  Generate new Python unit tests that cover this error case.
  Return only the new test functions, ready to append to the test_special_factorial.py file and avoid duplicate tests.
- Coverage Before (Benchmark tests):
  - Line Coverage: 89%
  - Branch Coverage: 85%
  - Stmts Missed: 1 of 9; Branches Missed: 1 of 4
- Coverage After (New Tests):
  - Line Coverage: 100%
  - Branch Coverage: 100%
  - All statements and branches exercised

```
Name                 Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------
special_factorial.py     9      0      4      0   100%
-----------------------------------------------------
TOTAL                    9      0      4      0   100%
```

- What Changed
  - Added a new unit test that triggers the error-handling branch for input n < 1, specifically verifying that ValueError is raised when calling special_factorial(0) or negative numbers.
  - This test successfully exercised the previously untested exception path, bringing coverage to 100%.
  - Improvement is entirely due to now covering the ValueError logic which was not exercised by the benchmark suite.
- Redundancy Note
  - The LLM-generated test directly targeted the missing error branch.
  - No duplicate or near-duplicate tests were produced, each addition added unique value to coverage.

Branch coverage reached 100% after the first round of LLM-generated tests. The convergence criterion (Coverage(i) - Coverage(i-2) ≤ 3%) is met. Therefore, further iterations were not required for this problem.

Part 3: Fault Detection Check

1. **HumanEval/160**

- Bug injected (do_algebra.py):
  The part of the code responsible for building the expression string and allowing Python to handle operator precedence was replaced with a left-to-right loop that evaluates the expression sequentially (using $result = eval(f"\{result\}\{op\}\{val\}")$ each time). This forces left-to-right grouping and ignores proper mathematical precedence, a common mistake when implementing calculators. For example, instead of evaluating $2 + 3 * 4 - 5$ as $2 + (3 * 4) - 5$, it evaluates as $((2 + 3) * 4) - 5$.

- Test Results:
  - The test function failed as soon as it ran the first test.

```
assert do_algebra(['+', '*', '-'], [2, 3, 4, 5]) == 9
AssertionError: assert 15 == 9

Name              Stmts   Miss  Cover
---------------------------------------
do_algebra.py        11      0   100%
test_do_algebra.py   16      4    75%
---------------------------------------
TOTAL                27      4    85%
```

- Conclusion:
  High branch and line coverage ensured that the test suite exercised a variety of operator and operand cases. However, it was the explicit inclusion of a test for mixed precedence (+,*,-) that exposed the bug. This shows that while coverage ensures code is run, strong, targeted test cases are what make effective fault detection possible.

2. **HumanEval/139**

- Bug injected (special_factorial.py):
  The loop range was changed from range(1, n + 1) to range(1, n) , causing the code to omit the final factorial multiplication (product only goes up to (n-1)! instead of n!). This off-by-one bug is realistic because forgetting to include an endpoint in a loop is a common error, given its exclusive upper-bound behavior.

- Test Results:
  - The test function failed:

```
def test_examples():
    assert special_factorial(1) == 1
    assert special_factorial(2) == 2   ---- Fails here, got 1 instead of 2
    ...
AssertionError: assert 1 == 2

Name                          Stmts   Miss  Cover
---------------------------------------------
special_factorial.py              9      0   100%
test_special_factorial.py        13      3    77%
---------------------------------------------
TOTAL                            22      3    86%
```

- Conclusion:
  The high coverage ensured the code for different valid inputs was executed, but it was the suite's direct checks of small n values like 2 and 3 that immediately exposed the off-by-one error. This confirms that robust, targeted value checks are necessary for effective fault detection, even when overall coverage is high.