# Code Complexity and Coverage Analysis

**Team Members**:
Herdi Saleh, Ahmed Yossef, Maxim Kvasov,
Kim Woojin, Annika Ekholm

February 21, 2025

## 1 Introduction

### 1.1 Project and Functions

For this project, we worked with **Pandas**, a widely used Python library for data manipulation and analysis. We aimed to analyse and increase coverage of the following high-complexity functions in the library:

- **quantile()**: Computes the quantile of grouped data based on specified interpolation methods, handling various data types and missing values efficiently. (Ahmed)
- **apply()**: Allows applying custom functions to each group in a grouped dataset, providing flexibility for complex transformations. (Herdi)
- **shift()**: Shifts values within each group by a specified number of periods, useful for time series analysis and lagging operations. (Ahmed)
- **_evaluate_numexp()**: Evaluates numerical expressions efficiently, leveraging optimized computation to improve performance for large datasets. (Kim)
- **to_time()**: Converts data into time-related formats, ensuring correct handling of datetime values and time zone adjustments. (Annika)

### 1.2 Onboarding

#### 1.2.1 Project Build Process

The onboarding documentation for the project is well-structured, and the build process generally works as documented. Most dependencies can be installed using standard package managers such as `pip`, `venv`, and `conda`, and the build was concluded without major errors. However, running `coverage.py` on Pandas posed a significant challenge due to a self-referencing dependency issue, where Pandas depends on itself, causing installation conflicts. This issue persisted

across multiple environments, including `pip`, virtual environments, and `conda`, and the only viable workaround was using Docker. However, Docker installation introduced additional complications, particularly on Apple M1 chips, where compatibility and performance issues required further troubleshooting. Despite this, the project's example tests ran successfully. While the documentation effectively guided the standard build process, it did not address the `coverage.py` issue, which required independent problem-solving.

#### 1.2.2 Project Continuation Decision

Despite the challenges faced with running `coverage.py`, the overall onboarding experience has been smooth. The documentation is clear for standard usage, and the project structure is well maintained. Given the usefulness of the project and our ability to work around the encountered issues, we plan to continue with this project rather than switching to another one.

### 1.3 Tools

- **Lizard**
  Lizard was used to analyze the cyclomatic complexity (CC) of the target functions. It provided automated measurements for metrics such as lines of code (LOC), complexity (CCN), token count, and function length. The tool ran without issues and produced consistent results.

- **Coverage.py**
  Coverage.py was used to assess code coverage. However, running Coverage.py on Pandas was problematic due to a self-referencing dependency issue where Pandas requires itself during installation. The issue was resolved by either using Docker or modifying local directories to work around the dependency conflict.

- **Manual Instrumentation**
  To manually measure branch coverage, we implemented instrumentation by inserting branch tracking counters at all decision points in the function. This was achieved by defining a dictionary structure to store execution counts for each branch. Each branch was assigned a unique identifier, and before executing a conditional statement, we recorded whether that branch was taken. At the end of execution, the collected coverage data was printed to the console for analysis, with an optional capability to write it to an output file. This method allowed us to measure branch execution independently and resulted in a higher reported coverage percentage compared to automated tools.

  Our instrumentation explicitly measures **branch coverage**, whereas Coverage.py, by default, evaluates **statement coverage**. This distinction likely accounts for the observed discrepancies between the two approaches, as branch coverage considers whether all possible paths within a function

are executed, while statement coverage only verifies whether individual lines of code were run.

# 2 Quantile Function Analysis

## 2.1 Complexity Measurement

The automated complexity measurement using Lizard provides the following results:

- **Pre-processing (pre_processor)**: CCN = 13, LOC = 28
- **Post-processing (post_processor)**: CCN = 8, LOC = 36
- **Computation Logic (blk_func)**: CCN = 8, LOC = 40
- **Quantile Function (quantile)**: CCN = 3, LOC = 38

Manually calculated cyclomatic complexity based on independent paths confirms the Lizard results. Each conditional branching structure increases the number of paths, leading to a total complexity score consistent with the automated report.

**Independent Paths Breakdown**:

- **Main quantile() Function**: 4 paths from branching on input conditions.
- **Pre-processing (pre_processor())**: 9 paths due to multiple type-checking conditions.
- **Computation Logic (blk_func())**: 12 paths from loops and missing value handling.
- **Post-processing (post_processor())**: 6 paths from type inference and formatting decisions.

The final complexity score is calculated as:

$$CC = (\text{Total Independent Paths}) + 1 = (4 + 9 + 12 + 6) + 1 = 32$$

## 2.2 Manual Instrumentation for Coverage

To manually measure coverage, we inserted branch tracking counters at all decision points. The collected data was stored in a dictionary structure and written to an output file at the end of execution. Our manual coverage resulted in **86.72%** coverage, compared to **82%** from Coverage.py.

## 2.3 Coverage Improvement

Additional test cases were implemented to improve coverage, ensuring edge cases and untested branches were validated:

- **test_quantile_mixed_dtypes**: Ensures proper handling of mixed integers and floats.
- **test_quantile_empty_dataframe**: Ensures function behavior with empty input.
- **test_quantile_single_row_group**: Tests when a group has only one row.
- **test_quantile_large_dataframe**: Ensures performance with large datasets.
- **test_quantile_all_zeroes**: Ensures correct output when all values are zero.
- **test_quantile_multiindex_unbalanced_groups**: Tests MultiIndex DataFrames.

### 2.4 Refactoring Plan and Pseudocode

To improve maintainability, the function was refactored into modular components:

- **Removed Redundant Nesting in `post_processor()`**: The function structure was simplified by flattening nested conditionals and consolidating type handling.
- **Early Exits to Reduce Unnecessary Checks**: If `inference` is `None`, the function now exits immediately, avoiding redundant operations.
- **Refactored `blk_func()` to Have Cleaner Mask and Reshape Logic**: The handling of `BaseMaskedArray` masks and `result_mask` was streamlined into a single section for clarity.
- **Localized `is_datetimelike` Handling in `blk_func()`**: The conversion of datetime values using `vals.view("i8")` is now performed exactly where needed, avoiding unnecessary operations.
- **Standardized Warnings Handling for Dtype Conversions**: Used `warnings.catch_warnings()` consistently to manage dtype conversion warnings efficiently.
- **Restructured Quantile Computation Call (`func()`) to Be More Modular**: The quantile function call was organized into a dedicated structure, improving maintainability and reducing duplicate logic.

## 3 Apply Function Analysis

### 3.1 Complexity Measurement

The automated complexity measurement using Lizard provides the following results:

- **Apply function (apply)**: CCN = 10, LOC = 163 (28 without comments)

Calculating the cyclomatic complexity by hand, however, gives us a result of 9, which was confirmed by multiple people in the group.

## 3.2 Manual Instrumentation for Coverage

The result from *Coverage.py* showed a branch coverage of **70**% before any improvements were made. The manual instrumentation, done by marking each branch on hit and comparing them to the total count of branches, gave us a coverage of **66.67**%. Interestingly, the amount of branches marked by the manual instrumentation is 12, whereas *Coverage.py* shows 14 branches. This could be due to calculating every *if*-statement with two and an *or* as two branches.

## 3.3 Coverage Improvement

To improve the coverage four addtional tests were implemented, primarily to trigger the branches that were not covered. The tests were:

- **test_passing_args_to_property**: Ensure that a property cannot be passed any arguments.
- **test_apply_with_property**: Ensure that the property of a dataframe may be used as a function in *apply*.
- **test_uncallable_string_as_function**: Ensure that we cannot pass a string to apply that is not a property of the object.
- **test_func_callable_given_args**: Ensure that a function must be callable if arguments are given.

After adding these test cases, the manual instrumentation showed a branch coverage of **100**%, which was also the result from *Coverage.py*.

## 3.4 Refactoring Plan

The *apply* function is divided into three main parts in the second conditional, the *if-elif-else*. The first condition checks whether the function is a string, and handles it as a property accordingly. The second condition checks whether arguments are passed into the function, leading us to either wrap the function or throw an error if the function is not callable. The last condition, the *else*, simply allows the function to run.

To refactor this, we can extract the logic of handling a property as a function and handling a functions with provided args, creating two new helper functions. Doing this, we lower the cyclomatic complexity from 10 to 5 according to *Lizard*.

Refactoring was done, and the results can be show through `git diff` between the following two hashes:

Before refactoring: `a95c53ec288d3f7dc4595a6f72cc02685783d73e`

After refactoring: `352d5b0a5f0d4e8bacfb38abdd63947ac90e406d`

# 4 Shift Function Analysis

## 4.1 Complexity Measurement

According to lizard, the CCM of the shift function is 13, NLOC is 64 and token count is 345.

If we calculate the cyclomatic complexity by hand, we get a discrepancy in our results.

In our source code we have:

- 12 if statements
- 1 loop
- 1 return statement
- 1 entry point

If we then apply the CCN formula to this data, we get $12 + 1 - 2 = 11$

## 4.2 Manual Instrumentation for Coverage

The manual instrumentation was implemented using a technique similar to how it is done in section 5.2. We have a larger parent class and a function defined in it which we call every time we enter a logical branch during execution. These are used in tandem with arrays that are outside of the

## 4.3 Coverage Improvement

Before making any changes to the $shift()$ function, our test coverage percentage was 76.92 % After adding the test cases for covering branches 2 and 6, our test coverage went up to 100 %. The branches are the ones responsible invalid

Upon adding manual instrumentation we get

## 4.4 Refactoring Plan

At the end of the function there was a conditional return which needed to be unrolled in order to include instrumentation.

The second refactoring that was implemented was

# 5 Numexpr Evalution Function Analysis

**Purpose**

- Attempts fast computation using numexpr
- Handles reversed operations (e.g., radd, rsub)
- Falls back to standard Python evaluation if necessary

## 5.1 Complexity Measurement

The complexity analysis of _evaluate_numexpr yielded consistent results across automated and manual measurements:

- **Lizard Tool Results**:
  - NLOC: 28
  - CCN: 9
  - Token count: 147
  - Parameter count: 4
- **Manual Calculation** (view calculation):
  - Nodes: 18
  - Edges: 25
  - CCN = E - N + 2 = 25 - 18 + 2 = 9

The complexity is influenced by multiple decision points, including exception handling paths and conditional logic in the function's optimization routines.

## 5.2 Manual Instrumentation for Coverage

Source code on GitHub

We implemented a branch tracking system with the following characteristics:

- **Tracking Mechanism**:
  - Dictionary-based branch execution counter
  - Support for if/else conditions, exceptions, and nested logic
  - Entry/exit point monitoring

## 5.3 Coverage Improvement

There was no test case for this function. **Thus, the coverage was 0%.** Initial coverage report (0%) showed no test cases existed.

Implemented comprehensive test cases targeting various scenarios to improve coverage: Coverage improvement discussion and test cases documentation detail our approach.

- **Basic Operations**:
  - Addition: [1,2,3] + [4,5,6] = [5,7,9]
  - Subtraction: [1,2,3] - [4,5,6] = [-3,-3,-3]
  - Multiplication: [1,2,3] * [4,5,6] = [4,10,18]
  - Division: [1,2,3] / [4,5,6] = [0.25,0.4,0.5]
- **Special Cases**:
  - Reversed operations
  - Mixed data types (int/float/boolean)
  - Array size variations
  - Edge cases (NaN, infinity)
- **Coverage Results**:
  - Manual Tool: 50.00% (6/12 branches)
  - Coverage.py: 50.00% (8/16 branches)

## 5.4 Refactoring Plan

Proposed improvements to reduce complexity and enhance maintainability:

### 5.4.1 Current Complexity Points (CCN=9)

The high cyclomatic complexity stems from multiple decision points:

1. **Initial Validation Check**
   - Verifies if numexpr can be used
   - First branching point in control flow
2. **Operation Type Check**
   - Determines if operation is reversed
   - Affects operand ordering
3. **Exception Management**
   - Try-except block structure
   - Introduces multiple paths
4. **Multiple Exception Handlers**

- TypeError handling
- NotImplementedError handling

5. **Fallback Logic**
   - Boolean arithmetic fallback check
   - Additional branching path
6. **Test Mode Verification**
   - Check for test mode status
   - Result storage decision
7. **Result Validation**
   - Final null check
   - Determines fallback necessity

#### 5.4.2 Proposed Component Restructuring

To reduce complexity, the function can be decomposed into four main components:

**Input Validation Component**

- Primary responsibility: numexpr compatibility check
- Single entry point for validation logic
- Clear prerequisites verification

**Operand Preparation Component**

- Handles reversed operation logic
- Manages operand ordering
- Ensures consistent input format

**Core Evaluation Component**

- Performs numexpr computation
- Focuses on primary operation
- Maintains clean evaluation logic

**Error Management Component**

- Centralizes exception handling
- Manages fallback scenarios
- Provides consistent error processing

# 6 To-Time Function Analysis

## 6.1 Complexity Measurement

Automated complexity analysis of the function (prior to coverage extension and refactoring) using Lizard gave the following results:

- **_convert_listlike:** CCN = 16, LOC = 48 *(Nested help function)*
- **to_time:** CCN = 7, LOC = 21

Manual complexity analysis, calculated as $\{\#\text{ of Decision Points}\} + 1$ provided the same result as Lizard for both **to_time** and **_convert_listlike**: 7 and 16 respectively.

## 6.2 Coverage Metrics

The manual coverage instrumentation, implemented as described in Section 1.3, resulted in **76%** coverage estimation, compared to **81%** as reported by Coverage.py.

## 6.3 Coverage Improvement

Guided by specific unaccessed branches in the function, the following test cases were introduced to improve coverage:

- **test_None_arg**: Tests that `arg=None`, returns `None`.
- **test_time_arg**: Tests that for `arg` of `'datetime.time'` object type, `arg` is returned.
- **test_matrix_arg**: Tests for expected TypeError when `arg` is a multidim numpy array.
- **test_coerce_error**: Tests that with `errors='coerce'`, ValueError is suppressed and `None` is returned for unparsable input string.

According to manual instrumentation, coverage was improved to **94%**. According to Coverage.py, the improved coverage is **86%**. The extended coverage test suite is found here.

## 6.4 Refactoring Plan

Proposed refactoring to reduce the function's complexity and enhance maintainability:

- **to_time**
  - Extract nested function **_convert_listlike**, similar to the existing help function **_guess_time_format**.
- **_convert_listlike**

- Modularise into smaller functions in order to scale down on the many nested conditionals.
  - **_parse_with_format**: Given format info in input, parse and return the time.
  - **_infer_and_parse**: Lacking format info, infer format from the _time_formats array. Parse and return the time.
  - **_try_parse_time**: Given a format, try to parse a time with it. Utility function for **_infer_and_parse**.
  - **_handle_error**: Used to raise error if the `errors` flag is set to `raise`, otherwise appends `None` to the current times array.

# 7 Way of Working

Our team currently occupies the stages of Formed and Collaborating in the Essence framework. Responsibilities and tasks have been assigned and adhered to with some efficiency, and efficient collaboration and discussion can take place when needed.

Some confusion arose with regards to the actual assignment criteria, which may have been resolved earlier with improved communication. Due to to this project's more modular nature, we did end up lacking a set standard for our work until the late phases.