Assignment 4: Issue Resolution

DD2480

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

March 7, 2025

1 Introduction

1.1 Project

For this project, we worked with **Pandas**, a widely used Python library for data manipulation and analysis. In our choice of issue within the project, we supplemented its existing `Period` class with additional functionality in parsing input strings.

The repo containing our work is found **here**, where the **main** branch features implementation of additional functionality alongside an updated test suite, and the **testing** branch keeps the original functionality with the updated test suite for ease of testing.

1.2 Onboarding

In spite of considerable complications in setup, we chose to continue working with Pandas, as we in Assignment 3.

While onboarding documentation for the project is well-structured, and the build process generally works as documented, compiling Pandas such modules like Pytest could be successfully run posed a significant challenge due to a self-referencing dependency issue, where Pandas ends up depending 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.

While in Assignment 3, we had to resort to ad hoc solutions in order to be able to utilise coverage.py and Pytest, we were during Assignment 4 (with considerable

time spent) able to set up fully functioning testing environments for most team members.

1.3 Contributions

Roles Assigned

- Ahmed: Implementation
- Herdi: Implementation
- Annika: Testing
- Kim: Testing
- Maxim: Testing

Effort Spent

	plen. meet.	disc.	read doc	config/setup	analyze code	write doc	write code	run code
Ahmed	1	3.25	3	2	2	3	4	3
Annika	1	3.25	3	2.5	1	4	4	3
Herdi	1	3.25	3	3	3	2	5	0.5
Kim	1	3.25	4	3	1	2.5	4	3
Maxim	1	3.25	3	5	1	2	4	1

Table 1: Time spent for each team member (hours)

Notes on specific activities

Meetings: One plenary meeting was held when finalising our decision on what issue to work with. Outside of this, five meetings were held within subsets of the team.

Reading documentation: In choosing an issue in a project of this size, it was unavoidable that a considerable amount of time had to be spend perusing the issue tracker. Existing documentation naturally had to be read and considered in designing and implementing planned functionality.

Configuration and setup: As mentioned in Section 1.2, setting up functioning environments in which the project could be compiled and tested for all team members proved to be a time sink. Specifically, time was spent resolving circular dependencies, incompatibilites with Docker, Unix/Windows conflicts, as well as conflicts with PyQt. Problems with the latter remained unresolved, and a small number of tests are excluded from regression testing due to it.

Running and compiling code: Due to the size of the project, time spent running and compiling code was not insignificant. Running the full test suite takes approximately an hour, and had to be done more times than was optimal due to the aforementioned conflicts.

2 Issue Resolution

2.1 Overview

Title: ENH: Improve `Period` parsing

URL: https://github.com/pandas-dev/pandas/issues/48000

The issue requests extended and/or altered functionality of string parsing when creating a `Period` object solely from a string representation. There are eight categories of dates that cannot be created from string representations alone, summarised as requirements in Section 2.2. The relevant code is located in the `Period` class definition in `/pandas/_libs/tslibs/period.pyx`

2.2 Requirements

1. **Business years not going from Jan - Dec:**
   Implement support for creating **non-standard business year** (i.e. not from Jan-Dec) representations with the `Period` class from string representation alone.

2. **Quarters starting with a different month than January/April/July/October:**
   Implement support for creating **non-standard quarter** (i.e. not starting in January, April, July, or October) representations with the `Period` class from string representation alone.

3. **Weeks:**
   Implement support for creating **week** representations with the `Period` class from string representation alone.

4. **Business days:**
   Implement support for retrieving a `Period` representation of all business days in a given range from string representation alone.

5. **Hours:**
   In the current version of Pandas, hours on the format `hh:mm` is misinterpreted as minutes. Implement support for parsing of hours.

6. **Weeks in the 60s, 70s, 80s or 90s of any century**
   Currently, weeks in the 60s, 70s, 80s or 90s of any century represented by a `Period` object cannot be recreated from their own string representations, and support for this should be implemented.

7. **Weeks from the 24th century onwards:**
   Currently, weeks from the 24th century onwards represented by a `Period` object cannot be recreated from their own string representations, and support for this should be implemented.

8. **ISO 8601 ordinal dates:**

Implement support for parsing **ISO 8601 ordinal dates** (i.e. YYYY-DDD) with `Period` from string representation alone.

Additionally, we noted another missing functionality and added the following requirement.

9. **Multi-Year Spans**
   Implement support for representations of **multi-year spans** and **quarter-based multi-year periods** with `Period` from string representation alone.

Of the requirements above, **3**, **6**, **7**, **8**, and **9** were chosen to be implemented for partial resolution of the issue. **1**, **2**, and **4** are left for future development, and **5** was deemed to be a matter of design rather than missing functionality, as hours can already be represented on a `HH:MM:SS` or `HHh` format without ambiguity.

2.3 Test Cases

The following test functions were added to the existing testing suite, located in `pandas/tests/tslibs/test_period.py`:

- `test_period_with_ordinal_dates`:
  Verifies that ISO 8601 ordinal dates (YYYY-DDD format) are correctly parsed. The test checks basic ordinal dates throughout a normal year and proper handling of leap years. Tests help ensure correct implementation of **requirement 8**.
- `test_period_with_24th_century_weeks`:
  Verifies that week-based Period strings (YYYY-MM-DD/YYYY-MM-DD format) are correctly parsed, especially for dates in the 24th century. The weekly periods in the 24th century and weeks that cross year boundaries are tested. Tests help ensure correct implementation of **requirement 7**.
- `test_problematic_years_roundtrip`:
  Verifies that Period object with years that could be misinterpreted as times can be correctly converted to strings and then parsed back to Period object. Previously, the year 2060s - 2090s were misinterpreted as 20:60 - 20:90. The test checks for the string conversion roundtrip for years in these periods. Tests help ensure correct implementation of requirement **6**.
- `test_period_parse_weeks_positive`:
  Checks for expected attribute values for `Period` objects created from the dedicated YYYYMMDD-YYYYMMDD week format with **valid** inputs. Tests help ensure correct implementation of **requirement 3**.
- `test_period_parse_weeks_equivalent`:
  Checks that string representation of objects created with the YYYYMMDD-YYYYMMDD format can successfully be **fed back into** `Period` to create the equivalent week period. Tests help ensure correct implementation of **requirement 3**.

- `test_period_parse_weeks_err`:
  Checks that the expected errors are thrown when attempting to create a Period objects from the string format YYYYMMDD-YYYYMMDD with **invalid** inputs. Tests help ensure correct implementation of **requirement 3**.
- `test_old_and_retro_periods_errors` This test checks if we araise the errors that ought to be raised when we get inputted an incorrectly formatted sequence that tries to follow the YYYYMMDD-YYYYMMDD format. This checks for requirement **3**.
- `test_old_and_retro_periods` This test checks if one-week long period can be initialised through the t YYYYMMDD-YYYYMMDD format but also whether they can span multiple years. This checks for requirements **3**, **6** and **9**. We test both decade switches and the formatting.
- `test_period_parse_quarters_retro` This test checks that we can initalise a priod using the YYYYQA-YYYYQB format. This meets the requirements **6**, **9**.
- `test_old_and_retro_periods_line_formatted_reinitialised` This test checks that we can re-initialise an Period object from it's string representation. This fulfills the requirements **3**, **6**, **9**.

2.4 Code Changes

In this section the code changes will be shown for each of the implemented features.

Ordinal dates

```
# Located in `period.pyx`
# In method `Period.__new__()`
class Period():
    ...
    # GHF#2
    if isinstance(value, str):
        # Pre-processing for ISO8601 ordinals
        # Converts `yyyy-ddd` or `yyyyddd` to `yyyy-mm-dd`
        # for the parser to then handle as usual
        match = re.search(r"^(\d{4}-?\d{3})( .+)?\$", value)
        if match:
            date, rest = match.groups()
            formatting = "\%Y-\%j" if '-' in date else "\%Y\%j"
            processed_date = datetime.strptime(date, formatting).
                                                strftime("\%Y-\%m-\%d
                                                ")
            value = processed_date + (rest if rest is not None else
                                      "")
...
```

This change converts `yyyy-ddd` formatted dates to `yyyy-mm-dd` format, so the parser can handle them correctly.

Week spans

```
# Located in ‘period.pyx‘
# In method ‘Period.__new__()‘
class Period():
    ...
    elif re.search(r"^\d{4}-\d{1,2}-\d{1,2}/\d{4}-\d{1,2}-\d{1,2}",
                                        value) or re.search(r"^\d{8}
                                        -\d{8}", value):
        # GHF#3
        # Case that cannot be parsed (correctly) by our datetime
        # parsing logic
        # 1. ‘yyyy-mm-dd/yyyy-mm-dd‘
        # 2. ‘yyyymmdd-yyyymmdd‘
        dt, freq = _parse_weekly_str(value, freq)
    else:
        dt, reso = parse_datetime_string_with_reso(value, freqstr)
...
# Outside of class Period
cdef _parse_weekly_str(value, BaseOffset freq):
    """
    Parse e.g. "2017-01-23/2017-01-29", which cannot be parsed by
                                        the general
    datetime-parsing logic.  This ensures that we can round-trip
                                        with
    Period.__str__ with weekly freq.
    """
    # GH#50803
    # GHF#2
    start, end = value.split("/") if ’/’ in value else value.split(
                                        "-")
    start = Timestamp(start)
    end = Timestamp(end)

    if (end - start).days != 6:
        # We are interested in cases where this is str(period)
        #  of a Week-freq period
        raise ValueError("Could not parse as weekly-freq Period")

    if freq is None:
        day_name = end.day_name()[:3].upper()
        freqstr = f"W-{day_name}"
        freq = to_offset(freqstr, is_period=True)
        # We _should_ have freq.is_on_offset(end)

    return end, freq
```

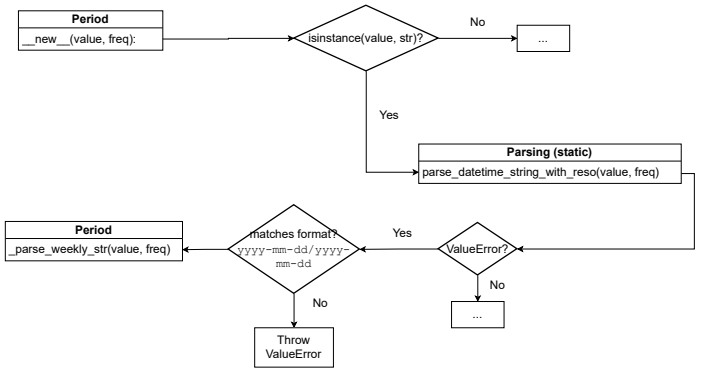
This function checks that the date string is a week span format, and parses this using an already existing function, that was extended to support dates of format `yyyymmdd-yyyymmdd` as well.

Multi-year spans

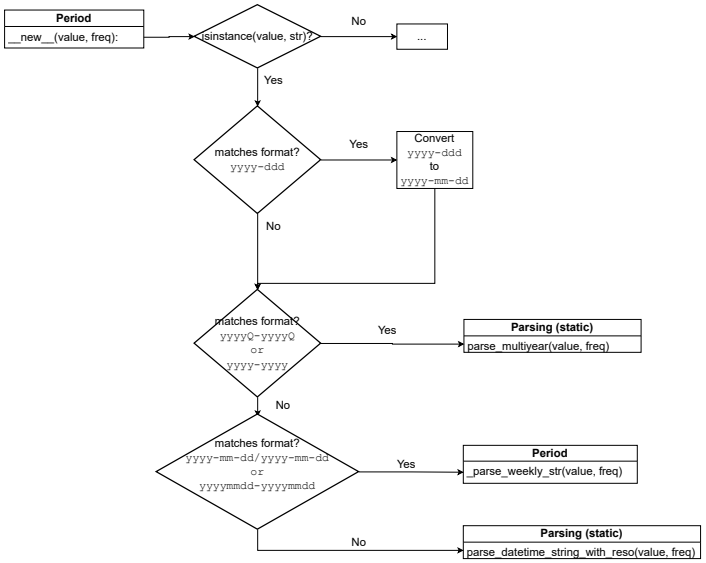
```
# Located in `period.pyx`
# In method `Period.__new__()`
class Period():
    ...
    # Parse multiyear spans
    # GHF#4
    multiyear = parse_multiyear(value, freq)
    if multiyear is not None:
        return multiyear
    elif ...
# Located in `parsing.pyx`
# GHF#4
def parse_multiyear(time_str, freq=None):
    """
    Extended parsing logic to handle:
    1. Multi-year spans (e.g., "2019-2021").
    2. Multi-quarter spans (e.g., "2019Q1-2021Q4").
    """
    # Handle Multi-Year Spans (e.g., "2019-2021")
    multi_year_match = re.match(r"^(\d{4})-(\d{4})$", time_str)

    # Handle Multi-Quarter Spans (e.g., "2019Q1-2021Q4")
    multi_quarter_match = re.match(r"^(\d{4}Q[1-4])-(\d{4}Q[1-4])$"
                                   , time_str)

    if multi_year_match:
        start_year, end_year = map(int, multi_year_match.groups())
        if start_year <= end_year:  # Ensure valid range
            return pd.period_range(start=f"{start_year}", end=f"{
                                                  end_year}", freq="Y")
        return None  # Invalid range

    elif multi_quarter_match:
        start_q, end_q = multi_quarter_match.groups()
        return pd.period_range(start=start_q, end=end_q, freq="Q")

    return None  # No match found
```

This code adds a check when creating a `Period` from a date string, checking if it is a multiyear-span format. If it is, it is handled by the function that parses these dates, `parse_multiyear`.

2.5 UML Class Diagram



(a) Before changes



(b) After changes

Figure 1: Behavioural UML Diagram showcasing the creation of a `Period` object before and after implementing our changes.

Figure 1 shows an overview of our changes, located mainly in the `Period` class. This change aims to do three things:

1. Add support for ISO8601 ordinal dates.
2. Fix the bug in handling week spans of format `yyyy-mm-dd / yyyy-mm-dd` and add support from `yyyymmdd-yyyymmdd`.

3. Add support for multi-year spans such as `yyyy-yyyy` and `yyyyQ-yyyyQ`.

Ordinal dates

Adding support for ordinal dates was done in a manner that was as non-invasive as possible. To support this format, while also allowing all other parameters to work correctly, we use a form of **pre-processing** which translates the `yyyy-ddd` format to `yyyy-mm-dd`, without changing anything else about the date string.

Week spans

Before starting work on this change, it was discovered that a solution was already implemented. However, as can be seen from Figure 1, the solution was to check if the week-span format `yyyy-mm-dd / yyyy-mm-dd` was used if the parser failed, and use a helper method `_parse_weekly_string` to parse it in that case. This design pattern of checking-after-fail was deemed a code smell and also lead to a bug. Any second date in the week span where the year could be seen as a valid time was parsed incorrectly by `parse_datetime_string_with_reso` and produced an incorrect date. However, this did not raise a `ValueError` and was therefore not caught and corrected. An example of a passing and failing week span can be seen in the example below:

```
>>> pd.Period("2023-01-01/2023-01-07")
Period('2023-01-01 20:23', 'min')
>>> pd.Period("2061-01-01/2061-01-07")
Period('2061-01-01/2061-01-07', 'W-FRI')
```

This was resolved by checking the date before allowing the default parser to take care of it. If it was a week span, this was handled directly instead of handling it after a failure.

Multi-year spans

Following the already established convention of having separate parsers for date strings not supported by `parse_datetime_string_with_reso`, we created a new static method `parse_multiyear`, to handle this. Following the adopted convention of choosing a parser after checking the date format, a check was created that caught all supported multi-year spans and parsed them separately.

2.5.1 Design considerations

One of the main design patterns used in this refactoring was establishing a **chain of responsibility**. Instead of the previous method of recovering from failure we introduced **guard clauses** that either process the date string or pass them along to the next handler. This also makes use of the **fail-fast principle**, meaning we do not try to recover or continue from a possibly flawed state as previously but have immediate failures. We combine this with the **single responsibility**

principle, by allowing each type of date string to have its own parser, which are separate functions with smaller scopes.

For the ordinal dates, we used a **pre-processing** pattern to minimally affect the rest of the parsing. We did not use a separate parser for this to avoid code duplication.

3 Testing

3.1 Regression test results

Links to regression test logs:

- **Before issue resolution:**
  `15 failed, 224074 passed, 6586 skipped, 4088 deselected, 1854 xfailed, 95 xpassed, 31 warnings, 41 errors`
- **After issue resolution:**
  `14 failed, 224161 passed, 6586 skipped, 4088 deselected, 1854 xfailed, 95 xpassed, 31 warnings, 41 errors`

Fewer tests fails after implementation of new functionality, suggesting that no regression has taken place in our implementation of new functionality.

3.2 Functionality test results

Links to functionality test logs:

- **Before issue resolution:** `55 failed, 70 passed`
- **After issue resolution:** `0 failed, 125 passed`

Functionality tests were run on the test suite in , containing 39 test cases prior to extension. The 31 new test cases passing before implementation of additional functionality account for tests checking for parsing errors, and were expected to pass even without the functionality. The remaining tests pass as expected when tested against the implemented functionality.

4 Discussion

4.1 Future Work Plan

Future work should implement requirements **1**, **2**, and **4**:

1. **Business Years Not Starting in January**
   - Support fiscal years with **custom start months** (e.g., "2023FY-Apr").

- Modify `Period` parsing to recognize and handle non-January year starts.

2. **Quarters Starting in Custom Months**
   - Extend quarter parsing to support non-standard starts (e.g., "2024Q1-Feb").
   - Ensure consistency with fiscal calendars and existing pandas behavior.

4. **Business Days Support**
   - Ensure consistency with fiscal calendars and existing pandas behavior.
   - Align with pandasâĂŹ existing BusinessDay (BD) frequency.

Each requirement should be accompanied by unit tests of sufficient coverage.

4.2 Overall Experience

Project Structure

During the course of this assignment, the main problems we've faced are a direct consequence of the scale of the project we chose, and it became clear rather quickly that it would have been prudent to work with a smaller codebase. The fact that a lot of Python tools rely on Pandas did not make things easier either, as mentioned in the onboarding description.

Pandas is also well documented, but is lacking in documentation of very specific behaviour and design, as seen in the existing parsing functionality of the `Period` class. This did not facilitate our own development. Ironically, we also failed to be specific in our own design intents, with requirements defined somewhat loosely before work began on implementation and testing. Design choices made in implementation conflicted with assumptions made in the design of the tests, both of which were worked on in parallel, which ultimately resulted in some amount of work needing to be redone.

Collaboration/Way of Working

During course, communication has improved, and most issues arise in having to explore the full scope of the tasks as we go, leading to differing interpretations and assumptions made as we worked in parallel. In general, communication and collaboration is open and efficient, and as the course has progressed, we've made concerted efforts in actively preventing miscommunication with check-in meetings.

However, wasted work and backtracking still occurs fairly regularly. Judging by the Essence standard, we currently occupy the Collaborating state.