Friedrich-Alexander-Universität
Technische Fakultät

**Department of Computer Science**
Computer Science 6
(Data Management)

## Knowledge Discovery in Databases with Exercises
## Winter Semester 2025/2026

# Submission 1:
# Frequent Patterns

## About this Assignment

Throughout the course of this assignment, you will independently implement the two methods, Apriori (Task 1) and FP-growth (Task 2). For this purpose, a basic code skeleton, several helper classes, and some test cases are provided to you.

### Key Data

- **Max. Group Size:** 3
- **Max. Points:** 40
- **Estimated Workload:** 4 - 6 hours

### How to Work on the Assignment

To start working on the assignment, you'll need to accept the assignment via GitHub Classroom by clicking the provided link. This will set up a new GitHub repository for your group, packed with all the necessary files for the assignment. If you're joining an existing group, it'll add you to that group's repository.[1]

Once that's done, you have two main options for working on your assignment. You can clone the repository[2] to your local machine by navigating to `Code` → `Local`, which allows you to work directly from your computer. Alternatively, you might prefer using GitHub Codespaces by selecting `Code` → `Codespaces` for a virtual online environment, complete with the ability to run Python through the `Terminal` provided.

Whichever method you choose, it's crucial to commit and push your changes back to the repository to submit your solution[2]. After your submission, GitHub Actions takes over to automatically grade your solution and provide feedback. You'll find this feedback in the `Actions` tab of your repository. If you didn't receive full points, you can improve your solution and push the changes back to the repository to trigger a reevaluation.

[1] Each student must join individually. You can join groups while accepting an assignment.
[2] If you're unfamiliar with Git or GitHub, check out this helpful guide: `https://github.com/git-guides/`

## How to Prepare the Transfer the Points to StudOn

In addition to joining the GitHub Classroom, you also need to register your GitHub username on StudOn. This is necessary to transfer the points you've earned on GitHub to StudOn. To do this, enter your GitHub username in `Submission 1 - GitHub Username`. Make sure to enter your username correctly, as otherwise, the points cannot be transferred.

After submission deadline, the points you've earned on GitHub will be transferred to StudOn. This process is not immediate and may take a few days. If you have any questions or issues, please contact us via the StudOn forum.

## Restrictions

Within the scope of your implementation, you are not permitted to modify the helper classes, the test cases, or the provided GitHub Actions.

This will be checked on a random basis, and any attempt to do so will result in zero points for the involved group, similar to the consequences of plagiarism.

# Task 1: Apriori

Apriori is a classic algorithm for frequent itemset mining over transactional databases. It proceeds by identifying the frequent individual items in the database and extending them to larger and larger itemsets as long as those itemsets appear sufficiently often in the database.

## Task 1.1 (2 Points)

At the beginning of Apriori, the identification of 1-itemsets is paramount.

Open `apriori.py` in your repository and implement the `_generate_one_itemsets`, which generates all 1-itemsets for a given dataset:

**_generate_one_itemsets**

```
def _generate_one_itemsets(
    self,
    dataset: Dataset
) -> Set[Itemset]:
```

**Description:**
- Generate all 1-itemsets for the given dataset.

**Parameters:**
- `dataset` (Dataset): The dataset for which the 1-itemsets should be generated.

**Returns:**
- `Set[Itemset]`: A set containing all 1-itemsets that are contained in the dataset.

Make sure that you expect a `Dataset` and return a `Set[Itemset]`[3].

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/apriori/test_generate_one_itemsets.py
```

[3] **Hint:** `Itemset` and `Database` are helper classes that can be found in the `classes/` folder.

## Task 1.2 (2 Points)

After the 1-itemsets have been identified, the next step is to count the occurrences of these itemsets in the dataset.

Complete the function `_count_occurrences_of_itemsets`, which counts the occurrences of all given itemsets in the dataset:

**_count_occurrences_of_itemsets**

```
def _count_occurrences_of_itemsets(
    self,
    dataset: Dataset,
    itemsets: Set[Itemset]
) -> ItemsetsWithOccurrenceCounts:
```

**Description:**

- Count the occurrences of the given itemsets in the dataset.

**Parameters:**

- `dataset` (Dataset): The dataset for which the itemset occurrences should be counted.
- `itemsets` (Set[Itemset]): The itemsets for which the occurrences should be counted. The itemsets do not need to be present in the dataset.

**Returns:**

- `ItemsetsWithOccurrenceCounts`: A dictionary containing the itemsets as keys and their occurrence counts as values.

Expect that the input consists of a `Dataset` and a `Set[Itemset]`. The method should return an instance of `ItemsetsWithOccurrenceCounts`.

Also be aware that the method should be able to count the occurrences of itemsets with any length, not just 1-itemsets.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/apriori/test_count_occurrences_of_itemsets.py
```

## Task 1.3 (2 Points)

After counting occurrences, it is necessary in Apriori to prune all itemsets falling below the minimum support threshold.

Complete the function `_prune_itemsets_below_min_support`, which prunes all itemsets that do not meet the minimum support threshold:

**_prune_itemsets_below_min_support**

```
def _prune_itemsets_below_min_support(
    self,
    itemsets_with_occurrence_counts: ItemsetsWithOccurrenceCounts
) -> Set[Itemset]:
```

**Description:**

- Prune itemsets that are below the minimum support threshold.

**Parameters:**

- `itemsets_with_occurrence_counts` (ItemsetsWithOccurrenceCounts): A dictionary containing the itemsets as keys and their occurrence counts as values.

**Returns:**

- `Set[Itemset]`: A set containing all itemsets that are considered frequent.

The input consists of an `ItemsetsWithOccurrenceCounts`. The (absolute) minimum support is a member variable of the Apriori object and can therefore be accessed via `self.min_support`. You have to return a `Set[Itemset]`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/apriori/test_prune_itemsets_below_min_support.py
```

## Task 1.4 (5 Points)

The last missing step in the Apriori algorithm is to generate the candidate itemsets for the next iteration.

Complete the function `_generate_candidate_itemsets`, which generates the candidate itemsets for the next iteration:

**_generate_candidate_itemsets**

```
def _generate_candidate_itemsets(
    self,
    frequent_itemsets: Set[Itemset]
) -> Set[Itemset]:
```

**Description:**

- Generate length-k+1 candidate itemsets based on the given frequent itemsets. k is the length of the longest frequent itemset.

**Parameters:**

- `frequent_itemsets` (Set[Itemset]): A set containing all frequent itemsets.

**Returns:**

- `Set[Itemset]`: A set containing all length-k+1 candidate itemsets.

The input consists of a `Set[Itemset]` containing all frequent itemsets. The method should return a `Set[Itemset]` containing all candidate itemsets for the next iteration.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/apriori/test_generate_candidate_itemsets.py
```

## Task 1.5 (5 Points)

All previous steps can be combined into a single algorithm: Apriori.

Complete the function `fit`, which implements the Apriori algorithm:

fit

```
def fit(
    self,
    dataset: Dataset
):
```

**Description:**
- Use the Apriori algorithm to find all frequent itemsets in the given dataset. Saves the frequent itemsets in the frequent_itemsets attribute.

**Parameters:**
- `dataset` (Dataset): The dataset to which the Apriori algorithm should be fitted.

**Returns:**
- None - The method saves the frequent itemsets in `self.frequent_itemsets`

The input consists of a `Dataset`. The method should not return anything but save the frequent itemsets in the `self.frequent_itemsets` attribute of the Apriori object.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/apriori/test_fit.py
```

## Task 2: FP-growth

While Apriori represents a very simple approach to mining frequent itemsets, there are alternative methods available. An interesting method is FP-growth, which necessitates only two passes on the original dataset. This is achieved through the utilization of the so-called FP-trees.

### Task 2.1 (3 Points)

The first step in FP-growth is to find all frequent 1-itemsets. At the same time, it is beneficial not to immediately discard the occurrence counts of the frequent 1-itemsets.

In `fpgrowth.py` implement `_generate_frequent_one_itemsets_with_occurrence_counts`, which generates all 1-itemsets together with their occurrence counts for a given dataset:

**_generate_frequent_one_itemsets_with_occurrence_counts**

```
def _generate_frequent_one_itemsets_with_occurrence_counts(
    self,
    dataset: Dataset
) -> ItemsetsWithOccurrenceCounts:
```

**Description:**
- Generate all frequent 1-itemsets for the given dataset.

**Parameters:**
- `dataset` (Dataset): The dataset for which the frequent 1-itemsets should be generated.

**Returns:**
- `ItemsetsWithOccurrenceCounts`: A dictionary containing the frequent 1-itemsets as keys and their occurrence counts as values.

Expect a `Dataset` as input and return an `ItemsetsWithOccurrenceCounts`. Remember that you did do a similar task in Apriori.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/fpgrowth/test_generate_frequent_one_itemsets_with_occurrence_counts.py
```

## Task 2.2 (2 Points)

After identifying the frequent 1-itemsets, the f-list can be generated. This is where the occurrence counts of the frequent 1-itemsets come into play.

Complete `_generate_f_list`:

**_generate_f_list**

```
def _generate_f_list(
    self,
    frequent_one_itemsets: ItemsetsWithOccurrenceCounts
) -> List[Itemset]:
```

**Description:**

- Generate the f-list for the given frequent 1-itemsets.

**Parameters:**

- `frequent_one_itemsets` (ItemsetsWithOccurrenceCounts): The frequent 1-itemsets with their occurrence counts for which the F-list should be generated.

**Returns:**

- `List[Itemset]`: A f-list containing the frequent 1-itemsets sorted by decreasing occurrence count.

The input consists of an `ItemsetsWithOccurrenceCounts`. The return value should be a `List[Itemset]` containing the frequent 1-itemsets sorted by decreasing occurrence count

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/fpgrowth/test_generate_f_list.py
```

## Task 2.3 (4 Points)

After generating the f-list, the dataset can be sorted according to the f-list. This is necessary to build the FP-tree.

Complete the function `_sort_dataset_accoring_to_f_list`, which sorts the dataset according to the f-list:

**_sort_dataset_according_to_f_list**

```
def _sort_dataset_according_to_f_list(
    self,
    dataset: Dataset,
    f_list: List[Itemset]
) -> SortedDataset:
```

**Description:**
- Sort the dataset according to the given f-list.

**Parameters:**
- `dataset` (Dataset): The dataset to be sorted.
- `f_list` (List[Itemset]): The f-list according to which the dataset should be sorted.

**Returns:**
- `SortedDataset`: The sorted dataset.

The input consists of a `Dataset` and a `List[Itemset]`. The method should return a `SortedDataset`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/fpgrowth/test_sort_dataset_according_to_f_list.py
```

## Task 2.4 (3 Points)

With the sorted dataset, the FP-tree can be built.

Complete the function `_construct_initial_fp_tree`, which builds the initial FP-tree:

**_construct_initial_fp_tree**

```
def _construct_initial_fp_tree(
    self,
    sorted_dataset: SortedDataset
) -> FPTree:
```

**Description:**

- Construct the initial FP-tree from the given sorted dataset.

**Parameters:**

- `sorted_dataset` (SortedDataset): The sorted dataset from which the initial FP-tree should be constructed.

**Returns:**

- `FPTree`: The initial FP-tree.

**Hint:** `FPTree` implements a method `add_items_to_tree`, which might be helpful for this task.

The input consists of a `SortedDataset`. The method should return an `FPTree`.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/fpgrowth/test_construct_initial_fp_tree.py
```

## Task 2.5 (5 Points)

In FP-growth, besides the initial FP-tree, the so-called conditional FP-trees also play a crucial role. To be able to build these, the conditional pattern base must be generated.

Complete the function `_get_conditional_pattern_base`:

**_get_conditional_pattern_base**

```
def _get_conditional_pattern_base(
    self,
    item: Item,
    fp_tree: FPTree
) -> ConditionalPatternBase:
```

**Description:**

- Get the conditional pattern base for the given item in the FP-tree.

**Parameters:**

- `item` (Item): The item for which the conditional pattern base should be generated.
- `fp_tree` (FPTree): The FP-tree from which the conditional pattern base should be extracted.

**Returns:**

- `ConditionalPatternBase`: The conditional pattern base for the given item.

The input consists of an `Item` and an `FPTree`. The output is a `ConditionalPatternBase`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/fpgrowth/test_get_conditional_pattern_base.py
```

## Task 2.6 (3 Points)

With the conditional pattern base, the conditional FP-tree can be built.

Complete the function `_construct_conditional_fp_tree`:

**_construct_conditional_fp_tree**

```
def _construct_conditional_fp_tree(
    self,
    conditional_pattern_base: ConditionalPatternBase
) -> FPTree:
```

**Description:**

- Construct a conditional FP-tree from the given sorted dataset.

**Parameters:**

- `conditional_pattern_base` (ConditionalPatternBase): The conditional pattern base for which the conditional FP-tree should be constructed.

**Returns:**

- `FPTree`: The conditional FP-tree.

The input consists of a `ConditionalPatternBase`. The method should return an `FPTree`.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/fpgrowth/test_construct_conditional_fp_tree.py
```

## Task 2.7 (4 Points)

The last missing step in FP-growth is to recursively mine the frequent itemsets.

Complete `fit`, which implements the FP-growth algorithm:

**fit**

```
def fit(
    self,
    dataset: Dataset
):
```

**Description:**
- Use the FP-growth algorithm to find all frequent itemsets in the given dataset. Saves the frequent itemsets in the frequent_itemsets attribute.

**Parameters:**
- `dataset` (Dataset): The dataset to which the FP-growth algorithm should be fitted.

**Returns:**
- None - The method saves the frequent itemsets in `self.frequent_itemsets`

The input consists of a `Dataset`. The method should not return anything but save the frequent itemsets in the `self.frequent_itemsets` attribute of the FP-growth object.

You can test whether your implementation is correct by executing the following command in the console:

```
pytest tests/fpgrowth/test_fit.py
```

# Appendices

In Task 1 and Task 2 test cases are provided and used to grade the submission.

## Dataset(s)

The most test cases are based on the following data sets:

### Small Fruit Dataset

All test cases starting with the prefix `test_with_small_fruit_dataset` are based on a small transactional dataset regarding fruits.

The dataset is structured as follows:

| TID | Items |
|---|---|
| 1 | Apple, Banana, Cherry |
| 2 | Banana, Cherry |
| 3 | Cherry, Apple |
| 4 | Dragonfruit, Apple, Cherry |
| 5 | Apple, Dragonfruit |

Table 1: Small Fruit Dataset

### Large Book Dataset

All test cases starting with the prefix `test_with_large_book_dataset` are based on a large(r)[4] transactional dataset.

The dataset is structured as follows:

| TID | Books |
|---|---|
| 1 | Book 1, Book 2, Book 3 |
| 2 | Book 2, Book 4, Book 5 |
| 3 | Book 3, Book 6, Book 7 |
| 4 | Book 4, Book 8, Book 9 |
| 5 | Book 1, Book 5, Book 10 |
| 6 | Book 6, Book 7, Book 8 |
| 7 | Book 9, Book 10, Book 2 |
| 8 | Book 3, Book 4, Book 5 |
| 9 | Book 6, Book 8, Book 1 |
| 10 | Book 7, Book 9, Book 10 |

| Book | Title |
|---|---|
| Book 1 | The Shadows of Tomorrow |
| Book 2 | Echoes of a Forgotten Realm |
| Book 3 | Whispers of the Ancient World |
| Book 4 | Chronicles of the Unseen |
| Book 5 | Legends of the Fallen Skies |
| Book 6 | Tales of the Crimson Dawn |
| Book 7 | Secrets of the Silent Ocean |
| Book 8 | Memories of the Last Horizon |
| Book 9 | Dreams of the Distant Stars |
| Book 10 | Visions of the Lost Empire |

Table 2: Large Book Dataset

[4] The term "large" is, of course, somewhat exaggerated. However, the datasets should still be comprehensible by humans, which is why this is the largest dataset we use for testing.

## Helper Classes

The following helper classes are provided in the `classes/` folder to support your implementation. Each class serves a specific purpose in the frequent pattern mining algorithms.

### Basic Data Structures

#### Item (`classes/item.py`)

Represents a single item in the dataset.

- **Attributes:** `name` (str) - The name of the item
- **Usage:** Create items like `Item("Apple")` or `Item("Book 1")`
- **Example:** `apple = Item("Apple")`

#### Itemset (`classes/itemset.py`)

Represents a set of items (an itemset). This is the fundamental data structure for representing collections of items.

- **Attributes:** `items` (FrozenSet[Item]) - An immutable set of items
- **Usage:** Create itemsets like `Itemset({Item("Apple"), Item("Banana")})`
- **Iteration:** You can iterate over items: `for item in itemset: ...`
- **Example:**

```
apple = Item("Apple")
banana = Item("Banana")
itemset = Itemset({apple, banana})
for item in itemset:
    print(item.name)
```

#### ItemTuple (`classes/item_tuple.py`)

Represents an ordered tuple of items. Unlike Itemset, the order of items matters here.

- **Attributes:** `items` (Tuple[Item]) - An ordered tuple of items
- **Usage:** Important for FP-Growth where item order according to the f-list is crucial
- **Example:** `ItemTuple((Item("Apple"), Item("Banana")))`

### Transaction and Dataset Classes

#### Transaction (`classes/transaction.py`)

Represents a single transaction in the database.

- **Attributes:**
  - `id` (int) - Unique transaction identifier
  - `items` (Itemset) - The items purchased in this transaction
- **Usage:** Access transaction data: `transaction.id`, `transaction.items`
- **Example:**

```
transaction = Transaction(1, Itemset({Item("Apple"), Item("Banana")}))
print(f"Transaction␣{transaction.id}␣contains␣{len(transaction.items.items)}␣items"
                                                )
```

#### Dataset (`classes/dataset.py`)

Represents a complete transactional dataset.

- **Attributes:** `transactions` (FrozenSet[Transaction]) - All transactions
- **Usage:** Iterate over transactions: `for transaction in dataset.transactions: ...`
- **Example:**

```
for transaction in dataset.transactions:
    print(f"Transaction␣{transaction.id}␣has␣{len(transaction.items.items)}␣items")
```

#### SortedTransaction (`classes/sorted_transaction.py`)

Represents a transaction where items are ordered (used in FP-Growth).

- **Attributes:**
  - `id` (int) - Transaction identifier
  - `items` (ItemTuple) - Items sorted according to f-list
- **Usage:** Similar to Transaction, but items maintain order

#### SortedDataset (`classes/sorted_dataset.py`)

Represents a dataset with sorted transactions.

- **Attributes:** `transactions` (FrozenSet[SortedTransaction])
- **Usage:** Used in FP-Growth after sorting transactions according to f-list

### Counting and Support Classes

#### ItemsetsWithOccurrenceCounts (`classes/itemsets_with_occurrence_counts.py`)

A dictionary-like container that stores itemsets with their occurrence counts.

- **Inherits from:** Python's `UserDict`
- **Key Methods:**
  - `add_occurrence(itemset)` - Increments count by 1
  - `set_occurrence_count(itemset, count)` - Sets specific count
  - `get_occurrence_count(itemset)` - Returns current count
  - `remove_occurrence(itemset)` - Decrements count by 1
- **Usage:** Essential for counting itemset frequencies in both algorithms
- **Example:**

```
counts = ItemsetsWithOccurrenceCounts({itemset1, itemset2})
counts.add_occurrence(itemset1)  # Count becomes 1
counts.set_occurrence_count(itemset2, 5)  # Set count to 5
current_count = counts.get_occurrence_count(itemset1)  # Returns 1
```

### FP-Tree Data Structures

#### FPTree (`classes/fp_tree.py`)

The main class representing an FP-Tree structure.

- **Attributes:** `root` (FPTreeRootNode) - The root node of the tree
- **Key Methods:**
  - `add_items_to_tree(item_tuple, occurrence_count)` - Adds items to tree
  - `get_header_table()` - Returns the header table
  - `get_all_item_nodes()` - Returns all item nodes
  - `is_single_path()` - Checks if tree has only one path
  - `is_empty()` - Checks if tree is empty
- **Usage:** Central data structure for FP-Growth algorithm
- **Example:**

```
fp_tree = FPTree()
item_tuple = ItemTuple((Item("Apple"), Item("Banana")))
fp_tree.add_items_to_tree(item_tuple, 3)  # Add with count 3
```

**FPTreeNode** (`classes/fp_tree_node.py`)

Base class for all FP-Tree nodes.

- **Attributes:**
  - `childs` (List) - List of child nodes
  - `occurrence_count` (int) - Frequency counter
- **Methods:** `get_predecessors()` - Returns predecessor nodes
- **Usage:** Base functionality for tree nodes (you typically don't create these directly)

**FPTreeRootNode** (`classes/fp_tree_root_node.py`)

The root node of an FP-Tree.

- **Inherits from:** FPTreeNode
- **Special Properties:**
  - Has no parent (`parent = None`)
  - Not included in header table
- **Key Methods:**
  - `add_to_header_table(header_table)` - Adds children to header table
  - `get_all_item_nodes()` - Returns all item nodes in subtree
  - `is_single_path()` - Checks if only one path exists
  - `is_empty()` - Checks if tree is empty
- **Usage:** Automatically created when you create an FPTree

**FPTreeItemNode** (`classes/fp_tree_item_node.py`)

Represents an item node in the FP-Tree.

- **Inherits from:** FPTreeNode
- **Attributes:**
  - `item` (Item) - The item this node represents
  - `occurrence_count` (int) - How often this path occurs
  - `parent` (FPTreeNode) - Parent node
- **Key Methods:**
  - `get_predecessors()` - Returns path from root to this node
  - `add_to_header_table(header_table)` - Adds node to header table
  - `get_all_item_nodes()` - Returns all item nodes in subtree
  - `is_single_path()` - Checks if subtree has only one path
- **Usage:** Automatically created when adding items to FP-Tree

### FPTreeHeaderTable (classes/fp_tree_header_table.py)

Represents the header table of an FP-Tree.

- **Attributes:** `elements` (List[FPTreeHeaderTableElement])
- **Usage:** Provides quick access to all nodes of a specific item
- **Example:**

```
header_table = fp_tree.get_header_table()
for element in header_table.elements:
    print(f"Item: {element.item.name}, Count: {element.overall_occurrence_count}")
```

### FPTreeHeaderTableElement (classes/fp_tree_header_table_element.py)

Represents one entry in the header table.

- **Attributes:**
  - `item` (Item) - The item this element represents
  - `overall_occurrence_count` (int) - Total occurrences across all nodes
  - `node_links` (List[FPTreeNode]) - Links to all nodes with this item
- **Usage:** Access all occurrences of a specific item in the tree
- **Example:**

```
for element in header_table.elements:
    print(f"Item {element.item.name} appears {len(element.node_links)} times in
                                                tree")
    for node in element.node_links:
        print(f"  Node with count: {node.occurrence_count}")
```

### Conditional Pattern Classes (FP-Growth specific)

### ConditionalPattern (classes/conditional_pattern.py)

Represents a single conditional pattern.

- **Attributes:**
  - `prefix_items` (ItemTuple) - The prefix path items (ordered according to f-list)
  - `occurrence_count` (int) - How often this pattern occurs
- **Usage:** Used internally in FP-Growth for conditional pattern bases
- **Note:** The prefix items must be ordered from top to bottom according to the f-list

### ConditionalPatternBase (`classes/conditional_pattern_base.py`)

Represents a conditional pattern base for an item.

- **Attributes:** `conditional_patterns` (FrozenSet[ConditionalPattern])
- **Usage:** Contains all conditional patterns for a specific item (one item can have multiple prefix paths in the FP-tree)
- **Example:**

```
for pattern in conditional_pattern_base.conditional_patterns:
    print(f"Pattern: {[item.name for item in pattern.prefix_items.items]}")
    print(f"Count: {pattern.occurrence_count}")
```

## Practical Tips

- **Creating Items and Itemsets:**
  Always use the provided classes instead of raw strings or sets
- **Iteration:**
  Most classes support Python's standard iteration patterns (`for item in itemset`)
- **Immutability:**
  Many classes use frozen sets/tuples for thread safety and hashability
- **Type Hints:**
  Pay attention to the expected return types in function signatures
- **Testing:**
  Use the provided test cases to understand expected behavior
- **Dictionary Access:**
  `ItemsetsWithOccurrenceCounts` works like a dictionary - use square brackets for access
- **Tree Traversal:**
  Use the provided methods like `get_all_item_nodes()` instead of manual traversal
- **Debugging:**
  Most classes have `__str__` methods for easy printing and debugging