# RF_Technical_Documentation

# Source Organization

1. **data**→Directory containing raw data files for storing graph and subscription datasets in various formats (ex. JSON)
2. **data_loader.py**→ Contains logic to read data from the `data` directory and parse them into usable data structures for further analysis
3. **lures.py**→ Provides functions for identifying phishing lures in newly registered domain names and notifying team members subscribed to the associated target terms. Encapsulates this logic in the `LureNotifier` utility class
4. **test_lure_notifier.py**→ Unit tests for the `LureNotifier` utility class ``

# High-Level Approach

## Development Strategy

- Stepped through test cases and made drawings by hand to understand the problem
    - Helped me understand that team-supervisor relationships can be represented as directed edges
    - The "head" supervisor is the *root* of the tree, and all other team members are *descendants*
    - Supervisor hierarchy can be represented by a **tree** (if each team member has one *parent* / supervisor)or a **DAG** (if one team member has multiple supervisors)
- Used a **Test-Driven Development** approach
    - Left stubs for `LureNotifier` methods before writing unit tests
    - Helped me assess *edge cases* before delving into the implementation
- Learned the `unittest` library in Python (based on JUnit5)
- Leveraged Python's type hinting and `typing` module

## Key Design Components

- Decoupling of data loading vs. data processing
- **Data loading**
    - Leveraged the `os` module from python's standard library
        - Allows for portability of code regardless of OS (Windows, Unix, etc.)
        - Separated logic from the *extraction* of data to its *processing*
        - We can easily modify the code to support future data loading functionality, such as modifying retrieval to use HTTP or file I/O, or adding support for

new formats (JSON, CSVs, etc.)

  - **Data Processing**
  - Chose to represent sets of domains / target terms as sets rather than lists as they are in starter code
  - Allows for easier testing (since sets are unordered)
  - `Frozensets` are used for hashability and immutability
  - Refactored starter code for the `LureNotifier` class to use static methods decorators, since the class has no attributes and serves more as a utility class for now
    - **Decomposed** data processing into *multiple steps*
      - Notification process uses a BFS helper
      - Benefits to BFS - can easily modified to output nodes in the level-order traversal (top down)
        - Ex. if we want high priority team members (team members close to the root supervisor) to be notified first , we can look at the BFS output as an ordered list
- **Looking forward**
  - Implementing more robust data loading error handling
  - Considering what we want to mitigate when identifying potential phishing lures
    - *False negatives*
      - When we think a domain name is safe when it is actually a phishing site
      - Mitigate at all costs
    - *False positives*
      - We think a domain name might be a phishing lure when it is in fact a safe site
      - We generally should be okay with this
  - **Edge cases**: The starting set of target terms contains both `gmail` and `mail`, however `mail` is a subsequence of `gmail`, so any domain name containing the word `gmail` gets flagged as containing 2 target terms with a naive implementation
  - **Scalability**
    - If we need to handle millions of terms and domain names, we can use more advanced data structures like *PriorityQueues* to examine 'high importance' domain names first, and leverage **parallelism** to speed up data processing

## Challenges Faced

- Dealing with edge cases: distinguishing if this domain has 3 matching target terms or 2 target terms

```python
('gmail.mass.gov', ['gmail', 'mail', '.gov'])
```

  - Decided to count both - may increase false positive rates but will reduce false negative rates
- Understanding the team member hierarchy
  - Assumed a tree / DAG structure
    - Each team member has one or more supervisors (parent), and any team lead can have multiple team members (children)
    - **No cyclic dependencies**
- Representation of method inputs / outputs
  - Wanted the order of identified lure terms to not matter, however starter code used lists
    - If the order mattered, we would have had to check target terms in a specific order, adding **unnecessary complexity**
  - Opted to use `sets`, and then `frozensets`
  - Encountered `'Unhashable type'` exceptions
    - We cannot have `sets` of `sets`, since sets are inherently modifiable and thus cannot be deterministically hashed
    - `Frozensets` allowed me to represent matched target terms in an unordered way while maintaining immutability

## Testing Overview

- Used a TDD approach
  - Wrote tests to think through edge cases
  - All tests initially failed for unimplemented stubs
  - Implemented the required stubs
  - Iteratively **tested** and **refactored**

```shell
python -m unittest .\test_lure_notifier.py
```

- Leveraged unit testing to isolate different utility methods in the `LureNotifier` class
  - Test cases include...
    - Multiple domain matches
    - Notifications from both *root* and *leaf* nodes
    - Ignoring domains with insufficient lures identified

```
.....
---------------------------------------------------------------------
-
Ran 5 tests in 0.002s
OK
```

## Edge Cases Not Tested

- Special characters
    - I did some research and it seems that domain names are **case insensitve**, so we do not need to worry about domain names like `GooGle` or `PaYPaL` sneaking through our lure identifier
- "Near" strings
    - Some phishing lures that would make it under the radar in the current implementation include `cicso` (a misspelled version of `cisco`) and `appple.com` (may look like `apple.com` at a glance)
        - More advnaced techniques may be needed, like *regular expression matching*

## Technologies & Resources Used

- I spent about 30 minutes understanding the problem and designing by hand, 2-3 hours in the test / development / reading python docs / documenting the codebase cycle
- Git & GitHub for VCS (github.com/DunnyBunny1)
- Obsidian Markdown Editor for writing README
- PyCharm (JetBrains) IDE
- Python official docs
    - `typing`, `unittest` in particular
- Python unittest module tutorial