Table of Contents ǵeǹeřaţed ẁiţh DöcTöc

midterm-practice

Practice problems for midterm

Short answer (2-3 sentences)

Atomic write implementation

Describe a strategy for implementing atomic writes.

Semver major

What is a "major" revision in Semantic Versioning?

Variables outside version control

List three examples of variables that should be kept out of version-controlled code.

Salted graph problem

What is the problem that Salted Graphs is addressing?

Docker build arg vs environment variable

What is the difference between a Docker build argument vs environment variable?

Luigi ExternalTask vs ExternalProgramTask

Explain the difference between Luigi's ExternalTask vs ExternalProgramTask. What is a use case for each?

.env file

Suppose we have the following local project structure. We have an `.env` file that is used to store private configuration variables.

```
project
|-- .env
|-- docker-compose.yml
|-- Dockerfile
|-- myutils/
    |-- __init__.py
    |-- utils.py
|-- data/
```

```
    |-- dataset.csv
    |-- output.csv
```

How can we ensure that the `.env` file is not committed to version control?

Pipenv files

What is the difference between pipenv's Pipfile and Pipfile.lock files?

Stemming in tokenizer

In natural language processing, "stemming" a word means to find its root or base form. The `PorterStemmer` from the nltk library is one implementation of a stemmer. Example usage of `PorterStemmer`:

```
from nltk.stem import PorterStemmer

words = ["game","gaming","Gamed","games"]
stemmer = PorterStemmer()

print([stemmer.stem(word) for word in words])
# output: ['game','game','game','game']
```

Suppose we changed the the `tokenize` method in our pset utils to implement stemming:

Before:

```
def tokenize(text):
    ...
    return words


# tokenize('I tested my code')
# output: ['I', 'tested', 'my', 'code']
```

After:

```
from nltk.stem import PorterStemmer

def tokenize(text, stem=True):
    ...
    if stem:
        stemmer = PorterStemmer()
        words = [stemmer.stem(word) for word in words]
    return words

# tokenize('I tested the code', stem=True)
# output: ['i', 'test', 'the', 'code']
```

Suppose the old semantic version was 1.0.0. What should be the new semantic version? Justify your answer.

Pseudocode

Rewrite as functional

Rewrite the `process_text` function using `map`, `reduce`, `filter`, or other python methods for functional programming:

```
from nltk.stem import PorterStemmer

common_words = ['the', 'and', 'or', ... ]

def process_text(words):
    """
    :param words: list of words, e.g. ["I", "tested", "the", "code"]
    """
    output_words = []
```

```
    for word in words:
        stem = PorterStemmer.stem(word)
        if stem not in common_words:
            output_words.append(stem)
    return output_words
```

Logging using a decorator

Suppose we have the following function that prints a message including the function result.

```
def my_fun(x):
    result = x + 1
    print("The result was {}".format(result))
    return result
```

Implement a decorator that prints a similar message, but works for any function that returns an output. Show an example of how the decorator is used on an example function, and the expected console output.

Descriptors for validation

Suppose we have the following class. Its constructor accepts two arguments, `a` and `b`, that are non-zero positive numbers.

```
class Model:
    def __init__(self, a, b):
        """
        :param a: non-zero positive number
        :param b: non-zero positive number
        """
        self.a = a
        self.b = b

    ...
```

Implement a descriptor class called `Parameter` that can be used for argument validation, and rewrite the `Model` class definition to use the descriptor in a declarative way.