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.

Solution Write to a temporary file, then move the temporary file to the target location.

Semver major

What is a "major" revision in Semantic Versioning?

Solution A major revision introduces a change that breaks backcompatibility.

Variables outside version control

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

Solution

- Database credentials
- AWS credentials
- Any references to user-specific local file system paths

Salted graph problem

What is the problem that Salted Graphs is addressing?

Solution Efficiently identifying the appropriate tasks to rerun in complex data pipelines when data dependencies or task definitions change.

Docker build arg vs environment variable

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

NOTE: we said that explicit docker/git etc questions would be out of scope for the midterm

Solution Docker build arguments are typically used at image build-time, but are not available at container runtime. Environment variables are available in container runtime.

Luigi ExternalTask vs ExternalProgramTask

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

Solution An `ExternalTask` represents the output of a task that is not triggered by Luigi; usually used to represent starting inputs (such as data) to a Luigi pipeline, that are produced by some external process. The run of a `ExternalProgramTask` can be triggered by Luigi, however the program is not part of the same python process, but rather invoked by the command line, and the work may even be sent to a remote system (such as starting a job on a Spark cluster).

.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?

Solution Create a `.gitignore` file in the top-level project directory that includes the following line:

```
.env
```

Pipenv files

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

Solution `Pipfile` specifies minimal package requirements (which can be a range of valid versions). `Pipfile.lock` specifies a specific, frozen version of those requirements, including any subdependencies.

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.

Solution The API for the `tokenize` function has changed, due to the addition of a new keyword argument. However, the API is still backwards compatible since the argument is optional. This is a minor revision and the new semantic version is `1.1.0`.

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
```

Solution

Nöţe öṅ `nltk` šẏṅţaẋ› Föřǵöţ ţö iṅšţaṅţiaţe PöřţeřSţeḿḿeř iṅ ţ̈he ǵiṽeṅ cöde#šö ţecẗ̈ṅicaĺĺẏ `PorterStemmer` šẗ̈öùĺd be `PorterStemmer()`

```
def process_text(words):
    return list(map(PorterStemmer.stem, filter(lambda x: x not in common_words, words)))

# Alternate
def process_text(words):
    stemmer = PorterStemmer()
    return [stemmer.stem(w) for w in words if w not in common_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.

Solution

```
from functools import wraps
def logger(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print("The result was {}".format(result))
    return wrapper

# decorator usage

@logger
def process_text(words):
    ...

process_text(["I", "tested", "the", "code"])
# >> The result was ['I', 'test', 'code']
```

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

Solution

```
# NB: this one is tricky and actually requires a function we didn't explicitly cover in class.  It is a bit too hard for the
real midterm

class PositiveDefinite:
    def __set__(self, obj, val):
        # validate input
        if val <= 0:
            raise ValueError("Value must be >0")
        setattr(obj, '_' + self.name, val)

    def __get__(self, obj, type=None):
        return getattr(obj, '_' + self.name)

    def __set_name__(self, owner, name):
        # We did not explicitly cover this function, which
        # is necessary for generic lookups, and is only
        # available in 3.6+

        # Alternately, __init__() could take the name val
        # eg a = PositiveDefinite('a'), though it's clunky
        self.name = name

class Model:
    a = PositiveDefinite()
    b = PositiveDefinite()

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

```
        self.a = a
        self.b = b
```