**Final Exam Practice Set**

N Questions, 6 points each

**Atomic Workflows**

**Bulk inserts**

DB's and Parquet are both great for storing tabular data. If your task generates multiple rows for some input (eg a dataset of students per class, homes per region, tweets per day, etc), what is the biggest weakness of a DB vs Parquet with respect to an atomic, decentralized workflow?

1. Not every DB backend offers atomic writes; Django assumes the "Lowest Common Denominator" in terms of features and you have to be careful to ensure the data is inserted atomically, if at all.
2. Parquet is a column store, whereas a DB typically is not; therefore, inserting multiple rows of data can be much more performant in a Parquet column store if the data is large.
3. DB's only offer true atomic writes for single row inserts, or may behave poorly with atomic bulk inserts.
4. Your output might legitimately contain 0 rows; there is no way know that 0 rows have been successfully inserted into a DB without storing additional metadata

**Normalization**

Would normalization of an existing database typically result in a greater, fewer, or unchanged number of ORM models?

1. Greater
2. Fewer
3. Unchanged
4. Depends on the models - there is no general pattern

**Higher Levels**

**Why Meta**

You want to create a family of classes where instances are automatically cached. Which of the following use cases can be solved with a metaclass?

1. You can use the metaclass to automatically wrap constructor functions with lru\_cache instead of wrapping those functions on every single subclass.
2. Perhaps the cache is a system which must be instructed to create a file for every class cache when your app starts; you create a metaclass which registers every new subclass when it is defined, and your app can inspect the registry to ensure all cache files are created before your code runs.
3. If the cache should behave differently for various subclasses, such as using slightly different key schemes to look up instances in the cache, a metaclass is the best way to customize how functions work across sub classes.
4. None of the above are appropriate use cases for a metaclass.

**Decorators^2**

Examine the following code:

**def** d(decorator):

@wraps(decorator)

**def** wrapped(func=None, \*\*kwargs):

**if** func **is** None: **return** **lambda** newfunc: decorator(func=newfunc, \*\*kwargs)

**return** decorator(func=func, \*\*kwargs)

**return** wrapped

**Rename**the function *d* to describe its purpose. Then, use it to modify the decorator *logme* below, which logs all args and kwargs to the **root logger by default**, to allow a different logger object to be specified during decoration. The change should be backwards compatible for all functions using logme as written.

**import** **logging**

**def** logme(func):

@wraps(func)

**def** wrapped(\*args, \*\*kwargs):

logging.root.info(str(args) + ' ' + str(kwargs))

**return** func(\*args, \*\*kwargs)

**return** wrapped

Use logme to decorate the following three functions as indicated:

other\_logger = logging.getLogger('other\_logger')

*# Accept defaults of logme*

**def** f(x):

**return** x + 1

*# Explicitly specify root logger, to show the syntax*

**def** g(x):

**return** x + 2

*# Specify other\_logger*

**def** h(x):

**return** x + 3

**Descriptors**

The following snippet exhibits an attempt to isolate functionality, but suffers a higher level design flaw:

**class** HashedGithubId:

*"""Descriptor to manage Github Id secretly"""*

**def** \_\_get\_\_(self, instance, owner):

**return** instance.\_github\_hashed\_id

**def** \_\_set\_\_(self, instance, value):

**if** **not** isinstance(value, bytes):

value = hash\_str(bytes)

instance.\_github\_hashed\_id = value

**class** User:

hashed\_id = HashedGithubId()

What is the best way to improve the code?

1. The descriptor should inspect its assigned name or implement \_\_set\_name\_\_ rather than hardcoding \_github\_hashed\_id.
2. The descriptor is too specific to GitHub; it should be renamed and used for any hashed values.
3. This should not be a descriptor. Since this is memoizing, lru\_cache or similar would be a better choice.
4. The descriptor assumes too much about the hash implementation, which should be parameterized, eg HashedGithubId(hash=sha256)

**Visualization**

**Grammars**

A grammar of graphics:

1. Suggests when to use a particular plot, such as boxplot or scatterplot
2. Describes the optimal data-ink ratio of a given plot
3. Allows us to gain insight into the deep structure that underlies statistical data
4. Describes elements of a plot independently from their presentation

**Plotting**

The following code snippet is likely to generate a graph with which problem?

**import** **numpy** **as** **np**

**from** **some\_plotting\_library** **import** scatter\_plot

*# Create data*

N = 50000

x = np.random.rand(N)

y = np.random.rand(N) + 3

colors = (0, 0, 0)

area = np.pi \* 3

*# Plot*

scatter\_plot(x, y, size=area, colors=colors, alpha=0.5)

1. A non-uniform, perceptually distracting colormap
2. Data privacy issues, if the data is sent to an external server
3. Over-plotting
4. Low data/ink ratio

**Python vs SQL**

**SQL Frameworks**

The best reason to choose SQLAlchemy over Django is:

1. It exposes more DB-specific features, without assuming the lowest common denominator of DB features
2. The backend, in general, produces higher quality and more performant SQL
3. SqlAlchemy operates better with Flask, which is a better web framework than the relevant web serving components of Django. Together, the two frameworks are considered the gold standard for serious python web applications.
4. Better ability to write arbitrary SQL code like joining tables that don't have explicit foreign keys

**Internals**

**Grains**

For 2-d arrays used by pandas and other DataFrame-like objects, is it better for them to be C-contiguous or F-contiguous? Why?

**Reversing**

Here, numpy can reverse an array about 200 times faster than pure python:

In [1]: a = list(range(10000))

In [2]: b = np.arange(10000)

In [3]: %timeit a[::-1]

30.9 µs ± 148 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

In [4]: %timeit b[::-1]

177 ns ± 3.93 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

Explain the performance boost, and any other benefits or drawbacks of using numpy in this manner.

**Frameworks**

**Optimization and Speed**

**LRU Caching**

**A bad design**

Which of the following is a poor application of lru\_cache, assuming a small cache size compared to the number of keys you anticipate caching?

1. Caching the results of instance methods
2. Caching the results of a function of the current time of day
3. Caching whether a given luigi task has completed
4. Caching a recursive function like fib(n)

**A fatal flaw**

An easy way to run an external program from python is to invoke it as a subprocess, and capture the results back. This is easy to do using subprocess and check\_output (which adds some nice error handling). The following is copied from the [python docs](https://docs.python.org/3/library/subprocess.html#subprocess.check_output); note how a shell command like echo Hello World! is handled as a sequence:

**from** **subprocess** **import** check\_output

>>> check\_output(["echo", "Hello World!"])

'Hello World!**\n**'

Let's say you want to wrap a program to avoid boilerplate, and additionally want to cache the results, like so:

@lru\_cache()

**def** some\_program(cmd):

**return** check\_output(['some\_program', '--default-option'] + cmd)

>>> some\_program(['--option2', 'file.txt'])

Why will this fail?

**Design patterns**

**Custom property**

Suppose we have the following method:

**def** custom\_property(func):

@property

@wraps(func)

**def** wrapped(self):

attr = '\_' + func.\_\_name\_\_

**try**:

**return** getattr(self, attr)

**except** **AttributeError**:

v = func(self)

setattr(self, attr, v)

**return** v

**return** wrapped

Which of the following design patterns best describes the provided custom\_property implementation?

1. descriptor
2. memoization
3. monkey patching
4. factory