provenance Documentation

Release 0.10.0+21.g3539f2b.dirty

Ben Mabey

Oct 08, 2018

Guides

`provenance` is a Python library for function-level caching and provenance that aids in creating Parsimonious Pythonic Pipelines™. By wrapping functions in the `provenance` decorator computed results are cached across various tiered stores (disk, S3, SFTP) and provenance (i.e. lineage) information is tracked and stored in an artifact repository. A central artifact repository can be used to enable production pipelines, team collaboration, and reproducible results. The library is general purpose but was built with machine learning pipelines in mind. By leveraging the fantastic joblib library object serialization is optimized for `numpy` and other PyData libraries.

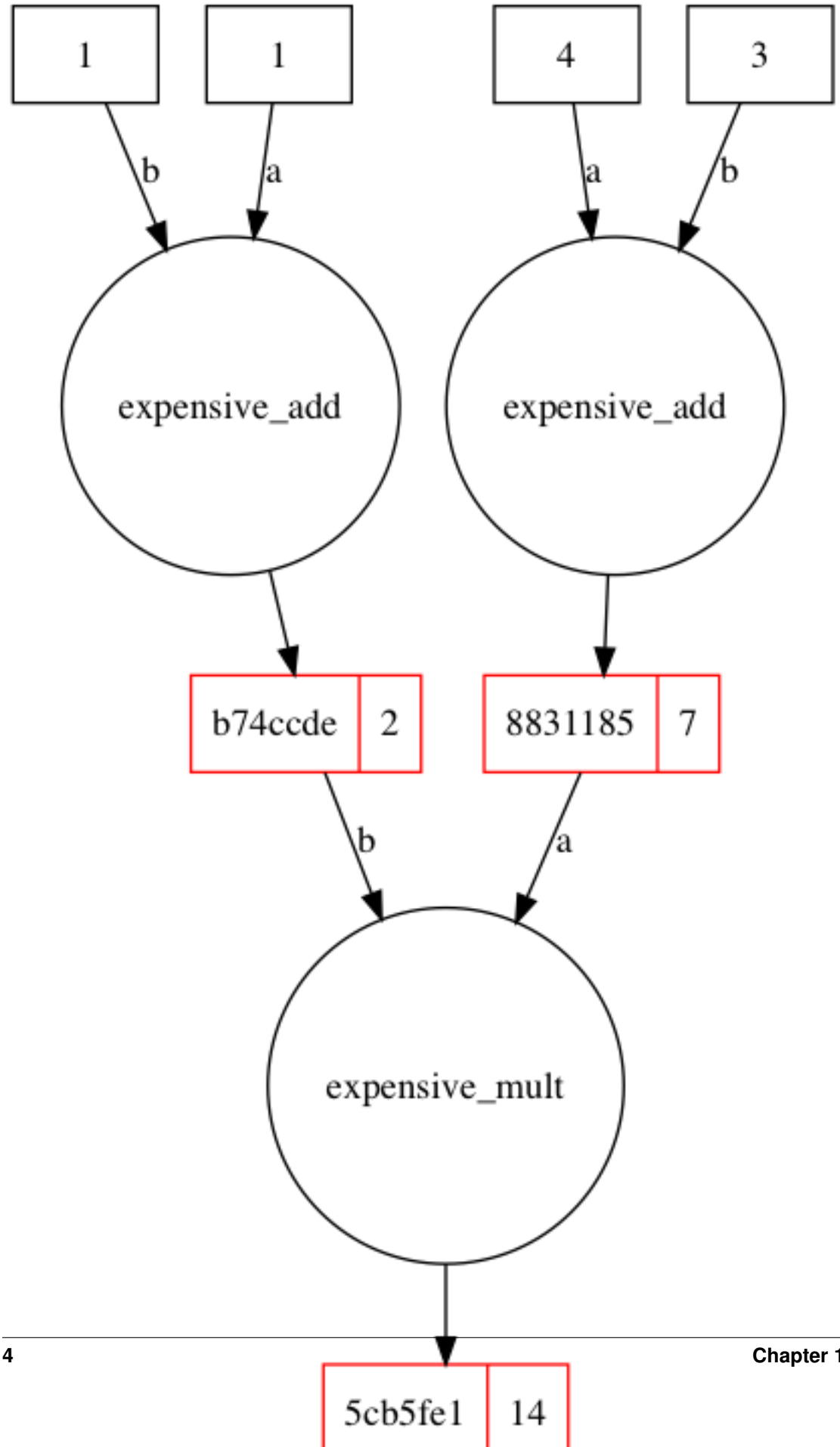
What that means in practice is that you can easily keep track of how artifacts (models, features, or any object or file) are created, where they are used, and have a central place to store and share these artifacts. This basic plumbing is required (or at least desired!) in any machine learning pipeline and project. `provenance` can be used standalone along with a build server to run pipelines or in conjunction with more advanced workflow systems (e.g. Airflow, Luigi).

CHAPTER 1

Example

For an explanation of this example please see the Introductory Guide.

```
import provenance as p

p.load_config(...)

import time

@p.provenance()
def expensive_add(a, b):
    time.sleep(2)
    return a + b


@p.provenance()
def expensive_mult(a, b):
    time.sleep(2)
    return a * b


a1 = expensive_add(4, 3)
a2 = expensive_add(1, 1)

result = expensive_mult(a1, a2)

vis.visualize_lineage(result)
```


1
1
4
3
b
a
a
b
expensive_add
expensive_add
b74ccde
2
8831185
7
b
a
expensive_mult
5cb5fe1
14

CHAPTER 2

Installation

For the base functionality:

```
pip install provenance
```

For the visualization module (which requires `graphviz` to be installed):

```
pip install provenance[vis]
```

For the SFTP store:

```
pip install provenance[sftp]
```

For everything all at once:

```
pip install provenance[all]
```

CHAPTER 3

Compatibility

`provenance` is currently only compatible with Python 3.5 and higher. Updating it to work with Python 2.7x should be easy, follow this ticket if you are interested in that.

3.1 Index

3.1.1 Introductory Guide

`provenance` is a Python library for function-level caching and provenance that aids in creating parsimonious pythonic pipelines™. By wrapping functions in the `provenance` decorator computed results are cached across various stores (disk, S3, SFTP) and provenance (i.e. lineage) information is tracked and stored in an artifact repository. A central artifact repostiory can be used to enable production pipelines, team collaboration, and reproducible results.

What that means in practice is that you can easily keep track of how artifacts (models, features, or any object or file) are created, where they are used, and have a central place to store and share these artifacts. This basic plumbing is required (or at least desired!) in any machine learning pipeline or project. `provenance` can be used standalone along with a build server to run pipelines or in conjunction with more advanced workflow systems (e.g. Airflow, Luigi).

Configuration

Before you can use `provenance` you need to configure it. The suggested way to do so is via the `load_config` function that takes a dictionary of settings. (ProTip: For a team environment take a look at ymlconf which does merging of configs, e.g. a shared config in a repo and a local one that isn't committed.)

We'll define our configuration map in YAML below. The config specifies a blobstore, where the cached values are stored, and an artifact repository, where the metadata is stored. In this example the cached values are stored on disk and the metadata is stored in a Postgres database.

```
In [1]: %load_ext yamlmagic

In [2]: %%yaml basic_config
        blobstores:
            disk:
```

```
                type: disk
                cachedir: /tmp/provenance-intro-artifacts
                read: True
                write: True
                delete: True
        artifact_repos:
            local:
                type: postgres
                db: postgresql://localhost/provenance-intro
                store: 'disk'
                read: True
                write: True
                delete: True
                # this option will create the database if it doesn't exist
                create_db: True
        default_repo: local
```

```
<IPython.core.display.Javascript object>
```

```
In [3]: import provenance as p


        p.load_config(basic_config)
```

```
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running stamp_revision  -> e0317ab07ba4
```

```
Out[3]: <provenance.repos.Config at 0x10fbd7b00>
```

Basic Example

To introduce the concepts we'll take a look at a small function that has been decorated with `provenance`.

Now lets define some decorated functions...

```
In [4]: import time

        @p.provenance()
        def expensive_add(a, b):
            time.sleep(2)
            return a + b


        @p.provenance()
        def expensive_mult(a, b):
            time.sleep(2)
            return a * b
```

```
In [5]: %%time
        result = expensive_add(4, 3)
        print(result)
```

```
7
CPU times: user 24.4 ms, sys: 14.8 ms, total: 39.2 ms
Wall time: 2.06 s
```

As expected, we have a slow addition function. To see the effect of the caching we can repeat the same function invocation:

```
In [6]: %%time
        result = expensive_add(4, 3)
        print(result)
```

```
7
CPU times: user 4.06 ms, sys: 1.62 ms, total: 5.69 ms
Wall time: 5.61 ms
```

Visualization

If you have used any caching/memoization decorator or library (e.g. `joblib`) then this is old hat to you. What is different is how the provenance of this result is recorded and how it can be accessed. For example, with the same result we can visualize the associated lineage:

```
In [7]: import provenance.vis as vis

        vis.visualize_lineage(result)
```

What is an artifact?

In the above visualization the `artifact` is outlined in red with the artifact `id` on the left and the `value` (result) on the right. How is this possible and what exactly is an `artifact`? Well, the `result` is not a raw `7` but rather is an `ArtifactProxy` which wraps the `7`.

```
In [8]: result
```

```
Out[8]: <provenance.ArtifactProxy(88311853d7c885426fc5686543fe1412c62c0aac) 7 >
```

The hash in the parenthesis is the `id` of the artifact which is a function of the name of the function used and the inputs used to produce the value.

All properties, methods, and operations called against this object will be proxied to the underlying value, `7` in this case. You can treat the result as you normally would:

```
In [9]: result + 3
```

```
Out[9]: 10
```

The one exception to this is the `artifact` property which returns the `Artifact`.

```
In [10]: artifact = result.artifact
         artifact
```

```
Out[10]: <provenance.Artifact(88311853d7c885426fc5686543fe1412c62c0aac)>
```

On this `artifact` is where you will find the provenance information for the value.

```
In [11]: # What function what used to create artifact?
         artifact.fn_module, artifact.fn_name
```

```
Out[11]: ('__main__', 'expensive_add')
```

```
In [12]: # What inputs were used?
         artifact.inputs
```

```
Out[12]: {'kargs': {'a': 4, 'b': 3}, 'varargs': ()}
```

This information is what powered the visualization above.

Each artifact has additional information attached to it, such as the `id`, `value_id`, `value`, and `run_info`. `run_info` captures information about the environment when the artifact was created:

```
In [13]: artifact.run_info
```

```
Out[13]: {'created_at': datetime.datetime(2017, 4, 30, 23, 59, 7, 29629),
          'host': {'cpu_count': 8,
           'machine': 'x86_64',
           'nodename': 'lambda',
           'platform': 'Darwin-16.5.0-x86_64-i386-64bit',
           'processor': 'i386',
           'release': '16.5.0',
           'system': 'Darwin',
           'version': 'Darwin Kernel Version 16.5.0: Fri Mar  3 16:52:33 PST 2017; root:xnu-3789.51.2
          'id': 'fff2aa27f17612b97cb7b9443bdfe0c8377343f1',
          'process': {'cmdline': ['/Users/bmabey/anaconda/envs/provenance-dev/bin/python',
            '-m',
            'ipykernel',
            '-f',
            '/Users/bmabey/Library/Jupyter/runtime/kernel-536bbad6-5cee-4aca-bdcb-cdeed82f2f8c.json'
           'cwd': '/Users/bmabey/w/provenance/docs/source',
           'exe': '/Users/bmabey/anaconda/envs/provenance-dev/bin/python3.5',
           'name': 'python3.5',
           'num_fds': 59,
           'num_threads': 9}}
```

You typically also want the `run_info` to include the `git` ref or build server job ID that was used to produce the artifact. This is easily done by using the provided `provenance.set_run_info_fn` hook. Please refer to the *API documenation* for details.

Loading Artifacts

Aside from calling decorating functions and having cached `artifacts` returned you can also explicitly load an `artifact` by it's `id`. This becomes useful in a team setting since you can have a shared store on S3, e.g. "Check out this amazing model I just built, load artifact 349ded4f...!". The other main usecase for this is in production settings. See the *Machine Learning Pipeline guide* for more details on that.

```
In [14]: # get the id of the artifact above
         artifact_id = artifact.id

         loaded_artifact = p.load_artifact(artifact_id)

         loaded_artifact
```

```
Out[14]: <provenance.Artifact(88311853d7c885426fc5686543fe1412c62c0aac)>
```

You can inspect the `loaded_artifact` as done above or load the value into a proxy:

```
In [15]: loaded_artifact.proxy()
```

```
Out[15]: <provenance.ArtifactProxy(88311853d7c885426fc5686543fe1412c62c0aac) 7 >
```

Alternatively, if you can load the proxy directly with `load_proxy`. `load_artifact` is still useful when you want the provenance and other metadata about an `artifact` but not the actual value.

```
In [16]: p.load_proxy(artifact_id)
```

```
Out[16]: <provenance.ArtifactProxy(88311853d7c885426fc5686543fe1412c62c0aac) 7 >
```

Multiple Functions Example

In any advanced workflow you will have a series of functions all pipelined together. For true provenance you will need to be able to track all artifacts to their source. This is why the computed results of functions are returned as

`ArtifactProxy`s, to enable the flow of artifacts to be tracked.

```
In [17]: a1 = expensive_add(4, 3)
         a2 = expensive_add(1, 1)

         result = expensive_mult(a1, a2)

         vis.visualize_lineage(result)
```

Note how the final artifact can be traced all the way back to the original inputs! As mentioned above this is enabled by the passing of the `ArtifactProxy` results, which means that for you to take advantage of this you must be passing the proxies throughout your entire pipeline. Best practice is to have your outer functions take basic Python data structures and then pass all resulting complex objects (e.g. scikit learn models) as `ArtifactProxy`s.

Underlying Storage

Under the hood all of the artifacts are being recorded to the artifact repository we configured above. A repository is comprised of a store that records provenance and metadata and a blobstore that stores the serialized artifact (`joblib` serialization is the default). While you rarely would need to access the blobstore files directly, querying the DB directly is useful and is a hallmark of the library.

```
In [18]: repo = p.get_default_repo()

         db = repo._db_engine

         import pandas as pd

         pandas.read_sql("select * from artifacts", db)
```

Out[18]:

	id	value_id	
0	88311853d7c885426fc5686543fe1412c62c0aac	bf68bf734f70b6643ce88133e90cf0f191aa704c	fff2a
1	b74ccde8b53c0e02b4f269df65a39fd6be5191f7	9df9e3b99f954bd73e4314d3006e8d5b21bdcb84	fff2a
2	5cb5fe1eeacbfe961bd03e63609ad64cf9c3ce12	483f2bffe8d13307d17ab5162facd02c09ff3b08	fff2a

	name	version	fn_module	fn_name	composite	value_id_duration
0	__main__.expensive_add	0	__main__	expensive_add	False	0.000047
1	__main__.expensive_add	0	__main__	expensive_add	False	0.000049
2	__main__.expensive_mult	0	__main__	expensive_mult	False	0.000030

	computed_at	added_at	inpu
0	2017-04-30 23:59:07.044956	2017-04-30 23:59:09.068981	
1	2017-04-30 23:59:09.257366	2017-04-30 23:59:11.272746	
2	2017-04-30 23:59:11.279636	2017-04-30 23:59:13.293786	[88311853d7c885426fc5686543fe1412c

	inputs_json	serializer	load_kwargs	dump_kwargs	cust
0	{'__varargs': [], 'b': 3, 'a': 4}	joblib	None	None	
1	{'__varargs': [], 'b': 1, 'a': 1}	joblib	None	None	
2	{'__varargs': [], 'b': {'id': 'b74ccde8b53c0e0...	joblib	None	None	

A few things to note here...

- The `input_artifact_ids` is an array of all of the artifact ids that were used in the input (even if nested in another data structure). This allows you to efficiently query for the progeny of a particular artifact.
- The `inputs_json` and `custom_fields` columns are stored as JSONB in Postgres allowing you to query the nested structure via SQL. For example, you could find all the addition artifacts that were created with a particular argument:

```
In [19]: pandas.read_sql("""select * from artifacts
                            where fn_name = 'expensive_add'
                            and (inputs_json ->> 'a')::int IN (1, 4)
                         """, db)
```

Out[19]:

	id	value_id	
0	88311853d7c885426fc5686543fe1412c62c0aac	bf68bf734f70b6643ce88133e90cf0f191aa704c	fff2a
1	b74ccde8b53c0e02b4f269df65a39fd6be5191f7	9df9e3b99f954bd73e4314d3006e8d5b21bdcb84	fff2a

	name	version	fn_module	fn_name	composite	value_id_duration	co
0	__main__.expensive_add	0	__main__	expensive_add	False	0.000047	
1	__main__.expensive_add	0	__main__	expensive_add	False	0.000049	

	computed_at	added_at	input_artifact_ids	
0	2017-04-30 23:59:07.044956	2017-04-30 23:59:09.068981	[]	{'__varargs': []
1	2017-04-30 23:59:09.257366	2017-04-30 23:59:11.272746	[]	{'__varargs': []

	custom_fields
0	{}
1	{}

```
In [25]: # the blobs are written to files with the names matching the hash of the content
         !ls -l /tmp/provenance-intro-artifacts | head
```

```
total 32
-rw------- 1 bmabey staff  60 Apr 30 17:59 1106f8117e45a6736cf21222b4539f8aa4d16197
-rw------- 1 bmabey staff   5 Apr 30 17:59 483f2bffe8d13307d17ab5162facd02c09ff3b08
-rw------- 1 bmabey staff 195 Apr 30 17:59 5cb5fe1eeacbfe961bd03e63609ad64cf9c3ce12
-rw------- 1 bmabey staff  60 Apr 30 17:59 88311853d7c885426fc5686543fe1412c62c0aac
-rw------- 1 bmabey staff   5 Apr 30 17:59 9df9e3b99f954bd73e4314d3006e8d5b21bdcb84
-rw------- 1 bmabey staff   5 Apr 30 17:59 b4f270ccdb89ba3cc7d413c99e364b779676d72d
-rw------- 1 bmabey staff  60 Apr 30 17:59 b74ccde8b53c0e02b4f269df65a39fd6be5191f7
-rw------- 1 bmabey staff   5 Apr 30 17:59 bf68bf734f70b6643ce88133e90cf0f191aa704c
```

Function Versioning

A potential pitfall of using `provenance` is getting stale cache results after updating your function. For example lets say we update our function definition:

```
In [21]: @p.provenance()
         def expensive_add(a, b):
             time.sleep(2)
             return a + b + a + b + a
```

```
In [22]: %%time
         expensive_add(4, 3)
```

```
CPU times: user 4.2 ms, sys: 1.95 ms, total: 6.15 ms
Wall time: 6.67 ms
```

```
Out[22]: <provenance.ArtifactProxy(88311853d7c885426fc5686543fe1412c62c0aac) 7 >
```

Noting the quick return time and incorrect result we see that we got an old (stale) cached result from our initial implementation. Rather than trying to determine if a function has been updated as `joblib` does the `provenance` library requires that you explicity version your functions to force new results to be computed. The rationale behind this is that it is quite hard (impossible in Python?) to tell when a funtion's definition or that of a helper function, which may be in another library, has semantically changed. For this reason the user of `provenance` must increment the version number of a function in the decorator:

```
In [23]: # the default version is 0 so lets set it to 1
         @p.provenance(version=1)
```

```
        def expensive_add(a, b):
            time.sleep(2)
            return a + b + a + b + a
```

```
In [24]: %%time
         expensive_add(4, 3)
```

```
CPU times: user 12.6 ms, sys: 3.25 ms, total: 15.8 ms
Wall time: 2.02 s
```

```
Out[24]: <provenance.ArtifactProxy(1106f8117e45a6736cf21222b4539f8aa4d16197) 18 >
```

While this may seem onerous in practice this is not a big problem in a mature code base once people are aware of this. For rapidly changing functions you may want to consider using the `use_cache=False` option temporarily as you iterate on your function. See the docs on the decorator for more information.

Next Steps

You now know the basics of the `provenance` library and can start using it now! Be aware that the `provenance` decorator takes a number of other options (such as `tags`) that can be quite helpful. Please refer to the *API documenation* for details.

3.1.2 Machine Learning Pipeline

** WORK IN PROGRESS ** This guide isn't complete but the code examples may be useful as is.

This guide assumes you are familiar with all the content in the *Introductory Guide*.

A typical machine learning pipeline consists of loading data, extracting features, training models and storing the models for later use in a production system or further analysis. In some cases the feature extraction process is quick and the features are transitory without any need of saving them independently of the finished trained model. Other times the features are a representation of the data that you wish to reuse in different settings, e.g. in a dashboard explaining predictions, ad-hoc analysis, further model development.

In the end a good deal of plumbing is required to wire up an app/service with the latest models and features in such a way that API calls can be traced back to the originating model, features, and even data sources. `provenance` abstracts much of this plumbing so you can focus on writing parsimonious pythonic pipelines™ with plain old functions.

```
In [1]: %load_ext yamlmagic
```

```
In [2]: %%yaml basic_config
        blobstores:
            disk:
                type: disk
                cachedir: /tmp/provenance-ml-artifacts
                read: True
                write: True
                delete: True
        artifact_repos:
            local:
                type: postgres
                db: postgresql://localhost/provenance-ml-guide
                store: 'disk'
                read: True
                write: True
                delete: True
                # this option will create the database if it doesn't exist
                create_db: True
        default_repo: local
```

```
<IPython.core.display.Javascript object>
```

```
In [3]: import provenance as p
```

```
        p.load_config(basic_config)
```

```
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running stamp_revision  -> e0317ab07ba4
```

```
Out[3]: <provenance.repos.Config at 0x11200ebe0>
```

```
In [4]: import numpy as np
        import pandas as pd
        import time
        from sklearn.utils import check_random_state
        import toolz as t
```

```
In [5]: @p.provenance()
        def load_data(query):
            # fetch something from the DB in real life...
            random_state = check_random_state(abs(hash(query)) // (10**10))
            return random_state.uniform(0, 10, 10)


        @p.provenance()
        def extract_features_a(data, hyperparam_a=5):
            time.sleep(2)
            rs = check_random_state(hyperparam_a)
            return data[0:5] + 1 + rs.rand(5)


        @p.provenance()
        def extract_features_b(data, hyperparam_x=10):
            time.sleep(2)
            rs = check_random_state(hyperparam_x)
            return data[5:] + 1 + rs.rand(5)


        @p.provenance()
        def build_model(features_a, features_b, num_trees=100):
            return {'whatever': 'special model with {} trees'.format(num_trees)}


        @p.provenance()
        def evaluate(model, data):
            return {'some_metric': 0.5, 'another_metric': 0.4}


        def pipeline(train_query='some query', valid_query="another query", hyperparam_a=5, hyperpara
            data = load_data("some query")
            features_a = extract_features_a(data, hyperparam_a)
            features_b = extract_features_b(data, hyperparam_x)
            model = build_model(data, features_a, features_b)

            validation_data = load_data("another query")
            evaluation = evaluate(model, validation_data)

            return {'features_a': features_a, 'features_b': features_b,
                    'model': model, 'evaluation': evaluation}
```

```
@p.provenance()
def make_decision(model, request):
    # make some sort of prediction, classification, with the model
    # to help make a 'decision' and return it as the result
    return {'prediction': 0.5, 'model': model.artifact.id}
```

TODO explain everything.. including the concept of artifact sets and how they simpify the building and deployment of models.

```
In [6]: def run_production_pipeline():
            with p.capture_set('production'):
                return pipeline()

In [7]: res = run_production_pipeline()

In [8]: res = p.load_set_by_name('production')

In [9]: res

Out[9]: ArtifactSet(id='08f3c7c6a84132faa155ca9996a26c4df92bd798', artifact_ids=frozenset({'24115211

In [10]: build_artifacts = res.proxy_dict(group_artifacts_of_same_name=True)

In [11]: build_artifacts.keys()

Out[11]: dict_keys(['__main__.load_data', '__main__.build_model', '__main__.extract_features_b', '__r

In [12]: model = build_artifacts['__main__.build_model']

In [13]: model

Out[13]: <provenance.ArtifactProxy(46268ac8c40932b63033b387aa0217974c82c717) {'whatever': 'special mo
```

3.1.3 Hashing and mutation

At the core of `provenance` is the ability to hash arbritary python data structures. `Artifact` ids are hashes of a function's metadata and the arguments for a given invocaiton. The `value_id` of an `Artifact` is the hash of the actual `value` object (the return value from the decorated function).

The hashing approach (and implementaiton) was created in `joblib` and it leverages `pickle` to walk any python data structures and produces a message digest of the resulting pickle bytecode. Therefore, any argument or return value to a `provenance` function must be picklable.

Hashing Semantics: Reference vs Value

`provenance` adopts value-based hashing semantics, which is in contrast to `joblib`'s referenced-based semantics. The best way to illustrate the difference is with some examples.

```
In [1]: a = [1, 2, 3]
        nested_references = [a, a]
        nested_values =  [[1, 2, 3], [1, 2, 3]]
```

Would you expect `hash(nested_references) == hash(nested_values)`? Lets take a look at the hashes with both appraoches.

```
In [2]: import joblib as jl
        import provenance as p

In [6]: print('Joblib Hashing')
        print('nested_references ',jl.hash(nested_references))
        print('nested_values     ', jl.hash(nested_values))
```

```
Joblib Hashing
nested_references  a9a5b8ad17431309cf00b14a45cedd7e
nested_values      70dc9c81d27cc89b76d98bcd52de04d0
```

```
In [7]: print('Provenance Hashing')
        print('nested_references ', p.hash(nested_references))
        print('nested_values     ', p.hash(nested_values))
```

```
Provenance Hashing
nested_references  ea971dde80510d0bd01ed7c9e246df52087ba7f4
nested_values      ea971dde80510d0bd01ed7c9e246df52087ba7f4
```

In the reference-based approach, as implemented by `joblib`, the references of nested objects are taken into account. With the value-based approach taken by `provenance` only the values of the datas strucrues involved are used to compute the hash.

Mutation and function behavior

While hahsing semantics may seem like an unimportant implementation detail choosing one or the other can result in stark differences in behavior. Consider the following:

```
In [8]: def mutating_nested(arrays):
            arrays[0][0] += 1
            return sum(map(sum, arrays))


        a = [1, 2, 3]
        nested_references = [a, a]
        nested_values =  [[1, 2, 3], [1, 2, 3]]

        {'with_references': mutating_nested(nested_references),
         'nested_values': mutating_nested(nested_values)}
```

```
Out[8]: {'nested_values': 13, 'with_references': 14}
```

The output of the function is not only dependent on the values but also on the references! If value-based hashing were to be used when memoizing this function you would result in incorrect cache hits. This is why `joblib` takes the conservative approach of hashing based on references. That way you can memoize the above function just fine and don't have to worry about false cache hits.

However, there are downsides in using reference-based hashing. In particular, you end up with more false cache misses resulting in needless recomputation. This was the original motivation for switching to use value-based semantics. Outside of that having functions that mutate state when caching is involved can lead to subtle bugs downstream (since a cache hit will prevent mutation of the inputs and potential downstream behavior). To avoid these types of issues and to allow for value-based hashing `provenance` only allows functions that do not mutate their inputs. In practice the majoirty of `provenance`ed functions end up being pure functions while a few functions that kick off the pipeline deterministicly fetch data from a data store.

When a `provenance`ed function mutates its input an error is thrown so the user can fix the mutation (typically by doing a `copy` or `deepcopy`):

```
In [9]: @p.provenance()
        def mutating_nested(arrays):
            arrays[0][0] += 1
            return sum(map(sum, arrays))


        a = [1, 2, 3]
        nested_references = [a, a]
```

```
        mutating_nested(nested_references)

---------------------------------------------------------------------------
ImpureFunctionError                       Traceback (most recent call last)
<ipython-input-9-5decafc47fd2> in <module>()
      8 nested_references = [a, a]
      9
---> 10 mutating_nested(nested_references)

<boltons.funcutils.FunctionBuilder-0> in mutating_nested(arrays)

/Users/bmabey/anaconda/envs/provenance-dev/lib/python3.5/site-packages/provenance-0.9.4.2+36.g7e45e2c
    293                 msg = "The {}.{} function modified arguments: ({})".format(
    294                     func_info['module'], func_info['name'], ",".join(modified_inputs))
--> 295                 raise ImpureFunctionError(msg)
    296
    297             if artifact_info_['composite']:

ImpureFunctionError: The __main__.mutating_nested function modified arguments: (arrays)
```

In this case you would avoid mutation like so:

```
In [10]: from copy import deepcopy

         @p.provenance()
         def nonmutating_nested(arrays):
             arrays = deepcopy(arrays)
             arrays[0][0] += 1
             return sum(map(sum, arrays))


         a = [1, 2, 3]
         nested_references = [a, a]

         nonmutating_nested(nested_references)
```

```
Out[10]: <provenance.ArtifactProxy(79e7a2510b6c796f510260816973f73178785384) 14 >
```

This will cause you to write your pipelines a bit different. For example, instead of creating a `scikit-learn` model in function and then fitting it another you would have a single function that creates the model and fits it all at once. Or you would decorate a function higher up in the stack and write mutating functions that get called from it. What is important is that inputs to functions are not mutated.

Warning about `copy` mutating the original object

In *rare* cases the act of `copy`ing of an object may mutate the orginal object, at least from the perspective of `pickle`. What is usually happenig is that underlying libraries are doing some sort of operation lazily prior to the `copy` which causes the `pickle` bytecode to be different. The only way to workaround this is to do a copy of the object before it is passed into a funciton with `provenance`.

Mutation of `ArtifactProxy`s

Once a cached result, i.e. an `ArtifactProxy`, is returned from a decorated function you should not mutate it before sending it into another decoarated function. By doing so you would be misrepresenting the value of the input, corrupting the provenance of the resulting artifact. Here is an example of what NOT to do and why it is a bad idea to mutate `ArtifactProxy`s:

```
In [11]: @p.provenance()
         def repeat(element, how_many):
             return [element] * how_many

         @p.provenance()
         def tracked_sum(array):
             return sum(array)

         numbers = repeat(1, 3)
         numbers[0] += 10
         result_after_mutation = tracked_sum(numbers)
         result_after_mutation
```

```
Out[11]: <provenance.ArtifactProxy(41ab5b4b904648618493b48c3f764f7abb0eab3a) 13 >
```

The `result_after_mutation` is correct but the lineage does not capture the mutation making the provenance incorrect:

```
In [16]: import provenance.vis as vis
         reloaded_result = p.load_proxy(result_after_mutation.artifact.id)

         vis.visualize_lineage(reloaded_result)
```

If you don't trust yourself (or your team) to not mutate `ArtifactProxys` then you can configure `provenance` to check that a proxy was not mutated prior to having it passed in. With this setting a `MutatedArtifactValueError` will be raised when the mutated artifact is passed to another decoarted function.

```
In [14]: p.set_check_mutations(True)

         numbers = repeat(1, 3)
         numbers[0] += 10
         tracked_sum(numbers)
```

```
---------------------------------------------------------------------------
MutatedArtifactValueError                 Traceback (most recent call last)
<ipython-input-14-a720157edb12> in <module>()
      3 numbers = repeat(1, 3)
      4 numbers[0] += 10
----> 5 tracked_sum(numbers)

<boltons.funcutils.FunctionBuilder-3> in tracked_sum(array)

/Users/bmabey/anaconda/envs/provenance-dev/lib/python3.5/site-packages/provenance-0.9.4.2+36.g7e45e2c
    261             inputs['filehash'] = value_id
    262
--> 263         input_hashes, input_artifact_ids = hash_inputs(inputs, repos.get_check_mutations(), i
    264
    265         id = create_id(input_hashes, **func_info['identifiers'])

/Users/bmabey/anaconda/envs/provenance-dev/lib/python3.5/site-packages/provenance-0.9.4.2+36.g7e45e2c
    127                 msg = msg.format(a.id, type(a.value), func_info.get('module'),
    128                                  func_info.get('name'), ",".join(arg_names))
--> 129                 raise MutatedArtifactValueError(msg)
    130
    131     input_hashes = {'kargs': kargs, 'varargs': tuple(varargs)}

MutatedArtifactValueError: Artifact 950970fbb2fd415674f14947091a8e7258082198, of type <class 'list'>
```

Keep calm!

All this talk of avoiding mutation may seem daunting and different from how you are used to writing functions in Python. It can take some time getting used to it but in the end most tasks in pipelines fit into the mold of a pure function and so it really isn't as bad as you might be thinking!

3.1.4 API

Primary API

provenance([version, repo, name, ...])	Decorates a function so that all inputs and outputs are cached.
load_artifact(artifact_id)	Loads and returns the `Artifact` with the `artifact_id` from the default repo.
load_proxy(artifact_id)	Loads and returns the `ArtifactProxy` with the `artifact_id` from the default repo.
ensure_proxies(*parameters)	Decorator that ensures that the provided parameters are always arguments of type ArtifactProxy.
promote(artifact_or_id, to_repo[, from_repo])	
provenance_set([set_labels, initial_set, ...])	
capture_set([labels, initial_set])	
create_set(artifact_ids[, labels])	
load_set_by_id(set_id)	Loads and returns the `ArtifactSet` with the `set_id` from the default repo.
load_set_by_name(set_name)	Loads and returns the `ArtifactSet` with the `set_name` from the default repo.
archive_file(filename[, name, ...])	(beta) Copies or moves the provided filename into the Artifact Repository so it can be used as an `ArtifactProxy` to inputs of other functions.

Configuration

from_config(config)	
load_config(config)	
load_yaml_config(filename)	
current_config()	
get_repo_by_name(repo_name)	
set_default_repo(repo_or_name)	
get_default_repo()	
set_check_mutations(setting)	
get_check_mutations()	
set_run_info_fn(fn)	This hook allows you to provide a function that will be called once with a process's *run_info* default dictionary.
get_use_cache()	
set_use_cache(setting)	
using_repo(repo_or_name)	

Utils

Fig. 1: Keep calm and pipeline on

is_proxy(obj)	
lazy_dict(thunks)	
lazy_proxy_dict(artifacts_or_ids[, ...])	Takes a list of artifacts or artifact ids and returns a dictionary whose keys are the names of the artifacts.

Visualization

visualize_lineage

Detailed Docs

Primary API

provenance.**provenance**(*version=0*, *repo=None*, *name=None*, *merge_defaults=None*, *ignore=None*, *input_hash_fn=None*, *remove=None*, *input_process_fn=None*, *archive_file=False*, *delete_original_file=False*, *preserve_file_ext=False*, *returns_composite=False*, *custom_fields=None*, *serializer=None*, *load_kwargs=None*, *dump_kwargs=None*, *use_cache=None*, *tags=None*, *_provenance_wrapper=<function provenance_wrapper>*)

Decorates a function so that all inputs and outputs are cached. Wraps the return value in a proxy that has an artifact attached to it allowing for the provenance to be tracked.

Parameters

version [int] Version of the code that is computing the value. You should increment this number when anything that has changed to make a previous version of an artifact outdated. This could be the function itself changing, other functions or libraries that it calls has changed, or an underlying data source that is being queried has updated data.

repo [Repository or str] Which repo this artifact should be saved in. The default repo is used when none is provided and this is the recommended approach. When you pass in a string it should be the name of a repo in the currently registered config.

name [str] The name of the artifact of the function being wrapped. If not provided it defaults to the function name (without the module).

returns_composite [bool] When set to True the function should return a dictionary. Each value of the returned dict will be serialized as an independent artifact. When the composite artifact is returned as a cached value it will be a dict-like object that will lazily pull back the artifacts as requested. You should use this when you need multiple artifacts created atomically but you do not want to fetch all the them simultaneously. That way you can lazily load only the artifacts you need.

serializer [str] The name of the serializer you want to use for this artifact. The built-in ones are 'joblib' (the default) and 'cloudpickle'. 'joblib' is optimized for numpy while 'cloudpickle' can serialize functions and other objects the standard python (and joblib) pickler cannot. You can also register your own serializer via the provenance.register_serializer function.

dump_kwargs [dict] A dict of kwargs to be passed to the serializer when dumping artifacts associated with this function. This is rarely used.

load_kwargs [dict] A dict of kwargs to be passed to the serializer when loading artifacts associated with this function. This is rarely used.

ignore [list, tuple, or set] A list of parameters that should be ignored when computing the input hash. This way you can mark certain parameters as invariant to the computed result. An example of this would be a parameter indicating how many cores should be used to compute

a result. If the result is invariant the number of cores you would want to ignore it so the value isn't recomputed when a different number of cores is used.

remove [list, tuple, or set] A list of parameters that should be removed prior to hashing and saving of the inputs. The distinction between this and the ignore parameter is that with the ignore the parameters the ignored parameters are still recorded. The motivation to not-record, i.e. remove, certain parameters usually driven by performance or storage considerations.

input_hash_fn [function] A function that takes a dict of all on the argument's hashes with the structure of {'kargs': {'param_a': '1234hash'}, 'varargs': ('deadbeef',..)}. It should return a dict of the same shape but is able to change this dict as needed. The main use case for this function is overshadowed by the ignore parameter and so this parameter is hardly ever used.

input_process_fn [function] A function that pre-processes the function's inputs before they are hashed or saved. The function takes a dict of all on the functions arguments with the structure of {'kargs': {'param_a': 42}, 'varargs': (100,..)}. It should return a dict of the same shape but is able to change this dict as needed. The main use case for this function is overshadowed by the remove parameter and the value_repr function.

merge_defaults [bool or list of parameters to be merged] When True then the wrapper introspects the argspec of the function being decorated to see what keyword arguments have default dictionary values. When a list of strings the list is taken to be the list of parameters you want to merge on. When a decorated function is called then the dictionary passed in as an argument is merged with the default dictionary. That way people only need to specify the keys they are overriding and don't have to specify all the default values in the default dictionary.

use_cache [bool or None (default None)] use_cache False turns off the caching effects of the provenance decorator, while still tracking the provenance of artifacts. This should only be used during quick local iterations of a function to avoid having to bump the version with each change. When set to None (the default) it defers to the global provenance use_cache setting.

custom_fields [dict] A dict with types that serialize to json. These are saved for searching in the repository.

tags [list, tuple or set] Will be added to custom_fields as the value for the 'tags' key.

archive_file [bool, defaults False] When True then the return value of the wrapped function will be assumed to be a str or pathlike that represents a file that should be archived into the blobstore. This is a good option to use when the computation of a function can't easily be returned as an in-memory pickle-able python value.

delete_original_file [bool, defaults False] To be used in conjunction with archive_file=True, when delete_original_file is True then the returned file will be deleted after it has been archived.

preserve_file_ext [bool, default False] To be used in conjunction with archive_file=True, when preserve_file_ext is True then id of the artifact archived will be the hash of the file contents plus the file extension of the original file. The motivation of setting this to True would be if you wanted to be able to look at the contents of a blobstore on disk and being able to preview the contents of an artifact with your regular OS tools (e.g. viewing images or videos).

Returns

ArtifactProxy Returns the value of the decorated function as a proxy. The proxy will act exactly like the original object/value but will have an artifact method that returns the Artifact associated with the value. This wrapped value should be used with all other functions that are wrapped with the provenance decorator as it will help track the provenance and also reduce redundant storage of a given value.

provenance.**load_artifact**(*artifact_id*)
Loads and returns the `Artifact` with the `artifact_id` from the default repo.

Parameters

artifact_id [string]

See also:

load_proxy

provenance.**load_proxy**(*artifact_id*)
Loads and returns the `ArtifactProxy` with the `artifact_id` from the default repo.

Parameters

artifact_id [string]

See also:

load_artifact

provenance.**ensure_proxies**(**parameters*)
Decorator that ensures that the provided parameters are always arguments of type ArtifactProxy.

When no parameters are passed then all arguments will be checked.

This is useful to use on functions where you want to make sure artifacts are being passed in so lineage can be tracked.

provenance.**promote**(*artifact_or_id*, *to_repo*, *from_repo=None*)

provenance.**provenance_set**(*set_labels=None*, *initial_set=None*, *set_labels_fn=None*)

provenance.**capture_set**(*labels=None*, *initial_set=None*)

provenance.**create_set**(*artifact_ids*, *labels=None*)

provenance.**load_set_by_id**(*set_id*)
Loads and returns the `ArtifactSet` with the `set_id` from the default repo.

Parameters

set_id [string]

See also:

load_set_by_name

provenance.**load_set_by_name**(*set_name*)
Loads and returns the `ArtifactSet` with the `set_name` from the default repo.

Parameters

set_name [string]

See also:

load_set_by_id, `load_set_by_labels`

provenance.**archive_file**(*filename*, *name=None*, *delete_original=False*, *custom_fields=None*, *preserve_ext=False*)
(beta) Copies or moves the provided filename into the Artifact Repository so it can be used as an `ArtifactProxy` to inputs of other functions.

Parameters

archive_file [bool, defaults False] When True then the return value of the wrapped function will be assumed to be a str or pathlike that represents a file that should be archived into the blobstore. This is a good option to use when the computation of a function can't easily be returned as an in-memory pickle-able python value.

delete_original [bool, defaults False] When delete_original_file True the file will be deleted after it has been archived.

preserve_file_ext [bool, default False] When True then id of the artifact archived will be the hash of the file contents plus the file extension of the original file. The motivation of setting this to True would be if you wanted to be able to look at the contents of a blobstore on disk and being able to preview the contents of an artifact with your regular OS tools (e.g. viewing images or videos).

Configuration

provenance.**from_config**(*config*)

provenance.**load_config**(*config*)

provenance.**load_yaml_config**(*filename*)

provenance.**current_config**()

provenance.**get_repo_by_name**(*repo_name*)

provenance.**set_default_repo**(*repo_or_name*)

provenance.**get_default_repo**()

provenance.**set_check_mutations**(*setting*)

provenance.**get_check_mutations**()

provenance.**set_run_info_fn**(*fn*)
This hook allows you to provide a function that will be called once with a process's *run_info* default dictionary. The provided function can then update this dictionary with other useful information you wish to track, such as git ref or build server id.

provenance.**get_use_cache**()

provenance.**set_use_cache**(*setting*)

provenance.**using_repo**(*repo_or_name*)

Utils

provenance.**is_proxy**(*obj*)

provenance.**lazy_dict**(*thunks*)

provenance.**lazy_proxy_dict**(*artifacts_or_ids*, *group_artifacts_of_same_name=False*)
Takes a list of artifacts or artifact ids and returns a dictionary whose keys are the names of the artifacts. The values will be lazily loaded into proxies as requested.

Parameters

artifacts_or_ids [collection of artifacts or artifact ids (strings)]

group_artifacts_of_same_name: bool (default: False)

If set to True then artifacts of the same name will be grouped together in

one list. When set to False an exception will be raised

Visualization (beta)

3.1.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/bmabey/provenance/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

provenance could always use more documentation, whether as part of the official provenance docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/bmabey/provenance/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *provenance* for local development.

1. Fork the *provenance* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/provenance.git
```

3. Setup your development environment. Assuming you have conda installed, the following commands can be used to create a development environment:

Initial environment creation

```
conda env create
source activate provenance-dev
pip install -r requirements.txt
pip install -r test_requirements.txt
```

Reactivating the environment after it has been created

```
source activate provenance-dev
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 provenance tests
$ python setup.py test
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring. Consider updating a guide or other documentation as well.
3. The pull request should pass the all the TravisCI builds. https://travis-ci.org/bmabey/provenance/pull_requests

3.1.6 History

0.12.0 (2018-10-08)

- Change default hashing algorithm to MD5 since SHA1 for performance considerations.
- Extends serialziaiton so the type used is inferred off of type.
- Makes the default serializer for Pandas DataFrames and Series to use Parquet.
- (breaking change!) Remove names from ArtifactSets, use a JSONB of labels instead.

- Doc tweaks.

0.11.0 (2018-08-23)

- Optional Google Storage support.
- Adds *persistent_connections* option to Postgres repo so NullPoll can be used when appropriate.
- Doc tweaks.

0.10.0 (2016-04-30)

- Change the default artifact name from the function name to the fully qualified module and function name. This will invalidate previously cached artifacts unless the names are migrated or explicitly set.
- Documentation! A start at least, more docstrings and guides will be added soon.
- Adds `use_cache` parameter and config option for when you only want to track provenance but not look for cache hits.
- Adds `check_mutations` option to prevent `Artifact` value mutations.
- Adds `tags` parameter to the `provenance` decorator for when you only want to track provenance but not look for cache hits.
- Adds experimental (alpha!) `keras` support.
- Adds a visualization module, pretty basic and mostly for docs and to illustrate what is possible.
- Adds `ensure_proxies` decorator to guard against non `ArtifactProxy` being sent to functions.

0.9.4.2 (2016-03-23)

- Improved error reporing when paramiko not present for SFTP store.

0.9.4.1 (2016-03-22) (0.9.4 was a bad release)

- Adds ability for a database and/or schema to be created when it doesn't exist.
- Adds SFTP blobstore as separate package provenance[sftp].
- Adds examples to illustrate how the library is used.

0.9.3 (2016-02-17)

- Patch release to fix packaging problems in 0.9.2.

0.9.2 (2016-02-17)

- Adds archive_file feature.

0.9.1 (2015-10-05)

- Python versions now supported: 2.7, 3.3, 3.4, 3.5

0.9.0 (2015-10-05)

- First release on PyPI. Basic functionality but lacking in docs.

Index