Noisify Documentation

Release 1.0.0

Declan Crew

Apr 24, 2019

Contents

Release v1.0.0. (*Installation*)

Noisify is a simple light weight library for augmenting and modifying data by adding 'noise'.

Let's make some noise:

Add some human noise (typos, things in the wrong boxes etc.)

```
>>> from noisify.recipes import human_error
>>> test_data = {'this': 1.0, 'is': 2, 'a': 'test!'}
>>> human_noise = human_error(5)
>>> print(list(human_noise(test_data)))
[{'a': 'tset!', 'this': 2, 'is': 1.0}]
>>> print(list(human_noise(test_data)))
[{'a': 0.0, 'this': 'test!', 'is': 2}]
```

Add some machine noise (gaussian noise, data collection interruptions etc.)

```
>>> from noisify.recipes import machine_error
>>> machine_noise = machine_error(5)
>>> print(list(machine_noise(test_data)))
[{'this': 1.12786393038729, 'is': 2.1387080616716307, 'a': 'test!'}]
```

If you want both, just add them together

```
>>> combined_noise = machine_error(5) + human_error(5)
>>> print(list(combined_noise(test_data)))
[{'this': 1.23854334573554, 'is': 20.77848220943227, 'a': 'tst!'}]
```

Add noise to numpy arrays

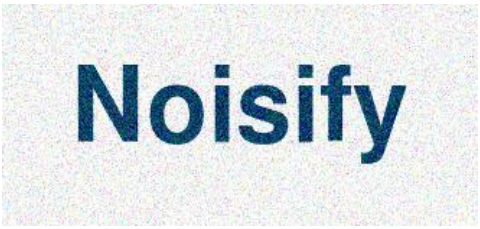
```
>>> import numpy as np
>>> test_array = np.arange(10)
>>> print(test_array)
[0 1 2 3 4 5 6 7 8 9]
>>> print(list(combined_noise(test_array)))
[[0.09172393 2.52539794 1.38823741 2.85571154 2.85571154 6.37596668
                  4.7135771  7.28358719 6.83600156 9.40973018]]
```

Read an image

```
>>> from PIL import Image
>>> test_image = Image.open(noisify.jpg)
>>> test_image.show()
```

And now with noise

```
>>> from noisify.recipes import human_error, machine_error
>>> combined_noise = machine_error(5) + human_error(5)
>>> for out_image in combined_noise(test_image):
...     out_image.show()
```



Noisify allows you to build flexible data augmentation pipelines for arbitrary objects. All pipelines are built from simple high level objects, plugged together like lego. Use noisify to stress test application interfaces, verify data cleaning pipelines, and to make your ML algorithms more robust to real world conditions.

CHAPTER 1

Features

Noisify provides data augmentation through a simple high level abstraction

- Build reporters to apply augmentation to any object, images, dataframes, database interfaces etc.
- Compose augmentations from configurable flaw objects
- Build recipes to deploy pipelines simply
- Everything is composable, everything is polymorphic

Noisify is built for Python 3+.

CHAPTER 2

The Basics

A brief high level guide of how to use noisify, mostly prose with illustrative examples.

2.1 Introduction

2.1.1 Background

Noisify is a project by Dstl (the Defence Science and Technology Laboratory). We are an executive agency of the UK Ministry of Defence.

Noisify was developed in part to expand the work done in image augmentation to other forms of data, and also to help test and perfect data cleaning and processing pipelines.

2.1.2 Copyright and usage information

2.2 Installation

Noisify is hosted on the PyPI central repo and can be installed as follows

```
$ pip install noisify
```

The only dependency is Python 3.5+ !

2.2.1 Build from Source

If you would prefer to install the latest version of the code, or perhaps to modify or contribute to it, the project is hosted on GitHub.

$ git clone https://github.com/dstl/noisify.git

$ cd noisify

$ pip install .

2.3 Quickstart

If Noisify is *installed* we can get to work with some examples!

2.3.1 Augmenting with recipes

Basic augmentation can be done very simply using basic recipes.

```
>>> from noisify.recipes import *
```

The built in recipes are designed to work with a wide variety of different object types. Let's give it a go with a simple Python dict.

```
>>> test_data = {'this': 1.0, 'is': 2, 'a': 'test!'}
>>> human_noise = human_error(5)
>>> print(human_noise(test_data))
<generator object Noisifier.generate_reports at 0x7f2d67e0f570>
```

Recipes create Noisifier objects, these objects then generate observations based on what they are given. To get a simple list, cast to list. The built in recipes take a single 'severity' argument. Bigger numbers lead to bigger effects on the data.

```
>>> print(list(human_noise(test_data)))
[{'a': 'tset!', 'this': 2, 'is': 1.0}]
```

You can also use a noisifier on a list of data.

```
>>> test_data = [{'test%d' % (index): "This is test run number %d" % index} for index␣
↪in range(5)]
>>> test_data
[{'test0': 'This is test run number 0'},
 {'test1': 'This is test run number 1'},
 {'test2': 'This is test run number 2'},
 {'test3': 'This is test run number 3'},
 {'test4': 'This is test run number 4'}]
>>> print(list(human_noise(test_data)))
[{'test0': 'This is test run number 0'},
 {'test1': 'This is test run number 1'},
 {'test2': 'hT iis testt unn umber2'},
 {'test3': 'This is test run number 3'},
 {'test4': 'This is test run number 4'}]
```

Let's have a closer look at what human_noise does.

```
>>> print(human_noise)
{'Noisifier': {'Reporter': {'Attributes': [],
               'Faults': [Fault: TypographicalFault {'likelihood': 0.5, 'severity': 0.
↪5},
                          Fault: ScrambleAttributes {'likelihood': 0.5, 'attribute_
↪identifiers': None}]}}}
```

That's a lot of information! The main thing to focus on is the 'Reporter' entry. This contains attributes (which we'll get to later) and Faults. Faults are the methods used to modify the incoming data steam, here you can see the two being used, typographical faults which scramble text and numbers, and attribute scrambling, this swaps values between keys in incoming dictionaries.

Let's have a look at another recipe.

```
>>> print(machine_error(5))
{'Noisifier': {'Reporter': {'Attributes': [],
               'Faults': [Fault: GaussianNoise {'sigma': 0.5},
                          Fault: InterruptionFault {'likelihood': 0.05}]}}}
```

Gaussian Noise is pretty self-explanatory, Interruption Fault leads to loss of data. Some values will be replaced with None.

Applying Gaussian noise to a string doesn't make much sense. That's no issue here though, if noisify doesn't know how to apply a given fault to a value, it won't try.

```
>>> print(list(machine_error(5)(test_data)))
[{'test0': 'This is test run number 0'},
 {'test1': 'This is test run number 1'},
 {'test2': None},
 {'test3': 'This is test run number 3'},
 {'test4': 'This is test run number 4'}]
```

2.3.2 Custom Noisifiers

Imagine we have a series of medical records, people's height and weight are generally measured in metres and kilograms. Occasionally however somebody has their weight entered in pounds and their height in inches. Let's say we've built a mechanism to find these wrongly entered values and we want to test it, how do we create this data? And more importantly, how do we tell when the noisifier has actually changed these values?

We need to create a custom noisifier.

First let's create some data.

```
>>> import random
>>> def build_patient_record():
...     return {'height': random.gauss(1.7, 0.1), 'weight': random.gauss(85, 10)}
>>> build_patient_record()
{'weight': 79.0702693462696, 'height': 1.690377702784025}
```

Now let's create some conversion functions for metric to imperial.

```
>>> def kilo_to_pounds(weight):
...     return weight * 2.205
...
>>> def metres_to_inches(height):
...     return height * 39.37
```

(continues on next page)

(continued from previous page)

```
...
>>>
```

Now let's create our *attributes*, this enables us to associate specific faults with specific values of the record. There are many different ways attributes can be looked up and modified, in this case we're using dictionary lookups.

```
>>> from noisify.attributes import DictValue
>>> from noisify.faults import UnitFault
>>> height = DictValue('height', faults=UnitFault(likelihood=0.25, unit_
↪modifier=metres_to_inches))
>>> weight = DictValue('weight', faults=UnitFault(likelihood=0.25, unit_modifier=kilo_
↪to_pounds))
```

Attributes take an identifier, this can be a key to a dictionary, or an attribute name of an object.

Now we build the reporter.

```
>>> from noisify.reporters import Reporter
>>> patient_reporter = Reporter(attributes=[height, weight])
```

That was easy, the reporter can be called on individual records, but won't accept data series.

```
>>> patient_reporter(build_patient_record())
{'height': 1.8157596382670191, 'weight': 199.97545102729777}
```

To apply more generally, create a noisifier.

```
>>> from noisify.recipes import Noisifier
>>> patient_noise = Noisifier(reporter=patient_reporter)
```

Let's build some data and noisify it.

```
>>> true_patients = [build_patient_record() for i in range(5)]
>>> true_patients
[{'height': 1.7831797462380368, 'weight': 84.70459461136014},
 {'height': 1.7661108421633465, 'weight': 87.20572747494349},
 {'height': 1.5047252739096044, 'weight': 102.7315276194823},
 {'height': 1.9371269447064758, 'weight': 78.54807087351945},
 {'height': 1.7624795973113694, 'weight': 76.47383227872784}]
>>> processed_patients = list(patient_noise(true_patients))
>>> processed_patients
[{'height': 1.7831797462380368, 'weight': 84.70459461136014},
 {'height': 1.7661108421633465, 'weight': 192.2886290822504},
 {'height': 59.24103403382112, 'weight': 102.7315276194823},
 {'height': 76.26468781309394, 'weight': 78.54807087351945},
 {'height': 1.7624795973113694, 'weight': 76.47383227872784}]
```

2.3.3 Report objects

Noisify reporters return report objects. These contain the observation made, but they also contain other information. These are stored as additional attributes on the object.

The faults triggered on an object can be retrieved through the triggered_faults attribute. Continuing from our example above:

```
>>> for patient in processed_patients:
...     print(patient.triggered_faults)
{'reporter': [], 'height': [], 'weight': []}
{'reporter': [], 'height': [], 'weight': [Fault: UnitFault {'unit_modifier':
↪<function kilo_to_pounds at 0x7f0b1fd17400>}]}
{'reporter': [], 'height': [Fault: UnitFault {'unit_modifier': <function metres_to_
↪inches at 0x7f0b1fd17488>}], 'weight': []}
{'reporter': [], 'height': [Fault: UnitFault {'unit_modifier': <function metres_to_
↪inches at 0x7f0b1fd17488>}], 'weight': []}
{'reporter': [], 'height': [], 'weight': []}
```

The ground truth is also stored.

```
>>> for patient in processed_patients:
...     print(patient.truth)
{'height': 1.7831797462380368, 'weight': 84.70459461136014}
{'height': 1.7661108421633465, 'weight': 87.20572747494349}
{'height': 1.5047252739096044, 'weight': 102.7315276194823}
{'height': 1.9371269447064758, 'weight': 78.54807087351945}
{'height': 1.7624795973113694, 'weight': 76.47383227872784}
```

2.3.4 Recipes

Recipes are simply factory functions for noisifiers. Consider the built in 'human_error' recipe.

```
>>> def human_error(scale):
...     return Noisifier(
...         reporter=Reporter(
...             faults=[TypographicalFault(likelihood=min(1, 0.1*scale), severity=0.
↪1*scale),
...                     ScrambleAttributes(scrambledness=0.1*scale)]
...         ),
...         faults=None
...     )
>>>
```

2.3.5 Combining reporters and noisifiers

The addition operator will combine reporters/ noisifiers into composites which will apply all faults from both original reporters.

```
>>> from noisify.recipes import machine_error, human_error
>>> print(machine_error(5))
{'Noisifier': {'Reporter': {'Attributes': [],
               'Faults': [Fault: GaussianNoise {'sigma': 0.5},
                          Fault: InterruptionFault {'likelihood': 0.05}]}}}
>>> print(human_error(5))
{'Noisifier': {'Reporter': {'Attributes': [],
               'Faults': [Fault: TypographicalFault {'likelihood': 0.5, 'severity': 0.
↪5},
                          Fault: ScrambleAttributes {'likelihood': 0.5, 'attribute_
↪identifiers': None}]}}}
>>> print(machine_error(5) + human_error(5))
{'Noisifier': {'Reporter': {'Attributes': [],
```

(continues on next page)

(continued from previous page)

```
              'Faults': [Fault: GaussianNoise {'sigma': 0.5},
                         Fault: InterruptionFault {'likelihood': 0.05},
                         Fault: TypographicalFault {'likelihood': 0.5, 'severity': 0.
↪5},
                         Fault: ScrambleAttributes {'likelihood': 0.5, 'attribute_
↪identifiers': None}]}}}
```

For custom faults and adding new datatype handlers to faults, see the *advanced* section.

2.4 Advanced Usage

This guide covers more advanced topics in noisify.

2.4.1 Defining Faults

Faults are defined by subclassing the base Fault class:

```
>>> from noisify.faults import Fault
>>> import random
>>> class AddOneFault(Fault):
...     def __init__(self, likelihood=1.0):
...         self.likelihood = likelihood
...
...     @register_implementation(priority=1)
...     def add_to_string(self, string_object):
...         return string_object + "1"
...
```

Let's unpack this definition.

We have the constructor, this behaves as expected. In this case adding a likelihood attribute to the object.

We also have an implementation. This describes how a fault will act on the data it is given.

2.4.2 Implementations And The Dispatch Queue

The power of noisify lies in its ability to take a large variety of different data types and intelligently apply noise. This mechanism is managed through the Dispatch Queue.

When an implementation is written for a given fault, it is decorated using the @register_implementation(priority=x) decorator. This gives the implementation its place within the queue. When a fault is called upon an unknown object it will attempt to apply each implementation in the queue to it in sequence. If all fail it will return the original object unaffected. Bigger numbers come first in the queue, so in the below example numpy_array will be called before python_numeric.

Let's look at some source code for an example

```
>>> class GaussianNoise(AttributeFault):
...     def __init__(self, sigma=0):
...         AttributeFault.__init__(self, sigma=sigma)
...         self.sigma = sigma
...         pass
...
```

(continues on next page)

(continued from previous page)

```
...     @register_implementation(priority=10)
...     def numpy_array(self, array_like_object):
...         import numpy as np
...         noise_mask = np.random.normal(scale=self.sigma, size=array_like_object.
↪size)
...         return array_like_object + noise_mask
...
...     @register_implementation(priority=1)
...     def python_numeric(self, python_numeric_object):
...         return random.gauss(python_numeric_object, self.sigma)
```

This fault will apply a gaussian noise filter to the input data. If the python_numeric implementation is called on a numpy array then a single random value will be added to the entire array, this is not desired behaviour. To fix this a second implementation with higher priority kicks in for numpy array like objects, this adds a separate offset to each value independently.

2.4.3 Dispatch Through Type Annotations

Dispatch should be handled through ducktyping where possible. However we recognise that cases exist where explicit dispatch on type is needed, this can be done through type annotations on the relevant implementations as follows.

```
>>> class TypographicalFault(AttributeFault):
...     @register_implementation(priority=1)
...     def impact_string(self, string_object: str):
...         return typo(string_object, self.severity)
...
...     @register_implementation(priority=1)
...     def impact_int(self, int_object: int):
...         return int(self.impact_string(str(int_object)) or 0)
```

2.4.4 Implementation Dispatch And Inheritance

Implementations are passed down through inheritance. The main example of this is the AttributeFault fault type, which adds a single implementation which will attempt to map the fault onto all elements of the input object. This can be given to a Reporter to cause it to apply the fault to all of its attributes. Negative priorities can be used in base class implementations to ensure that they are resolved last. Negative priorities should not be used in normal fault implementation annotation.

CHAPTER 3

The Community Guide

Our release process and community support process.

3.1 Support

3.1.1 File an Issue

If you spot a bug in noisify, or would like to suggest an additional feature, you can use our issue tracker on GitHub.

3.2 Community Updates

3.2.1 GitHub

The latest information on the status of the project is available on the GitHub repo.

3.2.2 Release and Version History

3.2.3 v1.0

- Initial release!

3.2.4 v0.9

- Looping behaviour for infinite generation
- Initial documentation

3.2.5 v0.8

- Ecosystem support for pandas, pil etc.

3.2.6 v0.7

- Type annotation dispatch added to priority dispatch mechanism

3.2.7 v0.6

- Renamed to Noisify
- First recipes

3.2.8 v0.5

- Priority dispatch mechanism first built

3.2.9 v0.4

- First reporter level faults
- Attribute introspection

3.2.10 v0.3

- Add composability by overloading addition to faults and reporters

3.2.11 v0.2

- Major rewrite from 0.1, focuses purely on fault generation.

3.2.12 v0.1

- Simulation and data augmentation together, too messy

3.3 Release Process and Rules

All code that adds new features will be required to implement unit tests to ensure that it does not introduce unexpected behaviour.

Pull requests that add new features will be very gladly accepted! Try and keep them small if possible. Larger requests will naturally take longer for us to review. Please avoid adding any dependencies, if you're adding support for an extra library then make sure this extra support is done in an optional way (importing a library in an implementation will skip the implementation if the library is not installed, please use this for ecosystem support).

Most importantly however, thank you for contributing back to Noisify!

Versioning follows the Semantic Versioning framework.

3.3.1 Major Releases

The first number in the version number is the major release (i.e `vX.0.0`). Changes to the core API that are not backwards compatible will result in a new major release version. Releases of this nature will be infrequent.

3.3.2 Minor Releases

Minor releases will change the second number of the version number (i.e `v0.Y.0`), these releases will add new features, but will be fully backwards compatible with prior versions.

3.3.3 Hotfix Releases

Hotfix releases will change the final number of the version (i.e `v0.0.Z`), these releases will consist of bug fixes between versions.

CHAPTER 4

The API Documentation / Guide

Full documentation of the noisify API

4.1 API reference

4.1.1 noisify.attribute_readers package

noisify.attribute_readers.attribute_readers module

Attribute Readers allow faults to be directed to specific attributes of an input object. These do not need to be literal attributes, they can be values in a dictionary or columns in a database for example, as long as they can be accessed via a key.

class noisify.attribute_readers.attribute_readers.**AttributeReader**(*attribute_identifier*, *faults=None*)

Bases: *noisify.helpers.fallible.Fallible*

The AttributeReader interface describes a mechanism to read and write values from an object

get_value(*truth_object*)
(Part of the interface) Must return the ground truth for the given attribute of the original object

measure(*truth_object*)
Takes a 'measurement' of the ground truth, applying all faults in the process

update_value(*output_object*, *new_value*)
(Part of the interface) Must update the new output object at the given attribute key with a new value

class noisify.attribute_readers.attribute_readers.**DictValue**(*attribute_identifier*, *faults=None*)

Bases: *noisify.attribute_readers.attribute_readers.AttributeReader*

Provides support for dictionary value lookups as attributes.

get_value(*truth_object*)
Queries the truth object using a dictionary lookup

update_value(*output_object*, *new_value*)
Sets using dictionary value assignment

class noisify.attribute_readers.attribute_readers.**ObjectAttribute**(*attribute_identifier*, *faults=None*)
Bases: *noisify.attribute_readers.attribute_readers.AttributeReader*

Provides support for literal object attributes as attributes.

get_value(*truth_object*)
Queries using getattr

update_value(*output_object*, *new_value*)
Sets using setattr

noisify.attribute_readers.inspection_strategies module

Inspection strategies are used by reporters to create attribute_readers for given objects when none are specified.

noisify.attribute_readers.inspection_strategies.**dictionary_lookup**(*unknown_dictionary*, *attribute_faults=None*)
Generates attribute_readers for each key/value pair of a given dictionary, enables reporters to map faults across dictionaries without further specification.

noisify.attribute_readers.inspection_strategies.**object_attributes_lookup**(*unknown_object*, *attribute_faults=None*)
Generates attribute_readers for each attribute of a given object, enables reporters to map faults across objects without further specification. Ignores methods and private attributes marked with '_'.

4.1.2 noisify.faults package

noisify.faults.fault module

The base classes for faults.

class noisify.faults.fault.**AttributeFault**(**args*, ***kwargs*)
Derived base class for attribute_readers, adds mapping behaviour which enables attribute faults to be added at higher levels of data representation.

For example:

```
>>> from noisify.faults import GaussianNoise
>>> noise = GaussianNoise(sigma=0.5)
>>> noise.impact(100)
100.66812113455995
>>> noise.impact({'A group': 100, 'of numbers': 123})
{'of numbers': 122.83439465953323, 'A group': 99.69284150349345}
```

condition(*triggering_object*)
Overrides the condition method to be constitutively active at the initial mapping stage.

Parameters **triggering_object** –

Returns

map_fault(*truth_object*)
Attempts to apply the fault to all subitems of the given object, in practice this means calling the fault on all values of a dict.

Parameters **truth_object** –

Returns

class noisify.faults.fault.**Fault**(**args*, ***kwargs*)
Fault base class.

Requires implementations to be registered in its subclasses. Subclasses register implementations with the "register_implementation(priority=x)" decorator.

All implementations will be attempted using a try except loop which will except Type, Attribute and Import errors. If no implementations succeed, the Fault will return the original object, unchanged.

By default faults are constitutively active, this can be overridden at instantiation by providing a 'likelihood' keyword argument with a probability of activation as a float.

Example Usage:

```
>>> class AddOneFault(Fault):
...     def condition(self, triggering_object):
...         return True
...
...     @register_implementation(priority=2)
...     def make_uppercase(self, lowercase_string):
...         return lowercase_string.upper()
...
...     @register_implementation(priority=1)
...     def add_to_int_string(self, integer_object):
...         return int(str(integer_object) + "1")
...
>>> adder = AddOneFault()
>>> adder.impact("testing priority")
'TESTING PRIORITY'
>>> adder.impact(1234)
12341
```

This decorator will also honour any type hints in the decorated function.

Example:

```
>>> class AddOneFault(Fault):
...     @register_implementation(priority=1)
...     def make_uppercase(self, lowercase_string: str):
...         print('Called uppercase function')
...         return lowercase_string.upper()
...
...     @register_implementation(priority=2)
...     def add_to_int_string(self, integer_object: int):
...         print('Called integer adding function')
...         return int(str(integer_object) + "1")
...
>>> adder = AddOneFault()
>>> adder.impact("testing annotation")
Called uppercase function
'TESTING ANNOTATION'
>>> adder.impact(1234)
```

(continues on next page)

(continued from previous page)

```
Called integer adding function
12341
```

apply(*not_faulted_object*)
Applies the fault to an object, returns self and the new object if the activation condition is met.

Parameters **not_faulted_object** –

Returns self or None, impacted_object

condition(*triggering_object*)
Base condition method, applies fault either constitutively or according to a likelihood argument at instantiation.

Parameters **triggering_object** – Can be used to create object-type dependant activation in overridden methods

Returns Boolean of whether or not the fault applies

impact(*impacted_object*)
Attempts to apply the fault to an object, cycles through all implementations until one succesfully executes. If none execute it will return the original object, unharmed.

Parameters **impacted_object** –

Returns

noisify.faults.attribute_faults module

Basic attribute level faults, mostly basic numeric manipulations. A good place to get started.

class noisify.faults.attribute_faults.**CalibrationFault**(*offset=0*)
Subclass of UnitFault, adds a constant offset to the input numeric.

```
>>> calibration_fault = CalibrationFault(10)
>>> calibration_fault.impact(200)
210
```

class noisify.faults.attribute_faults.**GaussianNoise**(*sigma=0*)
Applies a gaussian noise to a numeric object.

```
>>> noise = GaussianNoise(sigma=0.5)
>>> noise.impact(27)
28.08656007204934
```

Numpy arrays like objects apply noise separately to each element.

```
>>> import numpy as np
>>> test = np.arange(5)
>>> noise.impact(test)
array([0.56983913, 0.92835482, 2.36240306, 2.87398093, 3.92371237])
```

numpy_array(*array_like_object*)
Support for numpy arrays

pandas_df(*data_frame*)
Support for pandas dataframes

pil_image(*image_object*)
Support for PIL image objects, undetectable unless high sigma given

python_numeric(*python_numeric_object*)
Support for basic Python numeric types

class noisify.faults.attribute_faults.**InterruptionFault**(*likelihood=0*)
Replaces input with None, activates according to set likelihood.

```
>>> interrupt = InterruptionFault(1.0)
>>> interrupt.impact('This can be anything')
```

```
>>>
```

impact_truth(*truth*)
Basic behaviour, just returns None!

numpy_array(*array_like_object*)
Support numpy arrays and pandas dataframes

pil_image(*image_object*)
Support for PIL images

class noisify.faults.attribute_faults.**TypographicalFault**(*likelihood=0*, *severity=0*)
Applies a rough misspelling to the input using faults.utilities.typo()

```
>>> from noisify.faults import TypographicalFault
>>> typo_fault = TypographicalFault(1.0, 1)
>>> typo_fault.impact('This is the original text')
'Thhiisith heiginal etxt'
```

impact_float(*float_object: float*)
Scrambles floats, ensures still valid before returning

impact_int(*int_object: int*)
Scrambles ints

impact_string(*string_object: str*)
Scrambles strings

class noisify.faults.attribute_faults.**UnitFault**(*likelihood=1.0*, *unit_modifier=None*)
Applies a user defined adjustment to the input numeric object. Useful for modelling unit errors.

```
>>> def celsius_to_kelvin(celsius_value):
...     return celsius_value + 273.15
...
>>> kelvin_fault = UnitFault(unit_modifier=celsius_to_kelvin)
>>> kelvin_fault.impact(21)
294.15
```

numeric(*numeric_object*)
Support for basic numeric types, including dataframes and numpy arrays

pil_image(*image_object*)
Support for PIL images

noisify.faults.attribute_faults.**get_mode_size**(*mode*)
Converts a PIL image mode string into a dimension cardinality

noisify.faults.attribute_faults.**typo**(*string*, *severity*)
Roughly rearranges string with the occasional missed character, based on applying a gaussian noise filter to the string character indexes and then rounding to the closest index.

Parameters

- **string** –
- **severity** –

Returns mistyped string

noisify.faults.report_faults module

Report level faults typically comprise faults that depend on multiple attributes. For example switching attribute values.

class noisify.faults.report_faults.**ConfuseSpecificAttributes**(*attribute1*, *attribute2*, *likelihood=0*)
Swaps a specific pair of attribute values in a given object

impact_dictionary(*dictionary_object*)
Support for dictionary like objects

class noisify.faults.report_faults.**LoseEntireReport**(*likelihood=0*)
Replaces entire report with None, activates according to set likelihood.

impact_truth(*truth*)
Just returns None!

class noisify.faults.report_faults.**ScrambleAttributes**(*likelihood=0.1*, *attribute_identifiers=None*)
Switches the values of different attribute_readers within the object. By default it will apply to all attribute_readers.

impact_dictionary(*dictionary_object*)
Swaps random values in a dictionary

numpy_array(*array_like*)
Swaps random cells in a numpy array-like object

pillow_image(*pillow_image*)
Swaps random pixels in a PIL Image

noisify.faults.utilities module

Fault utility functions, general purpose code that is used by multiple functions.

noisify.faults.utilities.**dropped_scramble**(*collection*, *scrambledness*, *confusion_range*)
Scrambles objects in a collection, with a chance to lose some objects

Parameters

- **collection** –
- **scrambledness** – How likely two objects are to be switched
- **confusion_range** – How far apart objects can be confused with one another

Returns

noisify.faults.utilities.**scramble**(*collection*, *scrambledness*, *confusion_range*)
Scrambles the order of objects in a collection using a gaussian distribution, can lead to duplicate objects

Parameters

- **collection** –
- **scrambledness** – How likely two objects are to be switched
- **confusion_range** – How far apart objects can be confused with one another

Returns

4.1.3 noisify.helpers package

noisify.helpers.fallible module

class noisify.helpers.fallible.**Fallible**(*faults*)
Bases: object

Fallible mixin, adds faults to an object as well as getters and setters. Also provides methods for applying faults to an object.

add_fault(*fault*)
Add a fault to the fallible object

Parameters **fault** –

Returns

apply_all_faults(*incompletely_flawed_object*)
Runs through the fallible objects faults and applies them to an object, returns activated faults as well as the finished object

Parameters **incompletely_flawed_object** –

Returns

noisify.helpers.fallible.**evaluate_faults**(*faults*)
Enables faults to be given as a single fault, or a list of faults, or a function to generate a fault or list of faults, to the instantiation of the fallible object.

Parameters **faults** –

Returns

noisify.helpers.saved_init_statement module

class noisify.helpers.saved_init_statement.**SavedInitStatement**(**args*, ***kwargs*)
Bases: object

Init statement saving mixin, introspects on object instantiation arguments and saves them to the final object

4.1.4 noisify.recipes package

Default recipes, these are extremely simple and are mainly to provide examples for developing your own code.

noisify.recipes.default_recipes.**human_error**(*scale*)
Simple example Noisifier recipe, applies typos and attribute scrambling to the input depending on the scale given, recommended scale range from 1-10

noisify.recipes.default_recipes.**machine_error**(*scale*)
Simple example Noisifier recipe, applies gaussian noise and occasional interruptions to the input depending on the scale given, recommended scale range from 1-10

4.1.5 noisify.reporters package

noisify.reporters.reporter module

class noisify.reporters.reporter.**Reporter**(*attributes=None*, *attribute_type=<function dictionary_lookup>*, *faults=None*)
Bases: *noisify.helpers.fallible.Fallible*

The most important class in Noisify!

Reporters define how objects should be changed. They can be as specific or a general as needed.

create_report(*truth_object*, *identifier=None*)
Calling the reporter object directly on an object will call this method.

Parameters

- **truth_object** – Anything
- **identifier** – Optional identifier for the output report, defaults to a serial integer

Returns A report for the given input object

get_attribute_by_id(*attribute_identifier*)
Getter method for report attribute_readers

static merge_attributes(*report1*, *report2*)
Merges attribute_readers between two reporters, used for reporter addition

noisify.reporters.series module

class noisify.reporters.series.**Noisifier**(*reporter=None*, *faults=None*)
Bases: *noisify.helpers.fallible.Fallible*

The Noisifier class handles pipelining objects through an underlying reporter class, it can also be configured to apply faults at the pipeline level, such as confusing elements from one object to another.

get_series(*source_truths*, *key=None*, *loop=False*)
Calling the noisifier object directly on an object will call this method.

Parameters

- **source_truths** – a series of objects (or a single object)
- **key** – function which will extract a name from each object to be used as an

identifier for the resultant report. :param loop: whether to generate indefinitely by looping over the source truths :return: a report generator

noisify.reporters.series.**is_atom**(*unknown_object*)
Determines whether an object is an atom or a collection

Python Module Index

n

Index