Property Testing Pandas with Bulwark

Zax Rosenberg, CFA
Senior Data Scientist @ SPINS
github.com/ZaxR
www.zaxrosenberg.com

Agenda

- About Me
- About Property Testing
- About Bulwark
- Demo
- Contributing

About Me

- Senior Data Scientist @ SPINS
- Director of ChiPy's Mentorship Program
- Co-host of ChiPy's Project Night
- Blogger (zaxrosenberg.com/blog)
- Speaker (github.com/ZaxR/talks)

About Me

- NOT a testing expert

What is Property Testing?

Checking that some object has certain properties. For example:

```
    some_list = [1, 2, 3, 4]
```

One property of `some_list` is that its values are in a range of 1-4. Another is that it's mutable.

Why is it valuable?

- Testing reduces bugs/lowers development cost
- We don’t always have the exact data up front
- Property tests can be relatively fast to run
- Easy to include domain knowledge

Introducing Bulwark

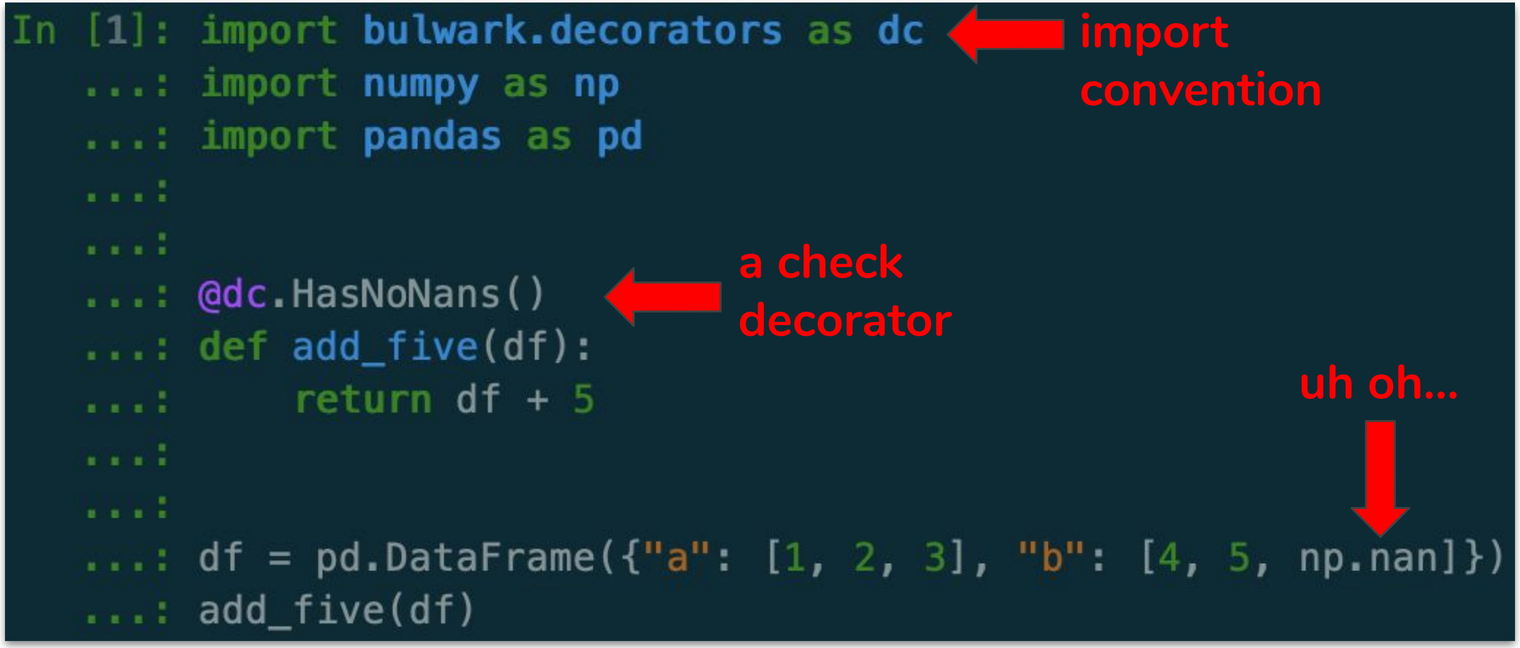
Introducing Bulwark

- Bulwark is an open-source library that lets you easily property test pandas dataframes.
- It’s designed to make it easier for data analysts and data scientists to test our own code.

Bulwark's Design

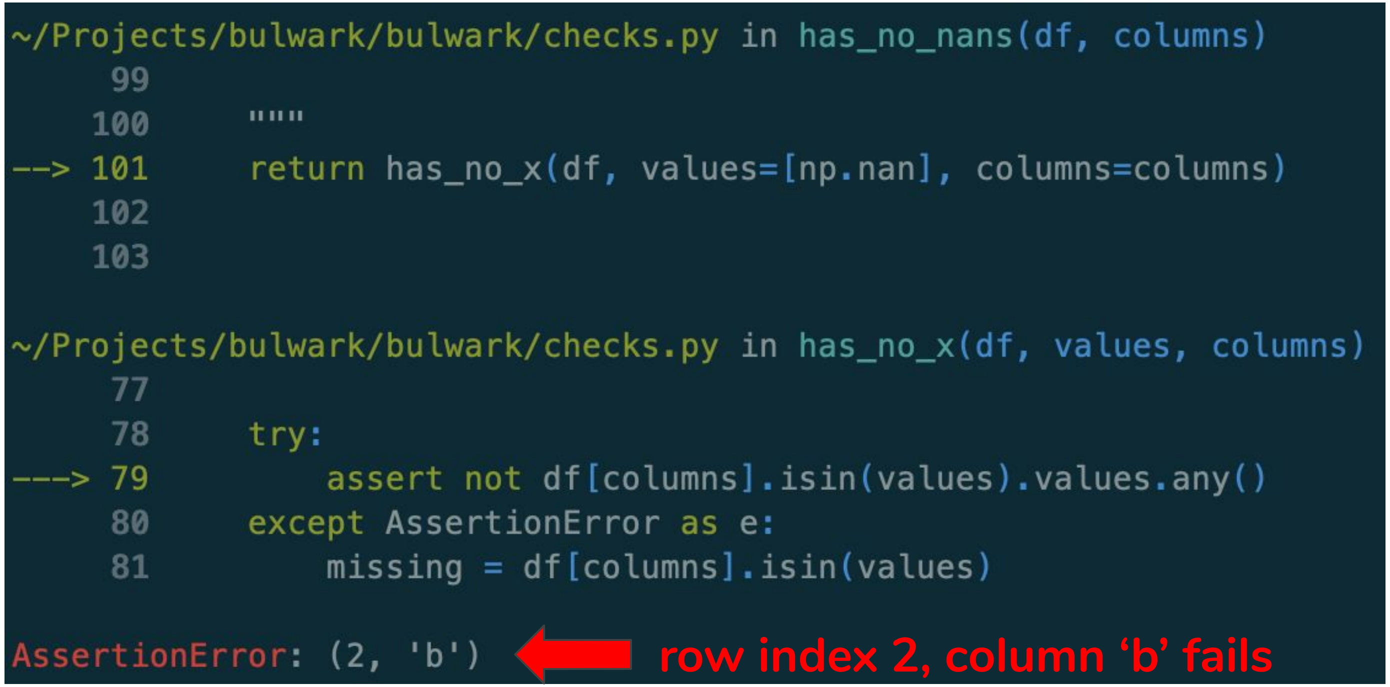
- Property tests are available as functions ("`checks`") and `decorators`.
- Each check:
    - Takes a pd.DataFrame and optional additional arguments,
    - Makes an assertion about the pd.DataFrame, and
    - Returns the original, unaltered pd.DataFrame.
- A failed checks raises an `AssertionError`, printing an informative message.
- Each `check` has an auto-magically-generated associated `decorator`, allowing you to make your assertions *outside* the actual logic of your code. This is a core benefit of Bulwark.

Quickstart - Input



```
In [1]: import bulwark.decorators as dc
   ...: import numpy as np
   ...: import pandas as pd
   ...:
   ...:
   ...: @dc.HasNoNans()
   ...: def add_five(df):
   ...:     return df + 5
   ...:
   ...:
   ...: df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, np.nan]})
   ...: add_five(df)
```

Quickstart - Result



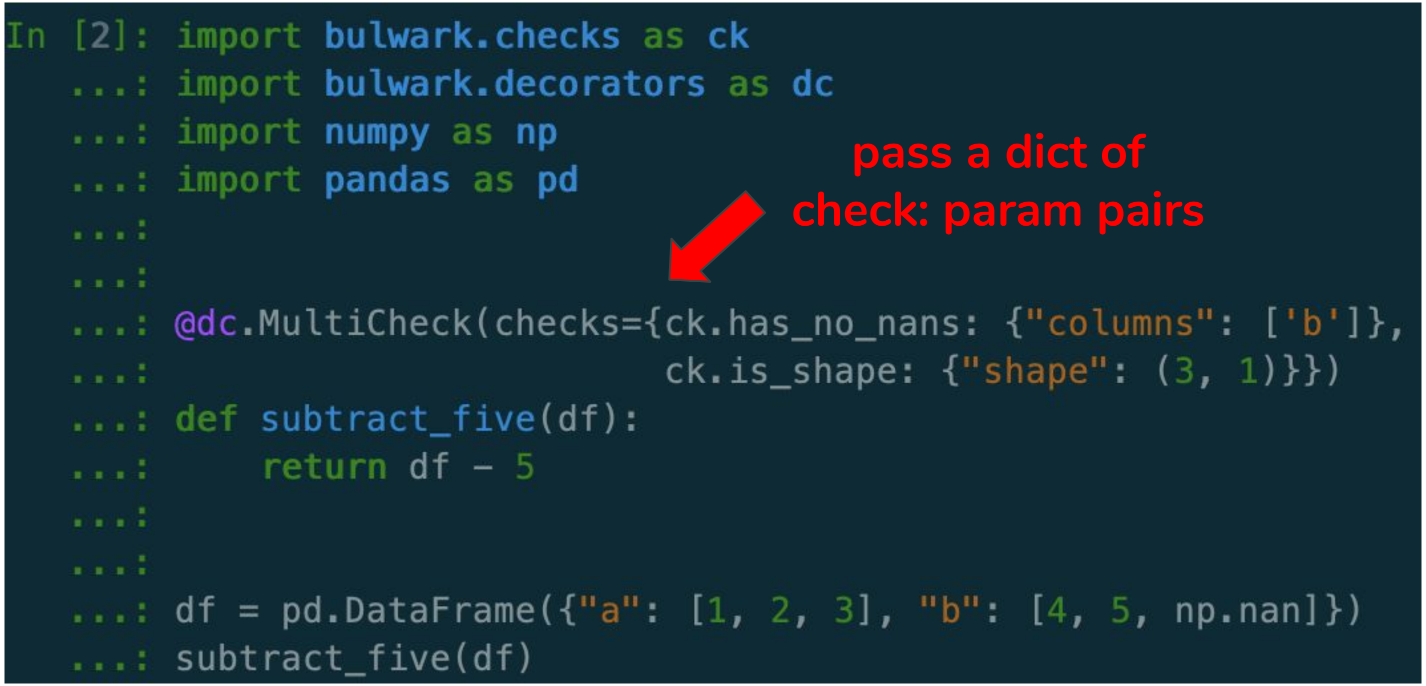
```
~/Projects/bulwark/bulwark/checks.py in has_no_nans(df, columns)
     99
    100     """
--> 101     return has_no_x(df, values=[np.nan], columns=columns)
    102
    103

~/Projects/bulwark/bulwark/checks.py in has_no_x(df, values, columns)
     77
     78     try:
---> 79         assert not df[columns].isin(values).values.any()
     80     except AssertionError as e:
     81         missing = df[columns].isin(values)

AssertionError: (2, 'b')
```

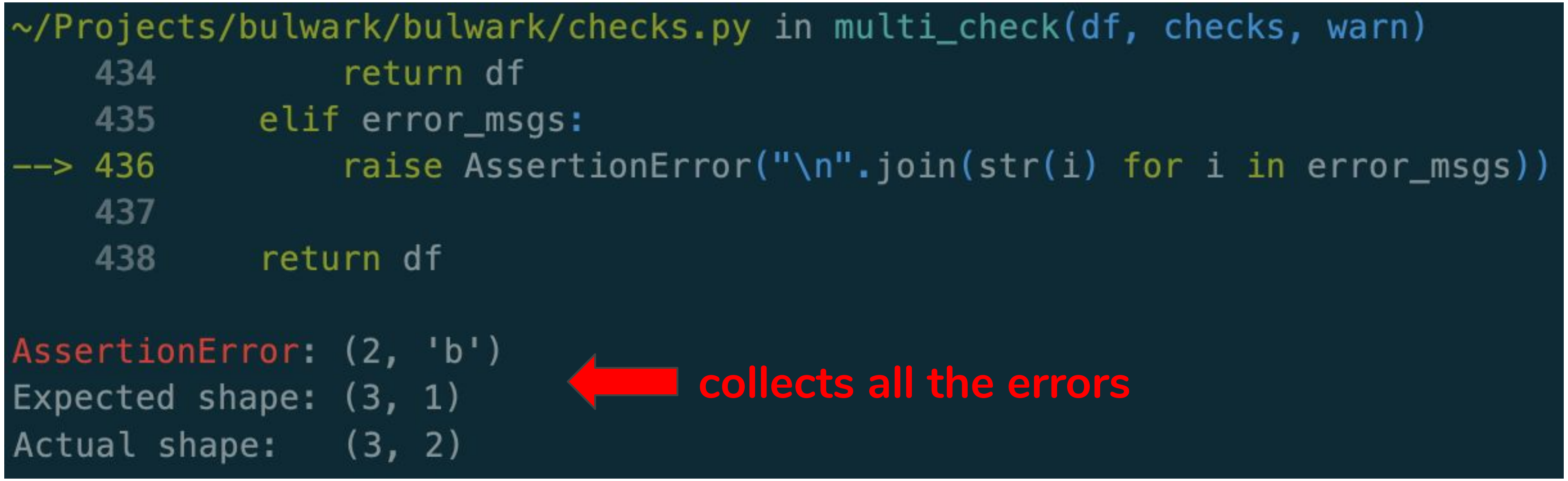
row index 2, column ‘b’ fails

What if I have multiple checks?



```
In [2]: import bulwark.checks as ck
   ...: import bulwark.decorators as dc
   ...: import numpy as np
   ...: import pandas as pd
   ...:
   ...:
   ...: @dc.MultiCheck(checks={ck.has_no_nans: {"columns": ['b']},
   ...:                        ck.is_shape: {"shape": (3, 1)}})
   ...: def subtract_five(df):
   ...:     return df - 5
   ...:
   ...:
   ...: df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, np.nan]})
   ...: subtract_five(df)
```

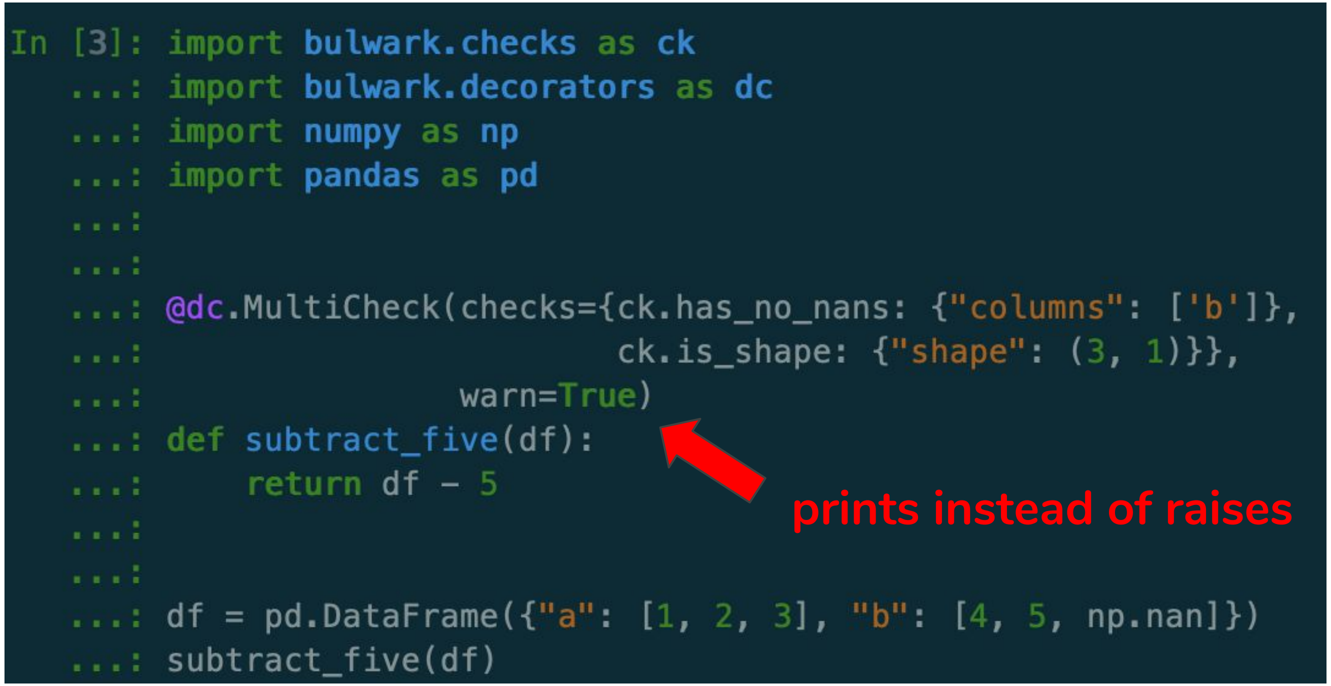
What if I have multiple checks?



```
~/Projects/bulwark/bulwark/checks.py in multi_check(df, checks, warn)
    434             return df
    435         elif error_msgs:
--> 436             raise AssertionError("\n".join(str(i) for i in error_msgs))
    437 
    438         return df

AssertionError: (2, 'b')
Expected shape: (3, 1)
Actual shape:   (3, 2)
```

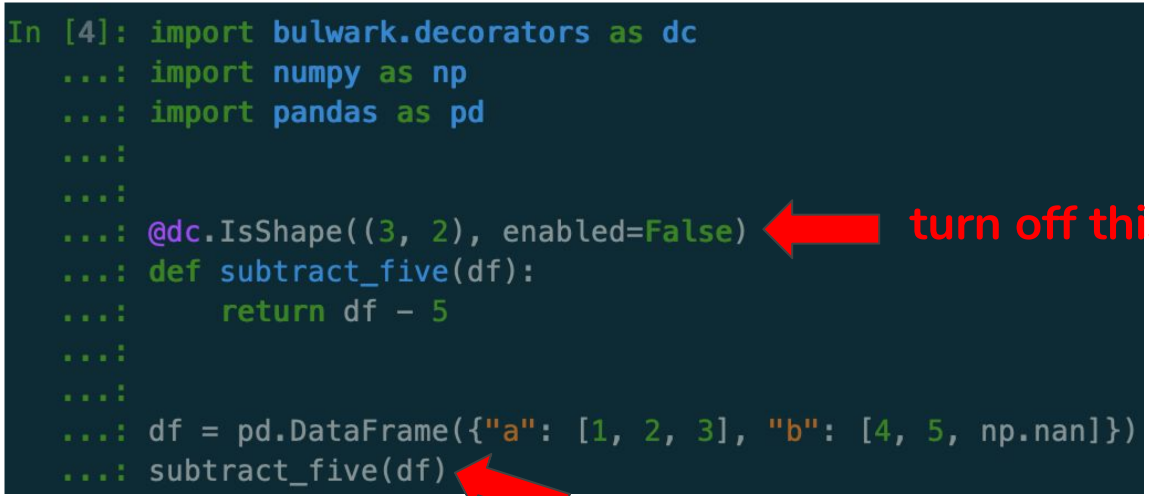
What if I don't want to raise errors?



```
In [3]: import bulwark.checks as ck
   ...: import bulwark.decorators as dc
   ...: import numpy as np
   ...: import pandas as pd
   ...:
   ...:
   ...: @dc.MultiCheck(checks={ck.has_no_nans: {"columns": ['b']},
   ...:                        ck.is_shape: {"shape": (3, 1)}},
   ...:                warn=True)
   ...: def subtract_five(df):
   ...:     return df - 5
   ...:
   ...:
   ...: df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, np.nan]})
   ...: subtract_five(df)
```

prints instead of raises

What about when I go to production?



```
In [4]: import bulwark.decorators as dc
   ...: import numpy as np
   ...: import pandas as pd
   ...:
   ...:
   ...: @dc.IsShape((3, 2), enabled=False)
   ...: def subtract_five(df):
   ...:     return df - 5
   ...:
   ...:
   ...: df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, np.nan]})
   ...: subtract_five(df)
```

turn off this check

doesn’t raise an error

Pro tip: set a centralized config variable that toggles all decorators’ statuses.

Demo Time!

Where should I use Bulwark?

- On ETL pipeline functions, especially E & L
  - Help enhance your understanding of the data upfront, even if you don’t do full EDA
  - Integration test by checking output
- In unit tests

How should I use Bulwark?

- Favor the decorator version within core code
    - Lets you disable/switch to warnings
    - Separates checks from code/business logic
- Use check version in unit tests

When should I use Bulwark?

- During development
- Maybe at run time.
  - If it’s acceptable for runs to fail
  - If the state should never be reached, and you can’t handle the error.

Who’s using Bulwark?

- 6,123 total downloads
- 256 downloads/month (excluding mirrors)
- Folks in > 45 countries

Contributing is easy, too!

- Very friendly to folks new to open source! Adding a new check is as easy as writing a single function.
- Full instructions available at: https://bulwark.readthedocs.io/en/latest/contributing.html

Find out more

- PyPI: https://pypi.org/project/bulwark/
- Read the Docs: https://bulwark.readthedocs.io
- GitHub: https://github.com/ZaxR/bulwark