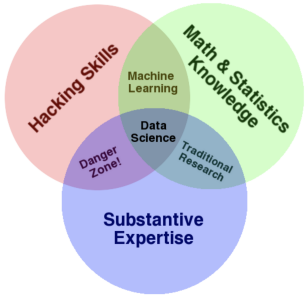
Getting Started Testing in Data Science

Jes Ford, PhD

Data Scientist



Jes Ford

Data Scientist at Recursion in Salt Lake City.

- Originally from Alaska, have followed the snow all around the western US/Canada
- PhD in Astrophysics from UBC, Vancouver
- Postdoc in Data Science at UW, Seattle
- like many data scientists, no formal training in software best practices

Drug discovery, reimagined through Artificial Intelligence
We are hiring data scientists, ML researchers, engineers, and more:
www.recursionpharma.com/careers

The plan

In [ ]:
```
def presentation():
    motivate_testing()
    introduce_testing_with_pytest()
    data_science_workflows()
    data_science_example_tests()
    wrap_up()
```

Why test?

- Tests can give you evidence that your code is working as expected
- Tests give you confidence to make changes without fear of breaking something
- Tests make other people trust your code more

Why *not* test?

Writing tests takes time!

The Struggle

As a data scientist I am constantly struggling with these competing goals:

- getting results as quickly as possible
- being as confident as possible that I've got the right answer

→ How do we balance these interests in the optimal way?

In this talk...

- I will *not* insist that you always write tests
- I will describe different scenarios I find myself in as a data scientist and how I try to be confident that my results are correct
- I will show you how to get started testing and share some tools for data science testing

Disclaimer

- I am not a testing expert or a software engineer
- "data science" covers a huge range of job duties and formal testing is less important in some of them (one-off analyses vs committing to production code base)

How do you know if your code is correct??

- manual sanity checks
- defensive programming
- tests

How do you know if your code is correct??

- manual sanity checks
- defensive programming: **assertions within the code**
- tests

In [1]:
```
# assertion example
def hello_to_all(list_of_names):
    assert len(list_of_names) > 0, 'There is no one here'
    print('Hello {}!'.format(', '.join(list_of_names)))
```

In [2]:
```
hello_to_all(['Parker', 'Missy', 'Taylor'])
```

```
Hello Parker, Missy, Taylor!
```

In [3]:
```
hello_to_all([])
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-3-976e84a8fe8e> in <module>
----> 1 hello_to_all([])

<ipython-input-1-1fb3826b63a8> in hello_to_all(list_of_names)
      1 # assertion example
      2 def hello_to_all(list_of_names):
----> 3     assert len(list_of_names) > 0, 'There is no one here'
      4     print('Hello {}!'.format(', '.join(list_of_names)))

AssertionError: There is no one here
```

Assertions

are a careful data scientist's best friend. This is your middle ground of checking for expected behavior with extremely minimal effort! Check that you don't have any duplicated data, missing values, consistent dataframe shapes, column data types, etc.

If you take nothing else away from this talk, start adding assertions within your code.

Simple test example

In [4]:
```
def backwards_allcaps(text):
    return text[::-1].upper()
```

In [5]:
```
backwards_allcaps('Python')
```

Out[5]:
```
'NOHTYP'
```

In [6]:
```
def test_backwards_allcaps():
    assert backwards_allcaps('pycon') == 'NOCYP'
    assert backwards_allcaps('Cleveland') == 'DNALEVELC'
```

pytest

- less boilerplate → easier/faster test writing
- automatically handles finding, collecting, running, evaluating your tests
- when tests fail you can get a lot of useful info
- lots of powerful built in features
- just works (with benefits) on existing tests written for unittest or nose

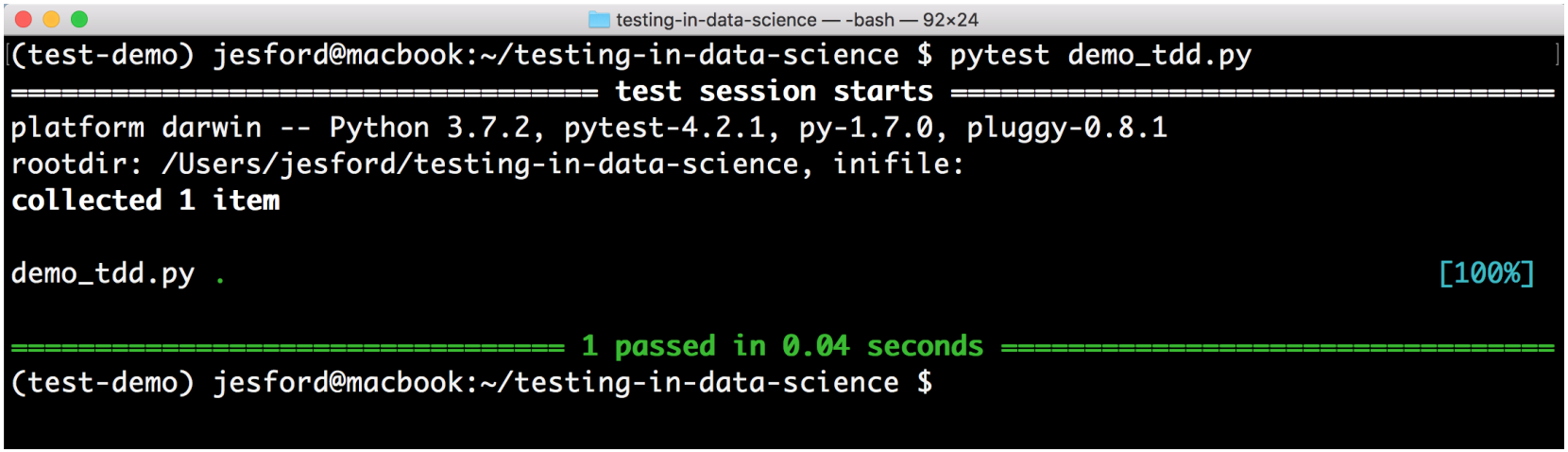
```
$ pip install pytest
```

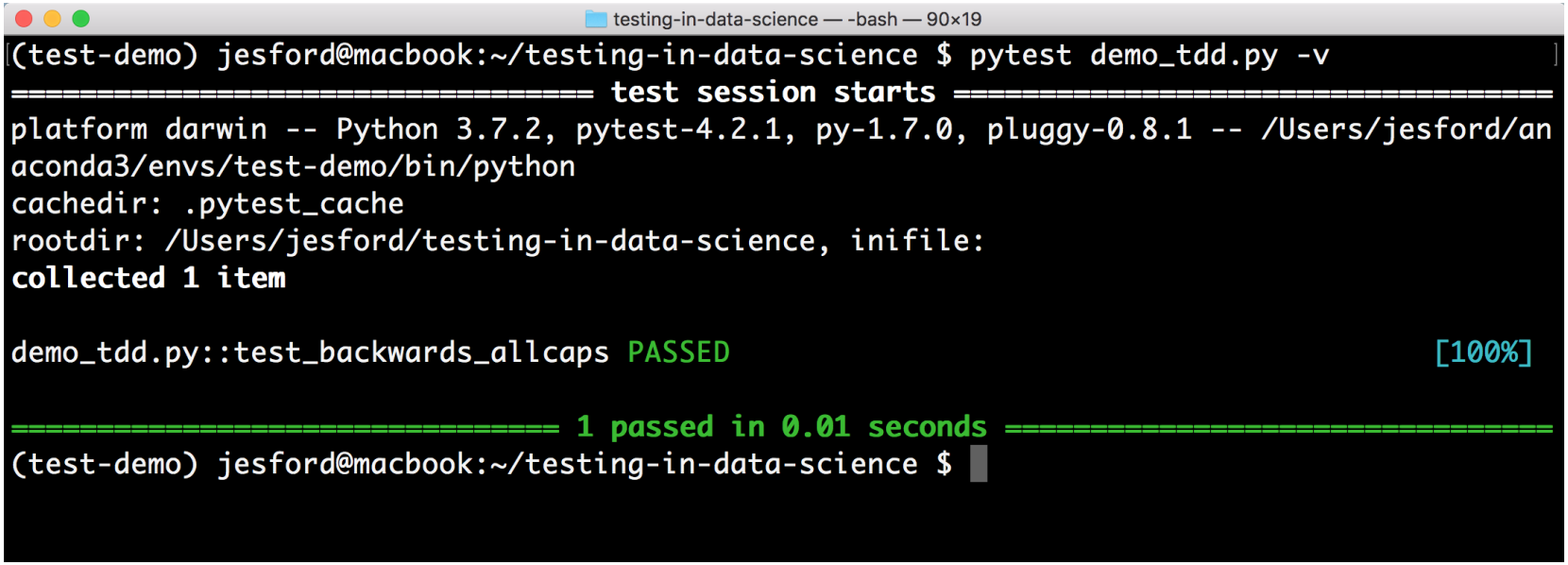
pytest demo

In [7]:
```python
# contents of demo_tdd.py

def backwards_allcaps(text):
    return text[::-1].upper()


def test_backwards_allcaps():
    assert backwards_allcaps('pycon') == 'NOCYP'
    assert backwards_allcaps('Cleveland') == 'DNALEVELC'
```

How to run tests?

```
$ pytest demo_tdd.py
```


testing-in-data-science — -bash — 92×24
(test-demo) jesford@macbook:~/testing-in-data-science $ pytest demo_tdd.py
test session starts
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
rootdir: /Users/jesford/testing-in-data-science, inifile:
collected 1 item
demo_tdd.py .
[100%]
1 passed in 0.04 seconds
(test-demo) jesford@macbook:~/testing-in-data-science $


testing-in-data-science — -bash — 90×19
(test-demo) jesford@macbook:~/testing-in-data-science $ pytest demo_tdd.py -v
test session starts
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1 -- /Users/jesford/an
aconda3/envs/test-demo/bin/python
cachedir: .pytest_cache
rootdir: /Users/jesford/testing-in-data-science, inifile:
collected 1 item
demo_tdd.py::test_backwards_allcaps PASSED
[100%]
1 passed in 0.01 seconds
(test-demo) jesford@macbook:~/testing-in-data-science $

New feature: whitespace should be removed from input text

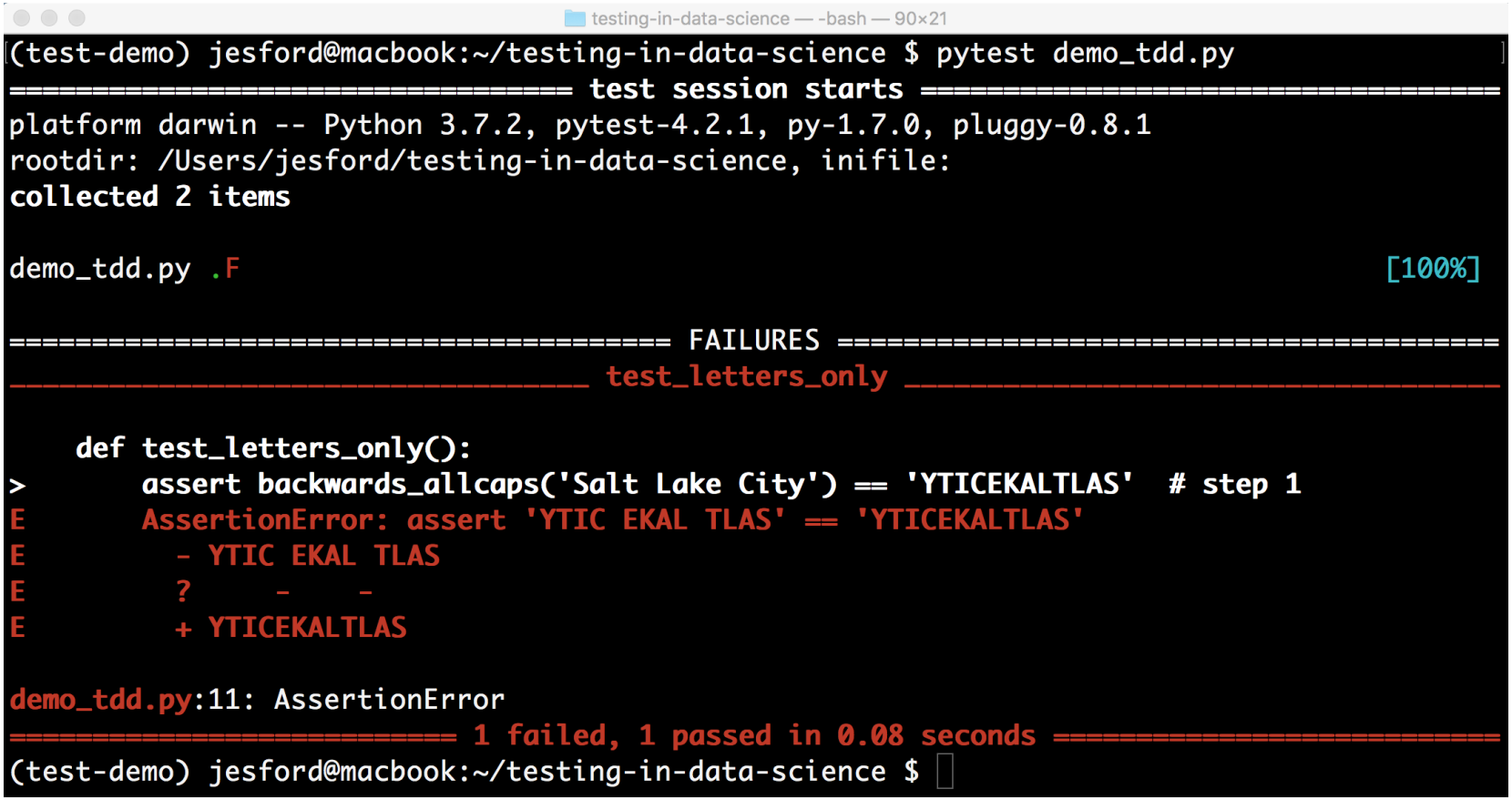
In [8]:
```
def backwards_allcaps(text):
    return text[::-1].upper()


def test_backwards_allcaps():
    assert backwards_allcaps('pycon') == 'NOCYP'
    assert backwards_allcaps('Cleveland') == 'DNALEVELC'
```

TDD:

1. add a test
2. run the test (it should fail)
3. add the feature
4. run the test

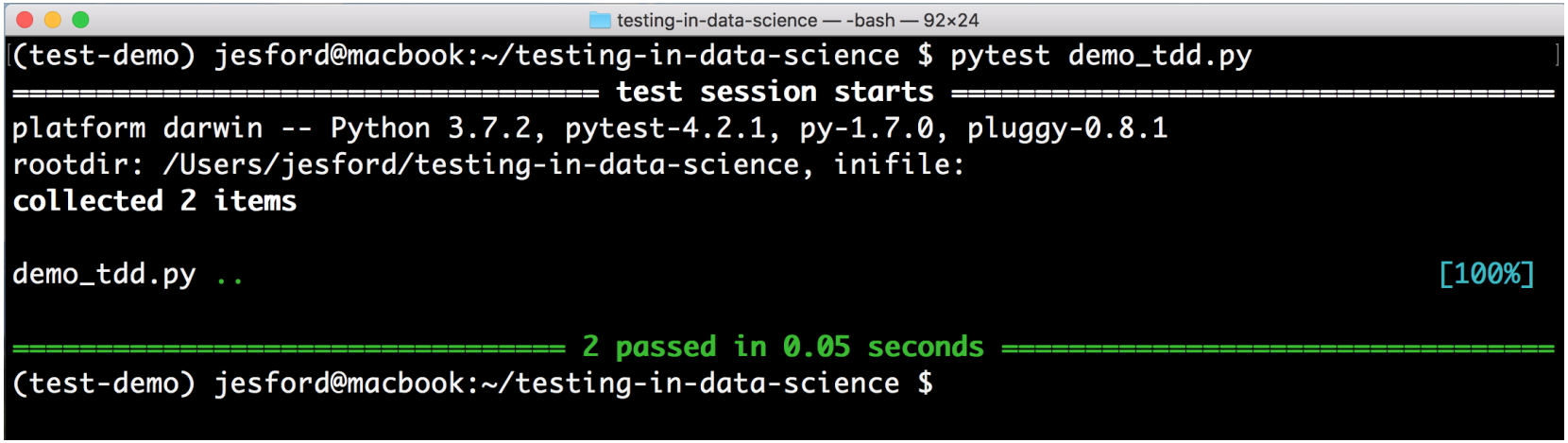
New feature: whitespace should be removed from input text

In [9]:
```
def backwards_allcaps(text):
    return text[::-1].upper()


def test_backwards_allcaps():
    assert backwards_allcaps('pycon') == 'NOCYP'
    assert backwards_allcaps('Cleveland') == 'DNALEVELC'


def test_letters_only():
    assert backwards_allcaps('Salt Lake City') == 'YTICEKALTLAS'  # step 1
```


testing-in-data-science — -bash — 90×21
(test-demo) jesford@macbook:~/testing-in-data-science $ pytest demo_tdd.py
test session starts
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
rootdir: /Users/jesford/testing-in-data-science, inifile:
collected 2 items
demo_tdd.py .F [100%]
FAILURES
test_letters_only
def test_letters_only():
> assert backwards_allcaps('Salt Lake City') == 'YTICEKALTLAS' # step 1
E AssertionError: assert 'YTIC EKAL TLAS' == 'YTICEKALTLAS'
E - YTIC EKAL TLAS
E ? - -
E + YTICEKALTLAS
demo_tdd.py:11: AssertionError
1 failed, 1 passed in 0.08 seconds
(test-demo) jesford@macbook:~/testing-in-data-science $

New feature: whitespace should be removed from input text

In [10]:
```
def backwards_allcaps(text):
    return text[::-1].replace(' ', '').upper()            # step 2


def test_backwards_allcaps():
    assert backwards_allcaps('pycon') == 'NOCYP'
    assert backwards_allcaps('Cleveland') == 'DNALEVELC'


def test_letters_only():
    assert backwards_allcaps('Salt Lake City') == 'YTICEKALTLAS'  # step 1
```


testing-in-data-science — -bash — 92×24
(test-demo) jesford@macbook:~/testing-in-data-science $ pytest demo_tdd.py
test session starts
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
rootdir: /Users/jesford/testing-in-data-science, inifile:
collected 2 items
demo_tdd.py ..
[100%]
2 passed in 0.05 seconds
(test-demo) jesford@macbook:~/testing-in-data-science $

That's great, but these examples were dumb

1. these test examples don't really apply to data science work
2. this TDD workflow isn't always reasonable during research & exploration

Data Science Domain Problems

- dataframes are the input and output of your functions
- working with databases
- ML models with non-deterministic outcomes
- acceptable tolerances on results
- testing for properties of things rather than exact values

Data Science Workflows

1. "One-off analysis"
2. Exploratory
3. Well defined problem

Data Science Workflows

1. **"One-off analysis"** ←
2. Exploratory
3. Well defined problem

For one-off analyses I do not write tests, but instead focus on clear documentation in case the analysis gets revisited.

If it *does* get revisited, I'll consider breaking the code out of a notebook and into a module (possibly refactoring) and adding some tests.

Data Science Workflows

1. "One-off analysis"
2. **Exploratory** ←
3. Well defined problem

Its impractical to write tests during the exploratory phase. However, if things go well there is almost always code created along the way which is useful in a later stage of the project.

Judgment call needed as my legacy/untested code base grows...

Data Science Workflows

1. "One-off analysis"
2. Exploratory
3. **Well defined problem** ←

If I'm writing code for a fairly well defined problem, which I know will be re-used, I try very hard to write tests as I develop the code.

Data Science Workflows

1. "One-off analysis"
2. **Exploratory** ←
3. Well defined problem
4. **Legacy code** ←

Once I realize I will need to reuse code, I try to start adding tests *when I modify it.*

Generally, if I'm confident something is working now, I'll only bother to add tests when I'm adding features or fixing bugs. (Inspired by Justin Crown's PyCon 2018 talk (https://www.youtube.com/watch?v=LDdUuoI_IIg)).

Data Science Domain Problems

Examples of tests for common data science problems

Working with Pandas DataFrames

Checking for duplicates and missing values.

In [11]:
```
import pandas as pd
import numpy as np


df = pd.DataFrame({'channel': ['email', 'paid_search', 'display', 'email'],
                   'customer': [1, 4, 4, 3],
                   'order': [1010, 2050, 2050, 3232]})
df
```

Out[11]:

	channel	customer	order
0	email	1	1010
1	paid_search	4	2050
2	display	4	2050
3	email	3	3232

In [12]:
```
assert df.notnull().all().all()
assert ~df.isnull().any().any()
assert df.isnull().sum().sum() == 0
```

Working with Pandas DataFrames

Checking for duplicates and missing values.

In [13]:
```
df
```

Out[13]:

	channel	customer	order
0	email	1	1010
1	paid_search	4	2050
2	display	4	2050
3	email	3	3232

In [14]:
```
assert ~df.duplicated().any()
```

In [15]:
```
if df.duplicated(subset=['order']).any():
    raise ValueError('Duplicate records found for order')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-15-d574c85f7f29> in <module>
      1 if df.duplicated(subset=['order']).any():
----> 2     raise ValueError('Duplicate records found for order')

ValueError: Duplicate records found for order
```

Working with Pandas DataFrames

Built in utilities that help you test.

In [16]:
```
from pandas.util.testing import assert_frame_equal
from pandas.util.testing import assert_index_equal
from pandas.util.testing import assert_series_equal
```

In [18]:
```
assert_frame_equal(df, df2,
                   check_like=True,       # order of columns/rows doesn't matter
                   check_dtype=False,     # check for identical data types
                   check_less_precise=4)  # number of digits to compare
```

Also handles NaN or None comparisons "as expected".

Working with Databases

Testing a function that queries the DB

In [ ]:
```python
# my_data_loader.py
import pandas as pd
import query_database


def load_data(condition=''):
    sql_query = f'select id, type, val from some_table {condition}'
    df_raw = query_database(sql_query)
    df = pd.get_dummies(df_raw, columns=['type'])
    df.index = df.pop('id')
    return df
```

In [ ]:
```python
# test_data_loader.py
import pytest
import my_data_loader
from pandas.util.testing import assert_frame_equal


@pytest.fixture(params=[{'condition': 'where val > 100', 'output': out1}])
def sample_data(request):
    return request.param


def test_load_data(sample_data):
    # problem: we might not want to query the DB as part of our tests
    output = my_data_loader.load_data(sample_data['condition'])
    assert_frame_equal(output, sample_data['output'])
```

mocker

pytest-mock is a plugin that lets you patch or swap out one piece of code for another

Testing a function that queries the DB

In [ ]:
```python
# my_data_loader.py
import pandas as pd
import query_database


def load_data(condition=''):
    sql_query = f'select id, type, val from some_table {condition}'
    df_raw = query_database(sql_query)
    df = pd.get_dummies(df_raw, columns=['type'])
    df.index = df.pop('id')
    return df
```

In [ ]:
```python
# test_data_loader.py
import pytest
import my_data_loader
from pandas.util.testing import assert_frame_equal


@pytest.fixture(params=[{'input': in1, 'output': out1}])
def sample_data(request):
    return request.param


def test_load_data(sample_data, mocker):
    mocker.patch('my_data_loader.query_database',
                 side_effect=lambda x: sample_data['input'])
    output = my_data_loader.load_data('')
    assert_frame_equal(output, sample_data['output'])
```

Generating DataFrames for testing

Because hardcoding input/output dataframes is *extremely* verbose

Hypothesis

Automatic data generation for property based testing

In [25]:
```
from hypothesis import strategies as st


print('Examples of integers:')
print(st.integers().example())
print(st.integers().example())
print(st.integers().example())
```

```
Examples of integers:
12
18697
-127
```

In [20]:

```python
# contents of demo_hypothesis.py
from hypothesis import given
from hypothesis import strategies as st


def backwards_allcaps(text):
    return text[::-1].upper()


@given(st.text())
def test_backwards_allcaps(input_string):
    modified = backwards_allcaps(input_string)
    assert input_string.upper() == ''.join(reversed(modified))
```

```
testing-in-data-science — -bash — 115×35
(test-demo) jesford@macbook:~/testing-in-data-science $ pytest demo_hypothesis.py --hypothesis-show-statistics
============================= test session starts ==============================
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/Users/jesford/testing-in-data-science/.hypothesis/examples')
rootdir: /Users/jesford/testing-in-data-science, inifile:
plugins: hypothesis-4.6.1
collected 1 item

demo_hypothesis.py .                                                     [100%]

=========================== Hypothesis Statistics ============================

demo_hypothesis.py::test_backwards_allcaps:

  - 100 passing examples, 0 failing examples, 0 invalid examples
  - Typical runtimes: 0-2 ms
  - Fraction of time spent in data generation: ~ 75%
  - Stopped because settings.max_examples=100

========================== 1 passed in 1.05 seconds ==========================
(test-demo) jesford@macbook:~/testing-in-data-science $
```

Hypothesis + Pandas

In [33]:
```
from hypothesis.extra.pandas import data_frames, column



data_frames([column('customer',
                    elements=st.integers(min_value=0, max_value=100_000),
                    dtype=int, unique=True),
             column('price', dtype='float'),
             column('prob_return',
                    elements=st.floats(min_value=0, max_value=1))
         ]).example()
```

Out[33]:

	customer	price	prob_return
0	80119	2.180319e+16	0.22176
1	99019	2.180319e+16	0.22176

Hypothesis + Pandas

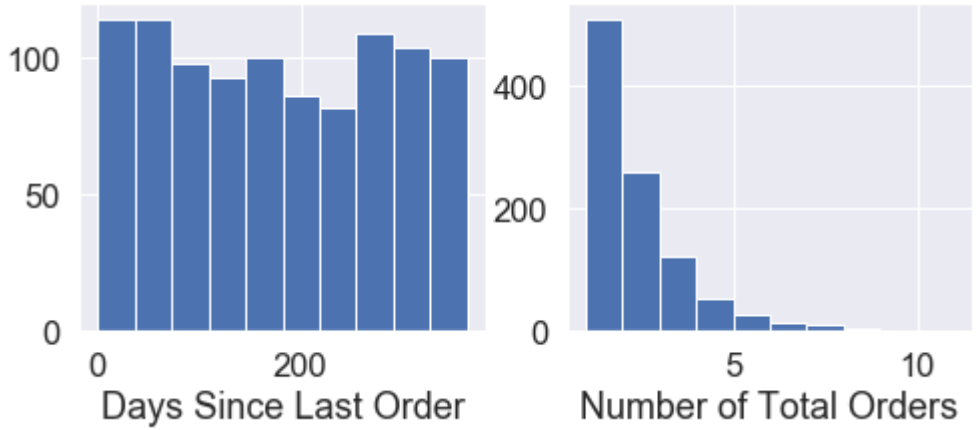
In [34]:
```
from hypothesis.extra.pandas import data_frames, column



data_frames([column('customer',
                    elements=st.integers(min_value=0, max_value=100_000),
                    dtype=int, unique=True),
             column('price', dtype='float'),
             column('prob_return',
                    elements=st.floats(min_value=0, max_value=1))
            ]).example()
```

Out[34]:

	customer	price	prob_return

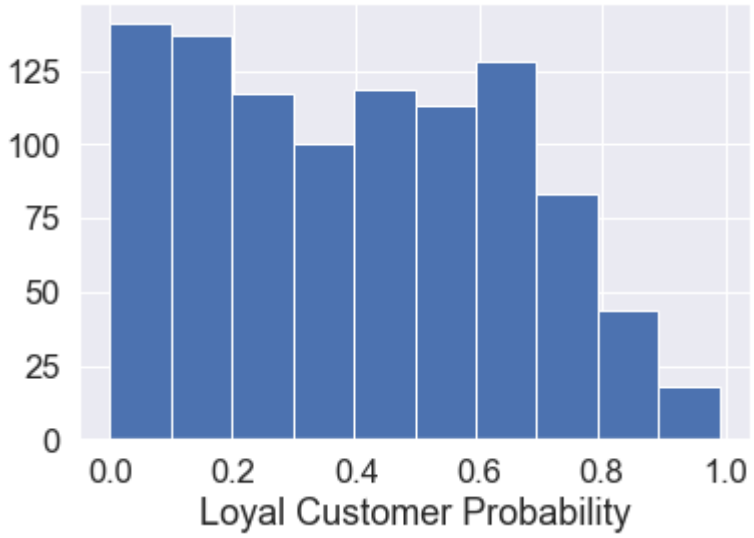
Testing properties of data

In [37]:
```
import matplotlib.pyplot as plt
import seaborn as sns


sns.set(font_scale=1.5)

df_customers = pd.DataFrame(
    {'days_since_last_order': np.random.randint(low=0, high=365, size=1000),
     'num_total_orders': np.random.geometric(0.5, size=1000)})

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
df_customers.days_since_last_order.hist(ax=ax1)
df_customers.num_total_orders.hist(ax=ax2)
ax1.set_xlabel('Days Since Last Order')
ax2.set_xlabel('Number of Total Orders')
plt.show();
```



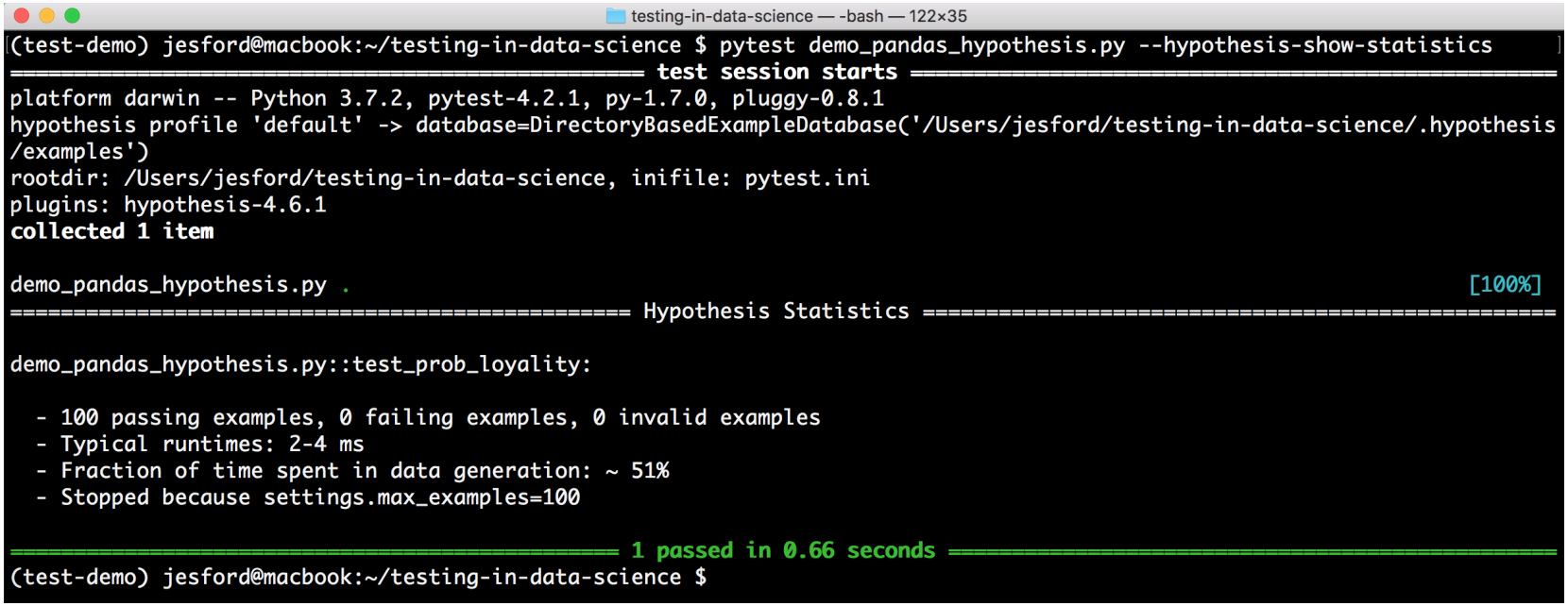
Testing properties of data

In [38]:
```
from scipy.special import expit  # logistic function


def probality_loyal_customer(df):
    "Return customer probability of returning."
    p_num_orders = df.num_total_orders.apply(expit)
    p_days_ago = df.days_since_last_order / df.days_since_last_order.max()
    p_loyal = p_days_ago * p_num_orders
    return p_loyal


prob_loyal = probality_loyal_customer(df_customers)
prob_loyal.hist()
plt.xlabel('Loyal Customer Probability');
```



In [39]:

```python
# contents of demo_pandas_hypothesis.py
from hypothesis import given
from hypothesis import strategies as st
from hypothesis.extra.pandas import data_frames, column
from scipy.special import expit


def probability_loyal_customer(df):
    "Return customer probability of returning."
    p_num_orders = df.num_total_orders.apply(expit)
    p_days_ago = df.days_since_last_order / df.days_since_last_order.max()
    p_loyal = p_days_ago * p_num_orders
    return p_loyal


@given(
    data_frames([
        column('days_since_last_order', dtype=int,
               elements=st.integers(min_value=0, max_value=365)),
        column('num_total_orders', dtype=int,
               elements=st.integers(min_value=0, max_value=1_000_000))])
)
def test_prob_loyality(df):
    p = probability_loyal_customer(df)
    assert p.between(0, 1, inclusive=True).all()
```


testing-in-data-science — -bash — 122×35
(test-demo) jesford@macbook:~/testing-in-data-science $ pytest demo_pandas_hypothesis.py --hypothesis-show-statistics
test session starts
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/Users/jesford/testing-in-data-science/.hypothesis/examples')
rootdir: /Users/jesford/testing-in-data-science, inifile: pytest.ini
plugins: hypothesis-4.6.1
collected 1 item
demo_pandas_hypothesis.py .
[100%]
Hypothesis Statistics
demo_pandas_hypothesis.py::test_prob_loyality:
- 100 passing examples, 0 failing examples, 0 invalid examples
- Typical runtimes: 2-4 ms
- Fraction of time spent in data generation: ~ 51%
- Stopped because settings.max_examples=100
1 passed in 0.66 seconds
(test-demo) jesford@macbook:~/testing-in-data-science $

Wrap up

- data scientists should not *always* write tests
    - (but we should always practice defensive programming)
- any reused or shared piece of code should probably be tested, especially in production
- strive for a balance between speed and confidence in your results
    - testing can help you acheive this!

Some aspects of data science code are really hard to test!

- ML results? probabilistic outcomes?
- Think about testing properties of your data
    - distributions, missing data, expected features and datatypes

Resources & Credits

- **General testing resources**
    - Andreas Pelme's Introduction to pytest (https://www.youtube.com/watch?v=LdVJj65ikRY) from EuroPython 2014
    - Mark Vousden's Python testing (https://www.youtube.com/channel/UCKaKhMyhboLoMwmeF9yxg9w) 3-part series of youtube videos
    - Justin Crown's "WHAT IS THIS MESS?" - Writing tests for pre-existing code bases (https://www.youtube.com/watch?v=LDdUuoI_IIg) from PyCon 2018
    - Ned Batchelder's Getting Started Testing (https://www.youtube.com/watch?v=FxSsnHeWQBY) from PyCon 2014 (focuses on unittest)
- **Data Science specific resources**
    - Trey Causey's Testing for Data Scientists (https://www.youtube.com/watch?v=GEqM9uJi64Q) from PyData Seattle 2015
    - Eric Ma's Best Testing Practice's for Data Science Tutorial (https://www.youtube.com/watch?v=yACtdj1_IxE) from PyCon 2017, with GitHub notebooks here (https://github.com/ericmjl/data-testing-tutorial)