On the importance of tests

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

Software tends to change

Common reasons for change:

- new functionality desired
- bugs need to get squashed
- performance needs to be improved

Core functionality rarely evolves

How to ensure stability in light of changes?

Rationale behind testing

- **improves chance of code being correct in the *future***
  - prevent introducing breaking changes

- **raises *confidence* (not a guarantee) that code is correct *now***
  - assert actuals match expectations

- **most up-to-date documentation**
  - form of documentation that is always in sync with what's running

The test pyramid: where to invest your efforts

Testing takes time

- thinking what to test
- writing tests
- running tests

Testing has a high return on investment

- when targeted at the correct layer
- when testing the non-trivial parts, e.g. distance between 2 coordinates ? uppercasing a first name

© Martin Fowler “TestPyramid”

The test pyramid: where to invest your efforts

Testing takes time

- thinking what to test
- writing tests
- running tests

Testing has a high return on investment

- when targeted at the correct layer
- when testing the non-trivial parts, e.g. distance between 2 coordinates ? uppercasing a first name

© Martin Fowler “TestPyramid”

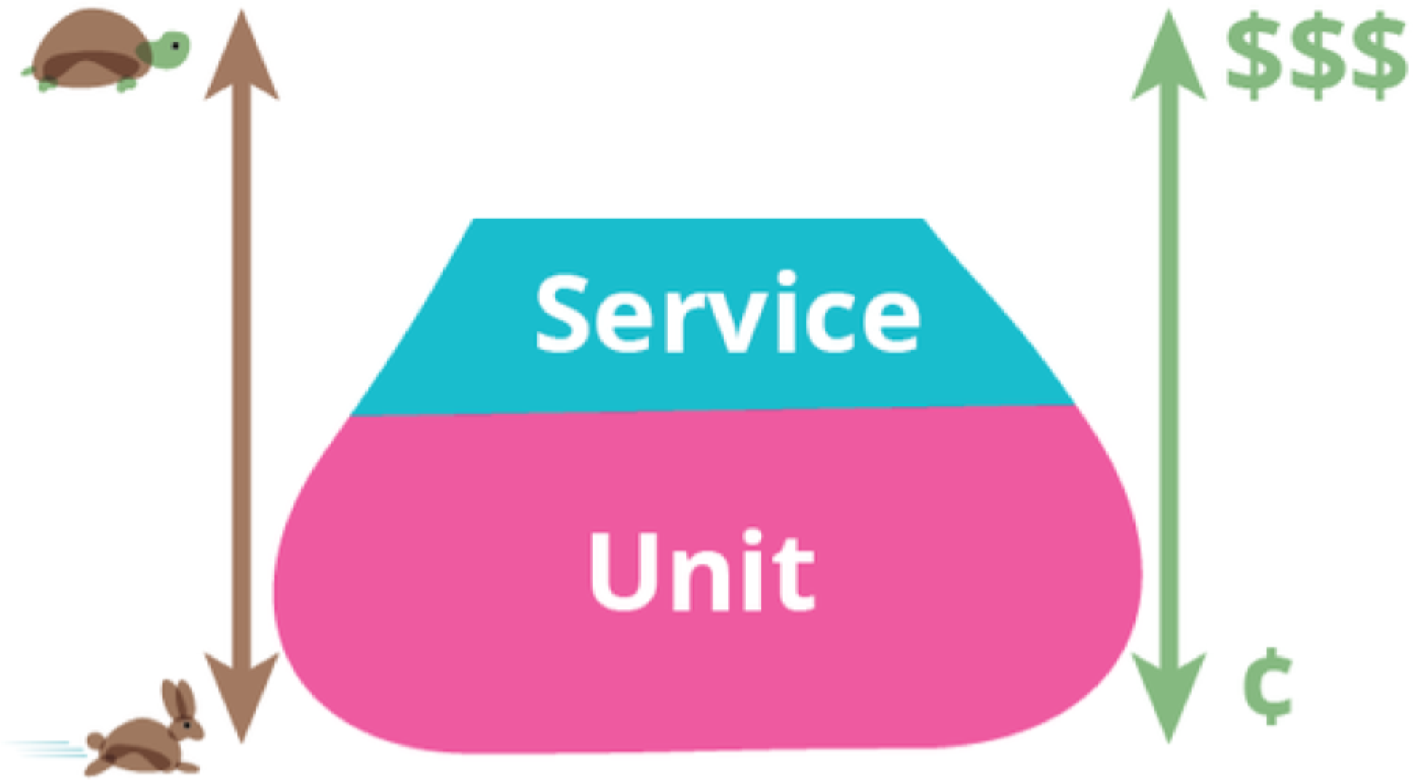
The test pyramid: where to invest your efforts

Testing takes time

- thinking what to test
- writing tests
- running tests

Testing has a high return on investment

- when targeted at the correct layer
- when testing the non-trivial parts, e.g.
  distance between 2 coordinates ?
  uppercasing a first name



© Martin Fowler “TestPyramid”

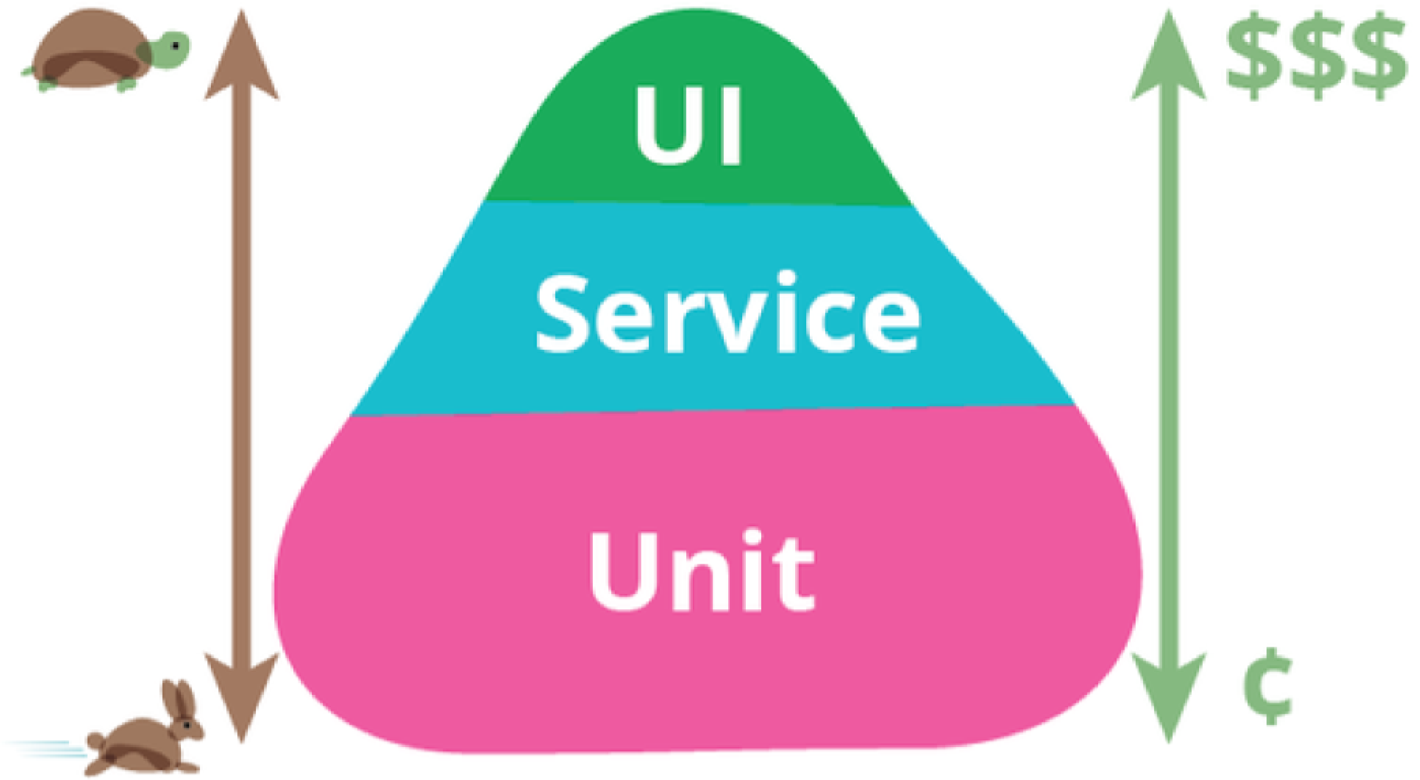
The test pyramid: where to invest your efforts

Testing takes time

- thinking what to test
- writing tests
- running tests

Testing has a high return on investment

- when targeted at the correct layer
- when testing the non-trivial parts, e.g. distance between 2 coordinates ? uppercasing a first name



© Martin Fowler "TestPyramid"

Let's have this sink in!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

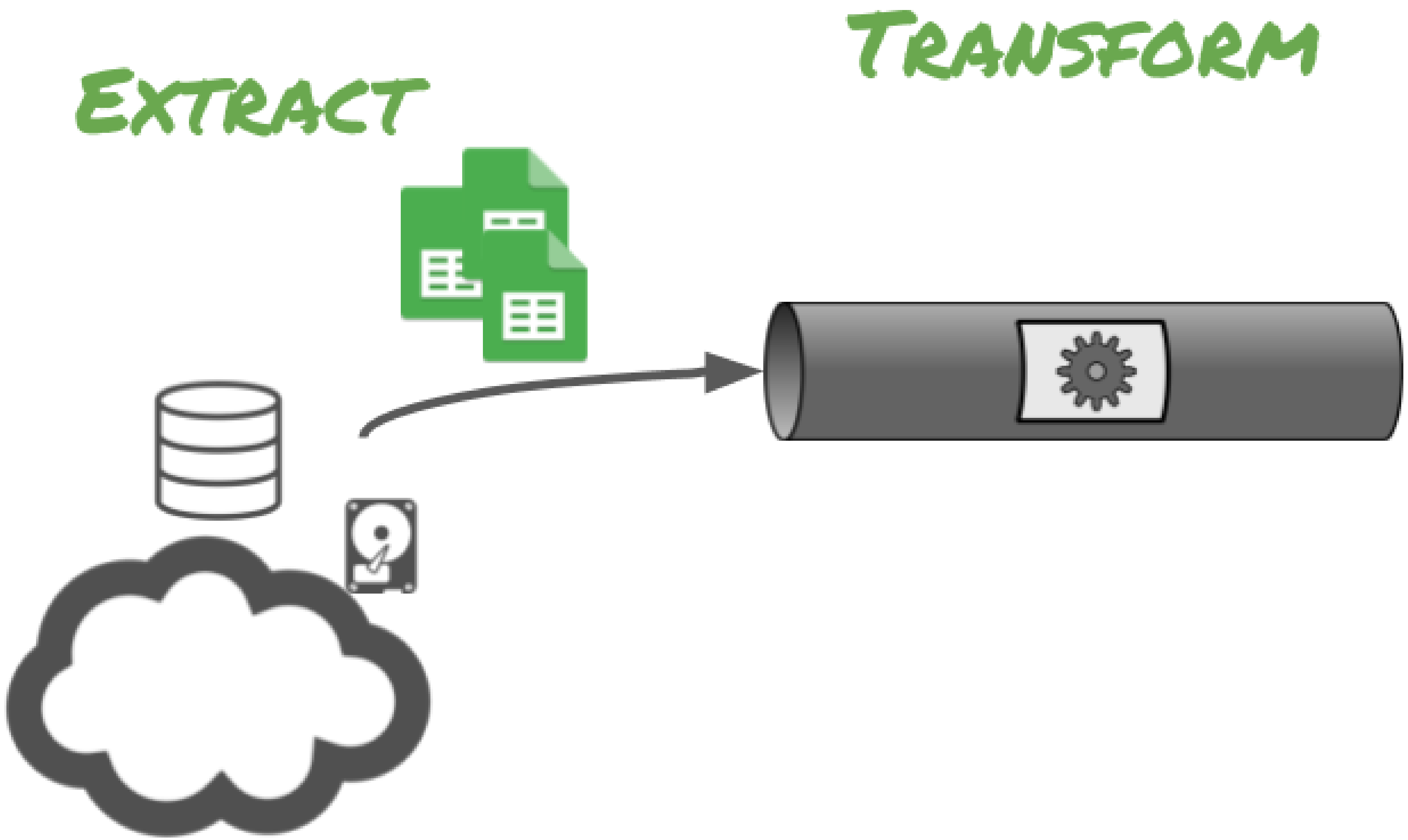
Writing unit tests for PySpark

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

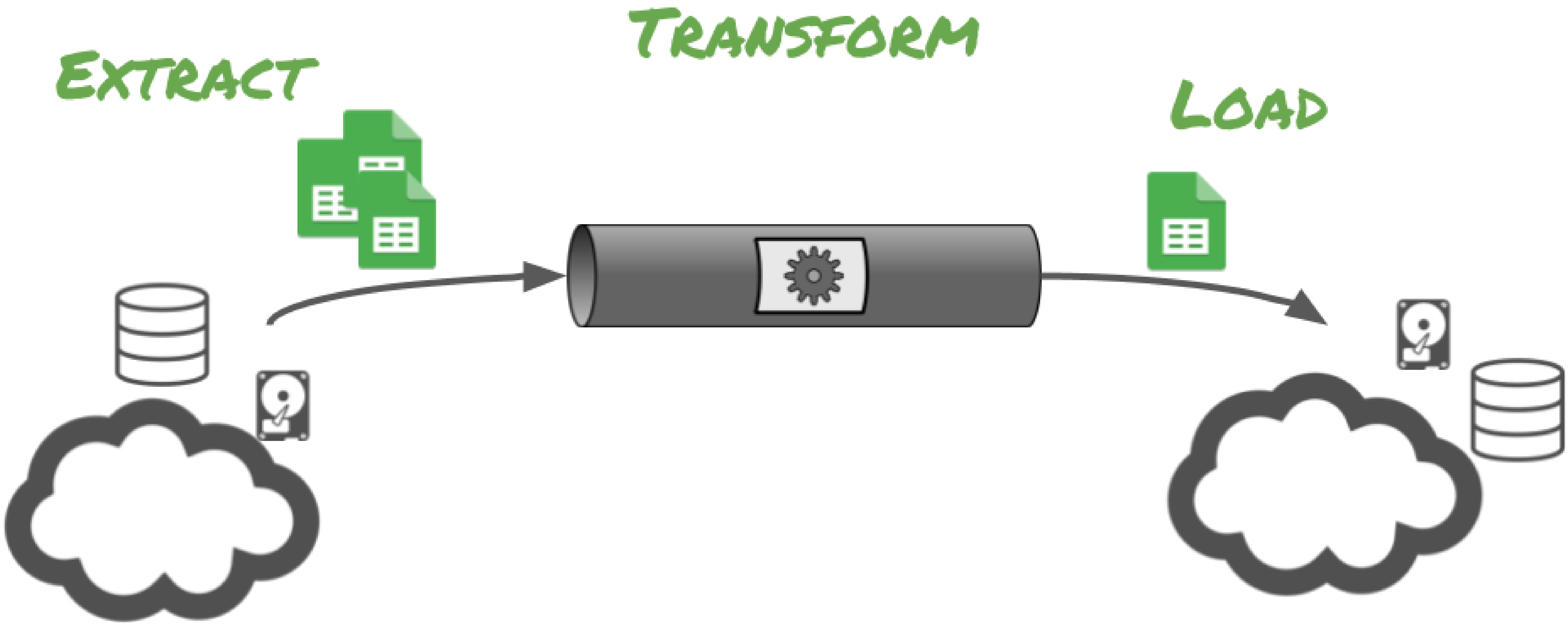
Oliver Willekens
Data Engineer at Data Minded

Our earlier Spark application is an ETL pipeline

Our earlier Spark application is an ETL pipeline



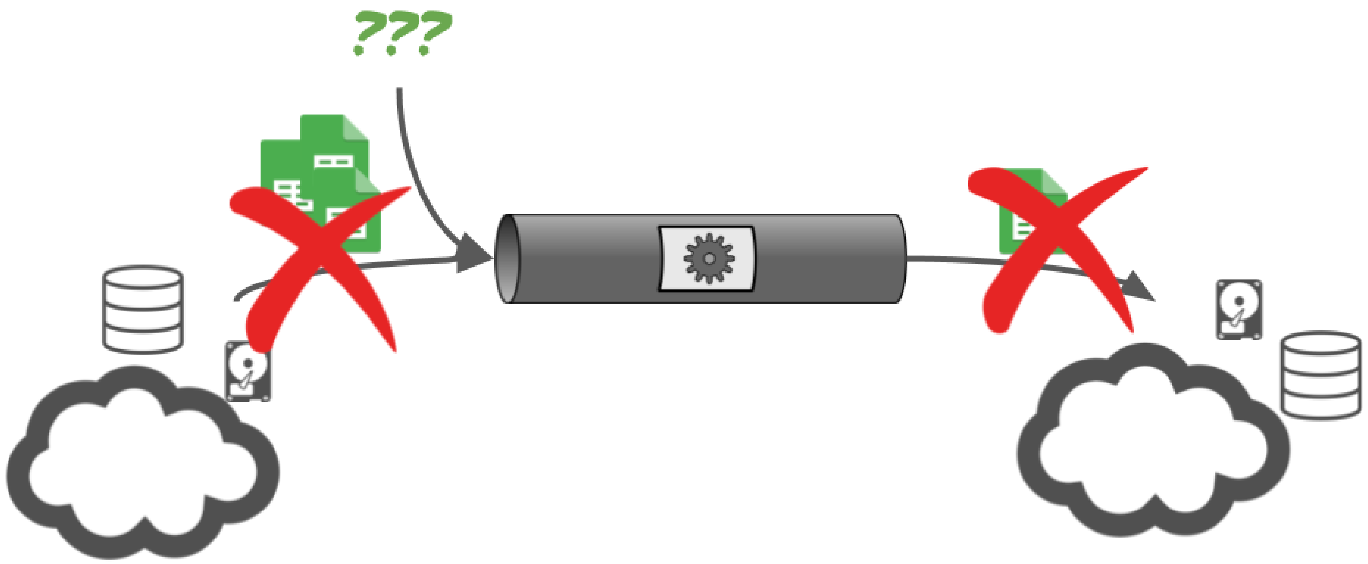
Our earlier Spark application is an ETL pipeline



Separate transform from extract and load

```
prices_with_ratings = spark.read.csv(…)  # extract
exchange_rates = spark.read.csv(…)  # extract

unit_prices_with_ratings = (prices_with_ratings
                            .join(…) # transform
                            .withColumn(…))  # transform
```



Solution: construct DataFrames in-memory

```
# Extract the data
df = spark.read.csv(path_to_file)
```

- depends on input/output (network access, filesystem permissions, ...)
- unclear how big the data is
- unclear what data goes in

```
from pyspark.sql import Row
purchase = Row("price",
               "quantity",
               "product")
record = purchase(12.99, 1, "cake")
df = spark.createDataFrame((record,))
```

+ inputs are clear
+ data is close to where it is being used ("code-proximity")

Create small, reusable and well-named functions

```
unit_prices_with_ratings = (prices_with_ratings
                            .join(exchange_rates, ["currency", "date"])
                            .withColumn("unit_price_in_euro",
                                        col("price") / col("quantity")
                                        * col("exchange_rate_to_euro"))
```

```
def link_with_exchange_rates(prices, rates):
    return prices.join(rates, ["currency", "date"])

def calculate_unit_price_in_euro(df):
    return df.withColumn(
        "unit_price_in_euro",
        col("price") / col("quantity") * col("exchange_rate_to_euro"))
```

Create small, reusable and well-named functions

```python
def link_with_exchange_rates(prices, rates):
    return prices.join(rates, ["currency", "date"])

def calculate_unit_price_in_euro(df):
    return df.withColumn(
        "unit_price_in_euro",
        col("price") / col("quantity") * col("exchange_rate_to_euro"))
```

```python
unit_prices_with_ratings = (
  calculate_unit_price_in_euro(
    link_with_exchange_rates(prices, exchange_rates)
  )
)
```

Testing a single unit

```
def test_calculate_unit_price_in_euro():
    record = dict(price=10,
                  quantity=5,
                  exchange_rate_to_euro=2.)
    df = spark.createDataFrame([Row(**record)])
```

Testing a single unit

```
def test_calculate_unit_price_in_euro():
    record = dict(price=10,
                  quantity=5,
                  exchange_rate_to_euro=2.)
    df = spark.createDataFrame([Row(**record)])
    result = calculate_unit_price_in_euro(df)
```

Testing a single unit

```python
def test_calculate_unit_price_in_euro():
    record = dict(price=10,
                  quantity=5,
                  exchange_rate_to_euro=2.)
    df = spark.createDataFrame([Row(**record)])
    result = calculate_unit_price_in_euro(df)

    expected_record = Row(**record, unit_price_in_euro=4.)
    expected = spark.createDataFrame([expected_record])
```

Testing a single unit

```
def test_calculate_unit_price_in_euro():
    record = dict(price=10,
                  quantity=5,
                  exchange_rate_to_euro=2.)
    df = spark.createDataFrame([Row(**record)])
    result = calculate_unit_price_in_euro(df)

    expected_record = Row(**record, unit_price_in_euro=4.)
    expected = spark.createDataFrame([expected_record])

    assertDataFrameEqual(result, expected)
```

Take home messages

1. Interacting with external data sources is costly
2. Creating in-memory DataFrames makes testing easier
   - the data is in plain sight,
   - focus is on just a small number of examples.
3. Creating small and well-named functions leads to more reusability and easier testing.

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Continuous testing

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

Running a test suite

Execute tests in Python, with one of:

in stdlib	3rd party
unittest	pytest
doctest	nose

Core task: **assert** or raise

Examples:

```
assert computed == expected
with pytest.raises(ValueError):  # pytest specific
```

Manually triggering tests

In a Unix shell:

```
cd ~/workspace/my_good_python_project
pytest .
# Lots of output…
== 19 passed, 2 warnings in 36.80 seconds ==
```

```
cd ~/workspace/my_bad_python_project
pytest .
# Lots of output…
== 3 failed, 1 passed in 6.72 seconds ==
```

Note: Spark increases time to run unit tests.

Automating tests

Problem:

- forget to run unit tests when making changes

Solution:

- Automation

How:

- Git -> configure hooks
- Configure CI/CD pipeline to run tests automatically

CI/CD

Continuous Integration:

- get code changes integrated with the master branch regularly.

Continuous Delivery:

- Create “artifacts” (deliverables like documentation, but also programs) that can be deployed into production without breaking things.

Configuring a CI/CD tool

CircleCI looks for *.circleci/config.yml*.

Example:

```
jobs:
  test:
    docker:
      - image: circleci/python:3.6.4
    steps:
      - checkout
      - run: pip install -r requirements.txt
      - run: pytest .
```

Often:

1. checkout code
2. install test & build requirements
3. run tests
4. package/build the software artefacts
5. deploy the artefacts (update docs / install app / ...)

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON