Python
Best Practices for
Data Science
in Production

INGKA

IKEA

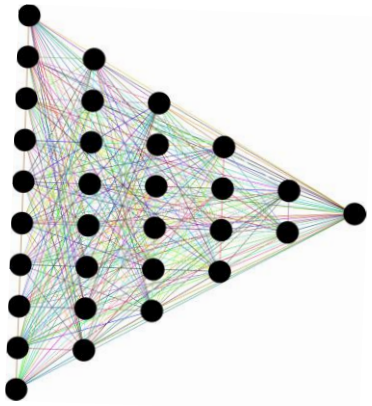
Fulfilment Development & Multichannel Network

IKEA Retail business

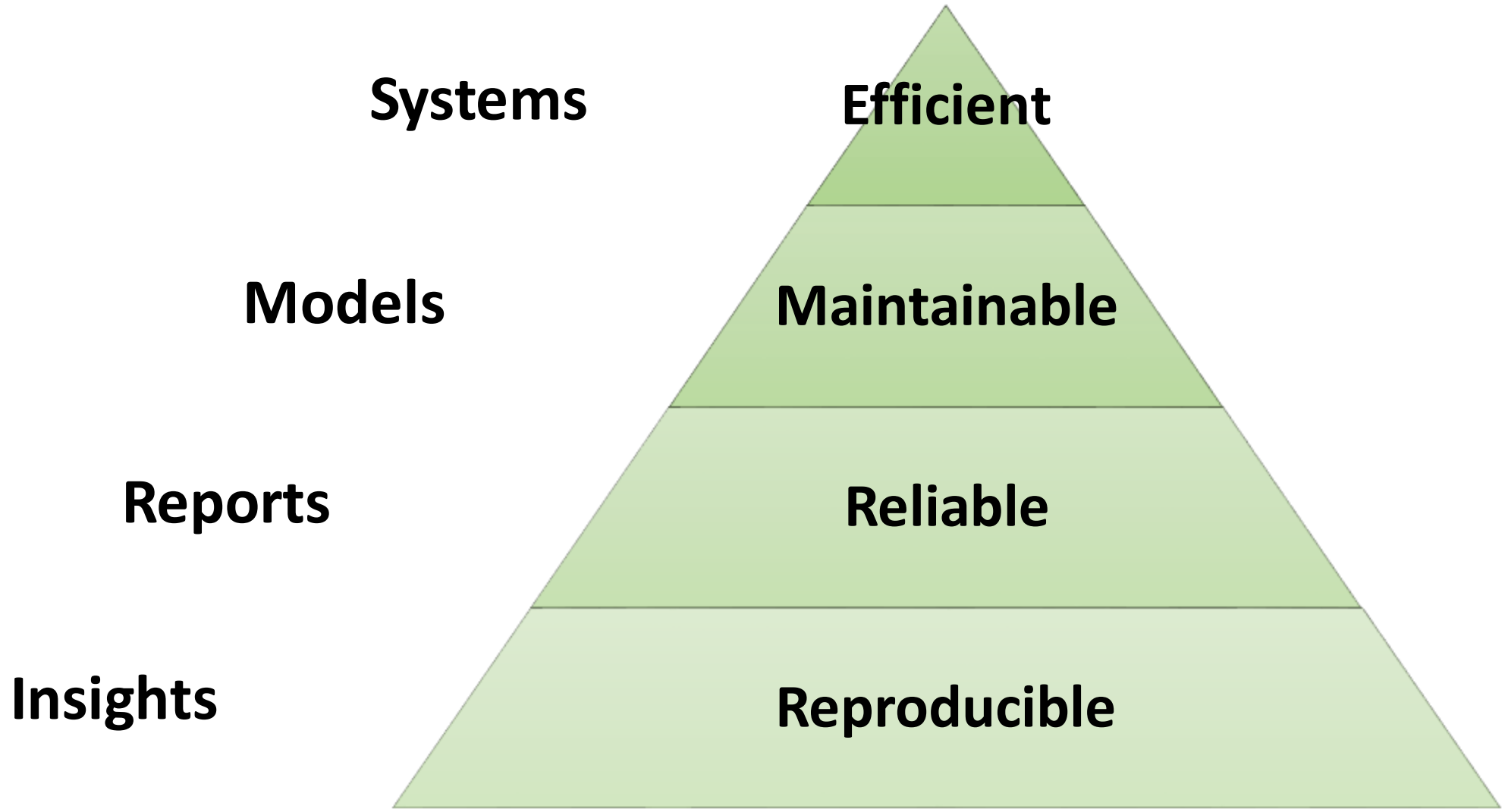
What is a product of Data Science?

- Insights?
- Reports?
- Models?
- Software systems?

All of it!



Importance of quality attributes



The **importance of best practices** is proportional to the:

- **Number of people affected**
- **Number of collaborators**
- **Running time of experiments**

How to produce reproducible code

- Use isolated & reproducible environments
- Control randomness
- Keep track of data sources and configuration

Virtual environments

- Virtualenv (environment manager) with PIP (package manager)
- Conda (environment and package manager)
    - Advantage #1: Integrated handling of Python versions
    - Advantage #2: Can track non-Python dependencies
    - Advantage #3: Conda is compatible with pip. Virtualenv is NOT compatible with conda.
- **Ana**conda (conda with a bunch of pre-installed packages)
    - WARNING: Makes the environment more clunky

Dependency versions

- Specify exact versions in requirements.txt or environment.yaml
    - Example: numpy==1.17.4
- Don’t do this: pip freeze > requirements.txt (results in Anaconda-like environment)
- Python is also a package!
    - Make sure to use the production version in your dev environment

Control randomness

- Set seed for ALL non-deterministic libs/frameworks. E.g.:
    - PYTHONHASHSEED
    - random.seed
    - np.random.seed
    - gurobipy.Params.Seed
    - tf.set_random_seed
    - tf.Dropout(0.2, seed=seed_value)
- When sharing code, add a seed parameter to non-deterministic public functions
- Remember to seed unit tests

Track experiments

- Log all data sources
- Can you trust the data source to never change? If not, keep copies.
- Log all configuration, e.g. user input, hyperparameters, etc.
    - sacred is a great library for experiment tracking

How to produce reliable code

- Detect errors early
- Write tests
- Automate testing
- Despite testing... there will be bugs!

Prevent errors

- Type hints
    - Introduced in Python 3.5
    - Helps the IDE detect errors in a dynamically typed language
- Use mypy together with type hints to enforce static type checking

Type hints help IDEs detect errors

```python
def line_cost(line: OrderLine, costs: Dict):
    """
    Return the costs of an order line, given a dict of different cost parts.

    :param line: the order line to compute costs for
    :param costs: a dict containing relevant costs
    :return: a tuple containing the fixed cost per line, the variable cost for this line and the total cost
    """

    fixed_cost_per_line = (
        costs["Picking cost (order line)"] + costs["Packaging cost (order line)"]
    )
    var_cost_for_this_line = var_cost_for_line(
        line.typo_in_attribute_name, line.volume, costs["Other costs (m3)"] / 1000
    )

    total_cost = var_cost_for_this_line + fixed_cost_per_line

    return fixed_cost_per_line, var_cost_for_this_line, total_cost
```

Unit testing

- Purpose: To isolate each individual functionality and show that it behaves correctly
- Rigorously test data parsing, cleaning, transformations, etc. Sanity-check models.
- pytest is a great test runner
  - Runs tests in parallel – reduces test suite execution time
  - Test discovery
- Provides a sort of living documentation
  - doctest takes this one step further by enabling executable docstrings

Unit testing – Data processing

For transformations of input data, be rigorous! Enumerate as many cases as possible.

```
80  class TestDataCleaning(unittest.TestCase):
81      def test_normalize_zip_codes(self):
82          assert normalize_zipcode("223 52", "SE") == "22352"
83          assert normalize_zipcode("A223 52", "SE") != "22352"
84          assert normalize_zipcode("000022352", "SE") == "22352"
85          assert normalize_zipcode("352", "SE") == "00352"
86          assert normalize_zipcode("000022352", "SE") == "22352"
87
88          assert normalize_zipcode("ABC 123", "CA") == "ABC"
89          assert normalize_zipcode("ABC123", "CA") == "ABC"
90          assert normalize_zipcode("ABC ", "CA") == "ABC"
```

Unit testing - Models

For complex model code, sanity check simple instances of the problem.

```python
class TestMIP(BaseData):
    def test_split1(self):
        self.solver = pywraplp.Solver(
            "MIP", pywraplp.Solver.CBC_MIXED_INTEGER_PROGRAMMING
        )
        results = {}
        for so, order in self.orders.items():
            results[so] = mip(order, self.f_units, self.solver)

        # Split due to range, store can only handle 1 OL
        assert results["1"][0][0] == "CDC001T"
        assert results["1"][0][1] == "CDC001T"
        assert results["1"][0][2] == "CDC001T"
        assert results["1"][0][3] == "CDC001T"
        assert results["1"][0][4] == "STORE002P"

        # Split due to range
        assert results["2"][0][0] == "CDC002P"
        assert results["2"][0][1] == "CDC002P"
        assert results["2"][0][2] == "STORE002P"

        # Split due to weight limit in CDC002
        assert results["3"][0][0] == "CDC002P"
        assert results["3"][0][1] == "STORE002P"

        # Closest unit with range for zip 2
        assert results["4"][0][0] == "STORE001T"

        # Highest prio unit, non-truck, non store parcel
        assert results["5"][0][0] == "CDC002P"

```

Example of doctest

```
27  def var_cost_for_line(item_qty, item_vol, cost_per_litre):
28      """
29      Return the variable cost part of an order line, given the item quantity, volume and cost per litre
30
31      :return: the variable cost part of an order line
32      :raises: ValueError: if item quantity or volume is negative
33
34      >>> var_cost_for_line(3, 5.0, 0.005)
35      0.075
36
37      Negative quantity or volume will raise an exception:
38      >>> var_cost_for_line(-1, 5.0, 0.005)
39      Traceback (most recent call last):
40          ...
41      ValueError: item_qty must be >= 0
42      >>> var_cost_for_line(3, -2.5, 0.005)
43      Traceback (most recent call last):
44          ...
45      ValueError: item_vol must be >= 0
46      """
```

System tests

- Validates a complete and fully integrated software product
- Purpose: Ensure that system behaviour does not change unintentionally

Automated tests

- Include tests in the CD/CI pipeline (if you have one)
- Keep master clean: prevent merging a branch to master if tests do not pass

Include tests in Dockerfile to run in docker build

```
11  FROM base as unit_tests
12  COPY ./test ./test
13  RUN python -m pytest test/
14
15  FROM base as system_tests
16  COPY ./system_tests ./system_tests
17  RUN python -m system_tests.allocation_logic_test
```

Anticipate errors

- Make exception handling as specific as possible
- Write clear error messages, log all useful information

Be specific in your exception handling

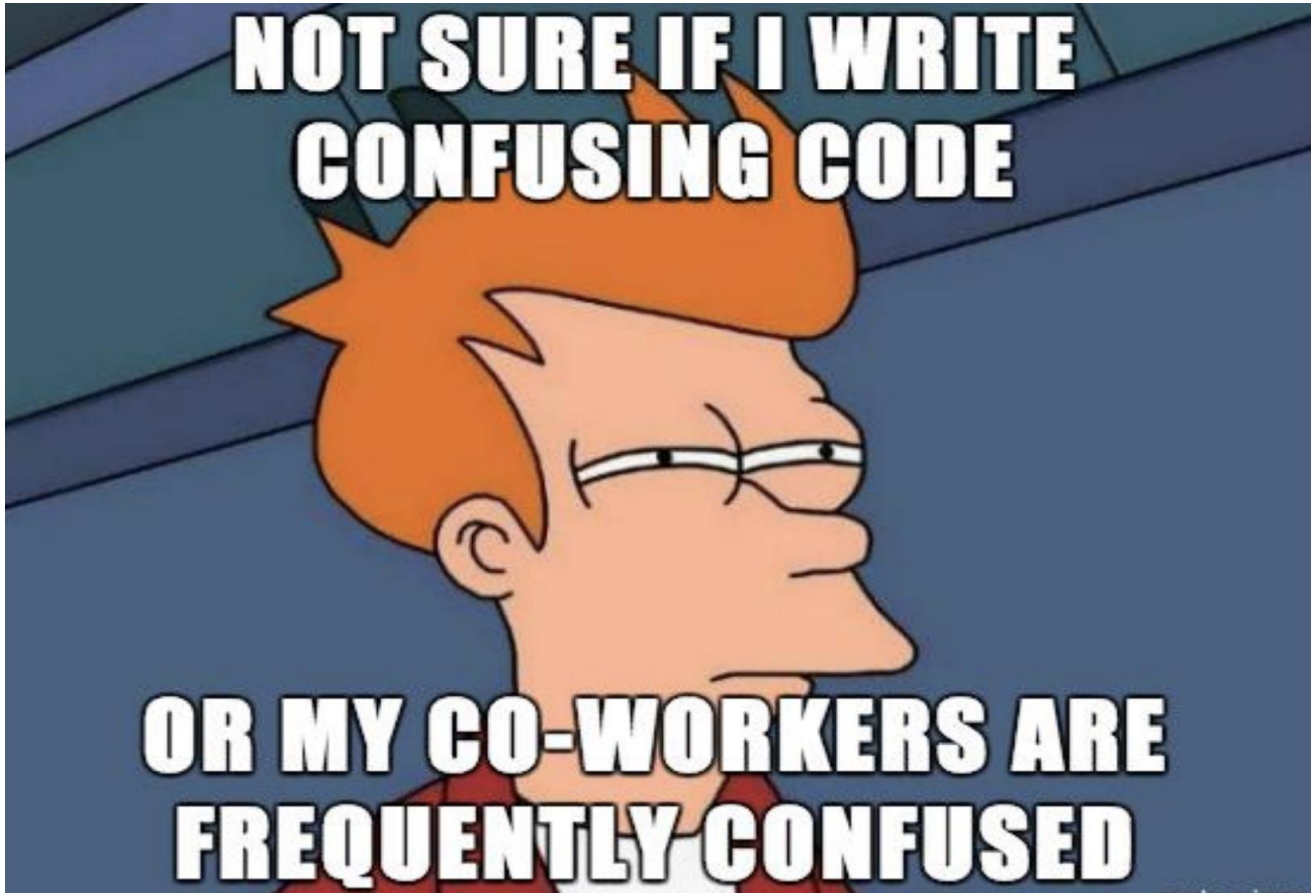
Don't do this:

```
20  def silent_remove(filepath):
21      """
22      Delete a file without raising exception if the file does not exist.
23
24      :param filepath: the path of the file to delete
25      """
26      try:
27          os.remove(filepath)
28      except Exception:
29          pass
30
```

Do this:

```
20  def silent_remove(filepath):
21      """
22      Delete a file without raising exception if the file does not exist.
23
24      :param filepath: the path of the file to delete
25      """
26      try:
27          os.remove(filepath)
28      except OSError:
29          pass
30
```

How to produce maintainable code



- Write clean code
- Write logs
- Document

Clean code

- Linting
    - Forces you to follow the style guide (PEP-8 for Python)
    - Finds syntax errors
    - PyCharm has built-in linting. pylint and flake8 are excellent command-line tools.
    - Pro-tip: Use pre-commit hooks to avoid any dirty commits
- Meaningful function names are better than comments

Clean code

- Linting
    - Forces you to follow the style guide (PEP-8 for Python)
    - Finds syntax errors
    - PyCharm has built-in linting. pylint and flake8 are excellent command-line tools.
    - Pro-tip: Use pre-commit hooks to avoid any dirty commits
- Meaningful function names are better than comments

Pre-commit hooks

1. pip install pre-commit

2. Add .pre-commit-config.yaml (only contains flake8 linting in this example)

```yaml
repos:
-   repo: https://gitlab.com/pycqa/flake8
    rev: 3.7.9
    hooks:
    - id: flake8
```

3. Run pre-commit run --all-files to test your pre-commit pipeline

Now, your commits will fail if the pre-commit pipeline fails

Logging

- Use the logging module instead of print statements
    - Contains diagnostic information
    - Can be selectively filtered
    - Can be conveniently piped to different channels (stdout, file, GCP, etc.)
- Set correct level of log messages (DEBUG, INFO, WARNING, ERROR)
    - Enables filtering and makes the logs much more readable

Documentation

- Add docstrings to all public classes, methods and functions
- In general:
    - Short functions with great docstrings -> Easier to read
    - Long functions with comments scattered throughout -> Harder to read
- Sphinx – a documentation generator for Python
    - Great if you need to publish your Python documentation online

How to produce efficient code

- Determine if performance is good enough
  - *"Pre-mature optimization is the root of all evil."* ™

    - Donald Knuth
- Profile the code
- Optimize performance

Profiling

- time or timeit for simple profiling
    - Conveniently used in a time measuring function decorator
- cProfile
    - Measures time spent in each function and number of times called
- Snakeviz
    - Supports different ways of visualizing results from cProfile

Profiling

To run cProfile:

```
python -m cProfile -o script.profile script_to_profile.py
```

To visualize the results:

```
pip install snakeviz
python –m snakeviz script.profile
```

Performance optimization

- Python provides simple built-in methods for caching

In-memory caching is better suitable for small objects:

```
from functools import lru_cache
@lru_cache(maxsize=256)
def fibonacci(n):
    if n < 2:
        return 1
    return fibonacci(n-2) + fibonacci(n-1)
```

Disk cache is more suitable for large objects:

```
from joblib import memory
@memory.cache
def costly_operation_returning_large_object(x):
    compute(x)
    return large_object
```

Performance optimization

- Two main ways of parallelizing Python code
    - Threading is suitable for I/O-bound applications
    - Multiprocessing is suitable for CPU-bound applications

Thank you!