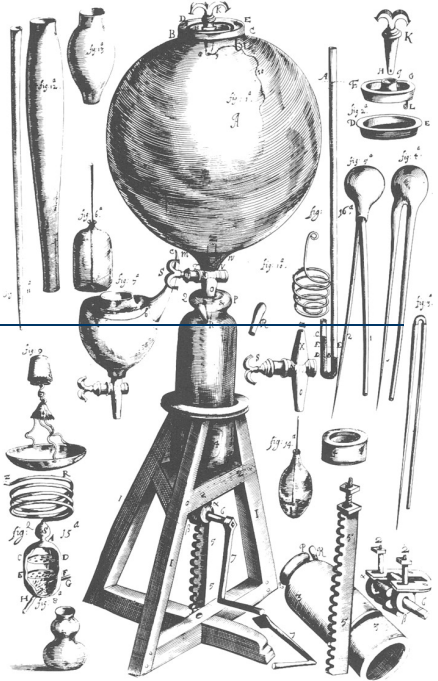
GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Data Science – Basics

Lecture 05 – Clean Code



Fabian Sinz

22. May 2023

Institute for Computer Science – Campus Institute for Data Science (CIDAS)

1. Comprehensibility

Comprehensibility

You (later) or another person can understand what you did (and why).

Tools

- Comments and docstrings
- Literate programming
- Dynamic reports (jupyter)
- Good programming style

1. Comprehensibility
2. Reproducibility

Reproducibility

You (later) or another person can obtain the same result.

Tools

- Dynamic reports (jupyter)
- Virtual environments
- Containerization (Docker)
- Versioning (git)

1. Comprehensibility
2. Reproducibility
3. Consistency

Consistency

The pieces of your code/report fit together. There is a clear order in which each step is executed.

Tools

- Versioning (git)
- Unit tests
- Databases and data models

Elements of good style

1. Comprehensibility
2. Reproducibility
3. Consistency
4. Correctness

Correctness

Your code/analysis is correct and does not produce erroneous results.

Tools

- Unit tests
- Toy examples

Elements of good style

1. Comprehensibility
2. Reproducibility
3. Consistency
4. Correctness
5. Reusability

Reusability

Parts of your code can be reused in a new (similar) context/analysis.

Tools

- Good code factorization/abstraction
- Good programming style

1. Comprehensibility
2. Reproducibility
3. Consistency
4. Correctness
5. Reusability
6. Clarity

Clarity

Your code/analysis clearly and easy to understand.

Tools

- Good code factorization/abstraction
- Good programming style
- Clear report writing.

1. Comprehensibility
2. Reproducibility
3. Consistency
4. Correctness
5. Reusability
6. Clarity
7. Thoroughness

Thoroughness

Your analysis checked all the necessary alternative explanations. You investigated and understood the cause of unexpected results.

Tools

- Clear goals and expectations.
- Good metrics to measure success.

1. Comprehensibility
2. Reproducibility
3. Consistency
4. Correctness
5. Reusability
6. Clarity
7. Thoroughness
8. Robustness

Robustness

Your conclusions do not change with slight variations in the dataset. Your code does not break when the user changes a few parameters. Error messages are meaningful.

Tools

- Unit tests.
- Type hints & docstrings
- Checks and errors.

Comprehensibility

Comprehensibility

A report is comprehensible if

1. if another person can understand how the results were generated.
2. another person can generate the the same/similar results with your report/code.
3. another person can us the insighs from the report to generate new results.

This includes you months after you produced the results.

Compact but uncomprehensible example

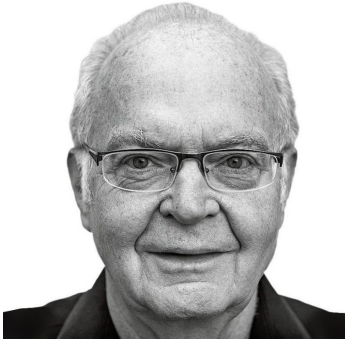
- Most of the results are generated with code.
- This means that your code needs to be comprehensible.

```
1 import json, glob, itertools
2
3 data = list(filter(lambda x: x['value'] == '$100', \
4     itertools.chain(*[json.load(open(f)) for f in glob.glob('data/*.json')])))
```


Literate programming

- The analysis/program provides the logic and explanations in natural language
- Pieces of code interleaved with the explanation carry out the task.
- Most common example in data science

which combines markdown text with code blocks (mostly python)



Donald E. Knuth
Inventor of literate programming and many other things

Literate programming

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

```
The goal of the following lines is to load all jeopardy questions with the value
↪  `$100`.
To this end we
* get all `json` files using `glob`
* load their content using `json`
* combine all questions into one big iterable using `itertools.chain`
* filter for the value `$100` using `filter` and a `lambda` function that returns
↪  true if the list element has `value='$100'`
```

```
1  import json, glob, itertools
2
3  data = list(filter(lambda x: x['value'] == '$100', \
4      itertools.chain(*[json.load(open(f)) for f in glob.glob('data/*.json')])))
```

Programming style (self-documentation)

- Write code in a way that it can be read and understood easily
- Give meaningful names to functions and variables
- Stick with the nameing convention of your programming language

```
1 import json, glob, itertools
2
3 data = list(filter(lambda x: x['value'] == '$100', \
4     itertools.chain(*[json.load(open(f)) for f in glob.glob('data/*.json')])))
```

Programming style (self-documentation)

```
import json, glob, itertools

# list of filenames
all_filenames = glob.glob('data/*.json')

data = list(filter(lambda x: x['value'] == '$100', \
    itertools.chain(*[json.load(open(f)) for f in all_filenames])))
```

Programming style (self-documentation)


```
1  import json, glob, itertools
2
3  # list of filenames
4  all_filenames = glob.glob('data/*.json')
5
6  # list of list of dictionaries
7  file_contents = [json.load(open(f)) for f in all_filenames]
8
9  data = list(filter(lambda x: x['value'] == '$100', \
10    itertools.chain(*file_contents)))
```

Programming style (self-documentation)

```python
import json, glob, itertools

# list of filenames
all_filenames = glob.glob('data/*.json')

# list of list of dictionaries
file_contents = [json.load(open(f)) for f in all_filenames]

# iterator of dictionaries (exhausts!)
all_questions = itertools.chain(*file_contents)

data = list(filter(lambda x: x['value'] == '$100', all_questions))
```

Programming style (self-documentation)

```python
import json, glob, itertools

# list of filenames
all_filenames = glob.glob('data/*.json')

# list of list of dictionaries
file_contents = [json.load(open(f)) for f in all_filenames]

# iterator of dictionaries (exhausts!)
all_questions = itertools.chain(*file_contents)

# returns true if value is $100
has_value_100 = lambda question: question['value'] == '$100'

# filtered list of questions (dictionaries)
data = list(filter(has_value_100, all_questions))
```

Programming style (self-documentation)

```python
import json, glob, itertools

def has_value_100(question: dict) -> bool:
    '''
    Returns True if the value of a question is `$100`
    '''
    return question['value'] == '$100'


filenames_list = glob.glob('data/*.json')
file_content_list = [json.load(open(f)) for f in filenames_list]
question_iterator = itertools.chain(*file_content_list)
question_100_dollar_list = list(filter(has_value_100, question_iterator))
```

Clean and comprehensible code: Zen of Python

The Zen of python (by Tim Peters)

```
1 import this
```

The Zen of python (by Tim Peters)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

The Zen of python – con'd (by Tim Peters)

In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

The Zen of python (by Tim Peters)

Beautiful is better than ugly.

- Code needs to readable and understandable
- Python is popular because it is easy and nice to read

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

Explicit is better than implicit.

- Don’t hide functionality behind obscure language features

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

Simple is better than complex.
Complex is better than complicated.

- Anything can be done with simple and complex techniques
- Use simple solutions for simple problem
- Using simple solutions for complex problems makes the solution complicated

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

Flat is better than nested.

- Programmers often use hierachies to organize a problem/solution
- Sometimes these hierarchies do not add *organization* but only *bureaucracy*.
- No one likes `import spam.eggs.bacon.ham.foo.bar`

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

Sparse is better than dense.

- Do not cram too much functionality into single lines.
- Lines like

```
print('\n'.join("%i bytes = %i bits which has %i possible values." \
    % (j, j*8, 256**j-1) for j in (1 << i for i in range(8))))
```

  will impress your friends but infuriate your coworkers who have to understand it.
- Code spread out over many lines is often easier to read than dense one-liners.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

Readability counts.

- Do not name functions like `strcmp`.
- Do not drop vowels or use obsure abbreviations.
- Code is read more often it's written.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

Special cases aren't special enough to break the rules.
Although practicality beats purity.

- There are best practices for coding that should be followed ...
- ... even when it's tempting to take a shortcut.
- However, if following best practices is very complicated in your code and makes it unreadable, break the rules.
- Use the latter with care.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

Errors should never pass silently.
Unless explicitly silenced.

- It is better for a program to fail fast and loudly than crash in silence.
- Silent errors are hard to debug.
- You can catch and ignore errors, but this should be a conscious choice and not lead to errors when reusing the code.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

In the face of ambiguity, refuse the temptation to guess.

- If your code isn't working, there is a reason and only careful, critical thinking will solve it.
- Refuse the temptation to blindly try solutions until something seems to work
- often you have merely masked the problem rather than solved it.
- This is especially also try for machine learning.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

There should be one—and preferably only one—obvious way to do it.

- If there are 4 ways to solve a thing, you have to learn 4 times as much when coding in that language.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

Although that way may not be obvious at first unless you're Dutch.

- Obviously a joke.
- Guido van Rossum, the creator and BDFL (Benevolent Dictator for Life) of Python, is Dutch.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea.

- Code should not only be understandable by the programmer but also by people who use and maintain it.
- Code that is hard to understand is also hard to debug.
- However, easy code can also be bad.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

The Zen of python (by Tim Peters)

Namespaces are one honking great idea—let's do more of those!

- Python specific.
- Namespaces and scopes avoid naming conflicts.
- But remember: *Flat is better than nested*.

[https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/]

Clean and comprehensible code: PEP 8

Clear code with PEP 8

Python uses Enhancement Proposals (PEPs) to extend the language.

Most important for here (check it out): PEP 8.

PEP 8 naming conventions:

- class names should be CamelCase (`MyClass`)
- variable names should be snake_case and all lowercase (`first_name`)
- function names should be snake_case and all lowercase (`quick_sort()`)
- constants should be snake_case and all uppercase (`PI = 3.14159`)
- modules should have short, snake_case names and all lowercase (`numpy`)
- single quotes and double quotes are treated the same (just pick one and be consistent)

[https://testdriven.io/blog/clean-code-python/]

PEP 8 line formatting:

- indent using 4 spaces (spaces are preferred over tabs)
- lines should not be longer than 79 characters
- avoid multiple statements on the same line
- top-level function and class definitions are surrounded with two blank lines
- method definitions inside a class are surrounded by a single blank line
- imports should be on separate lines

[https://testdriven.io/blog/clean-code-python/]

PEP 8 whitespace:

- avoid extra spaces within brackets or braces
- avoid trailing whitespace anywhere
- always surround binary operators with a single space on either side
- if operators with different priorities are used, consider adding whitespace around the operators with the lowest priority
- don't use spaces around the = sign when used to indicate a keyword argument

[https://testdriven.io/blog/clean-code-python/]

PEP 8 comments:

- comments should not contradict the code
- comments should be complete sentences
- comments should have a space after the # sign with the first word capitalized
- multi-line comments used in functions (docstrings) should have a short single-line description followed by more text

[https://testdriven.io/blog/clean-code-python/]

Formatters and linters

- Formatters help you automatically format your code. Common formatters are
  - black (also available for jupyter notebooks)
  - flake8
  - autopep8
  - yapf
- Linters are tools to detect small coding errors or fragile code patterns. Common linters are
  - PyLint
  - PyFlakes
  - mypy

Clean and comprehensible code: Variable naming conventions

Variable names

- Use nouns for variable names
- Use descriptive/intention-revealing names: Other should be able to correctly guess what the variable stores

```
# This is bad
c = 5
d = 12

# This is good
city_counter = 5
elapsed_time_in_days = 12
```

[https://testdriven.io/blog/clean-code-python/]

Use pronounceable names

- You should be able to explain the code to someone else.
- This means your variable names should be pronounceable.

```python
from datetime import datetime

# This is bad
genyyyymmddhhmmss = datetime.strptime('04/27/95 07:14:22', '%m/%d/%y %H:%M:%S')

# This is good
generation_datetime = datetime.strptime('04/27/95 07:14:22', '%m/%d/%y %H:%M:%S')
```

[https://testdriven.io/blog/clean-code-python/]

Avoid using ambiguous abbreviations

- Do not come up with creative abbreviations
- It's better to have a long name than a confusing name

```
  # This is bad
fna = 'Bob'
cre_tmstp = 1621535852

# This is good
first_name = 'Bob'
creation_timestamp = 1621535852
```

[https://testdriven.io/blog/clean-code-python/]

Always use the same vocabulary

- Avoid using synonyms when naming variables.

```python
# This is bad
client_first_name = 'Bob'
customer_last_name = 'Smith'

# This is good
client_first_name = 'Bob'
client_last_name = 'Smith'
```

[https://testdriven.io/blog/clean-code-python/]

Don’t hard-code ”magic numbers”

- It should be clear what constants mean
- For clarity, you can give them a name

```python
import random

# This is bad
def roll():
    return random.randint(0, 36)  # what is 36 supposed to represent?

# This is good
ROULETTE_POCKET_COUNT = 36

def roll():
    return random.randint(0, ROULETTE_POCKET_COUNT)
```

[https://testdriven.io/blog/clean-code-python/]

Use solution domain names

- To help the reader you can add a suffix with the intended variable type

```python
# This is good
score_list = [12, 33, 14, 24]
word_dict = {
    'a': 'apple',
    'b': 'banana',
    'c': 'cherry',
}

# This is bad
names = ["Nick", "Mike", "John"]
```

[https://testdriven.io/blog/clean-code-python/]

Don’t add redundant context

- Do not add unnecessary data to variable names, especially if you’re working with classes.

```python
# This is bad
class Person:
    def __init__(self, person_first_name, person_last_name, person_age):
        self.person_first_name = person_first_name
        self.person_last_name = person_last_name
        self.person_age = person_age

# This is good
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
```

[https://testdriven.io/blog/clean-code-python/]

Clean and comprehensible code: Function naming conventions

Use verbs for function names

- Do not use different words for the same concept

```python
# This is bad
def get_name(): pass
def fetch_age(): pass

# This is good
def get_name(): pass
def get_age(): pass
```

[https://testdriven.io/blog/clean-code-python/]

Write short and simple functions

- Functions should only perform a single task
- If your function name contains 'and' you can probably split it into two functions.

```python
# This is bad
def fetch_and_display_personnel():
    data = # ...

    for person in data:
        print(person)

# This is good
def fetch_personnel():
    return # ...

def display_personnel(data):
    for person in data:
        print(person)
```

Write short and simple functions

- Keep your arguments at a minimum

```
# This is bad
def render_blog_post(title, author, created_timestamp, \
                     updated_timestamp, content):
    # ...
```

- Don't use flags in functions

```
# This is bad
def transform(text, uppercase):
  return text.upper() if uppercase else text.lower()
```

- Avoid side effects: Try not to modify global variables

[https://testdriven.io/blog/clean-code-python/]

Clean and comprehensible code: Comments

Difference between comment and documentation

- Documentation tells users when and how to use the code.
- Comments tells other developers (us) why the code was written as it is.
- Clean code let's other developers (us) read what was done.

[https://testdriven.io/blog/clean-code-python/]

Rules for comments

1. Do not comment bad code, rewrite it
2. Readable code does not need comments
3. Do not add noise comments
4. Do not leave commented out code

[https://testdriven.io/blog/clean-code-python/]

Clean and comprehensible code: Coding principles

General advice

Some general advice that I learned the hard way (and you will too).

1. It's ok (good/fast) to have messy prototyping code.
2. First make it run, then make it correct.
3. Do not clean up/abstract too early.
4. Do not write code for problems that you do not have.
5. Keep your code flexible/extendible but not too flexible.

DRY (Don’t repeat yourself)

- Code/functionality should not be duplicated.
- If you change the functionality, you should only need to change it at one location.
- DRY code is easy to maintain.

Downsides

- Can result in complex code.
- Can be hard to change bigger parts of the code base.
- Do not DRY to early.

[https://testdriven.io/blog/clean-code-python/]

KISS (Keep it simple, stupid)

- Systems work better and are easier to maintain if they are simple.
- Do not solve problems that you do not have.
- Do not use fancy language features if they do not serve a clear purpose.

[https://testdriven.io/blog/clean-code-python/]

- Clean and comprehensible code

- Clean and comprehensible code
  - helps you understand the code better

- Clean and comprehensible code
  - helps you understand the code better
  - helps you to avoid bugs or find them faster

- Clean and comprehensible code
  - helps you understand the code better
  - helps you to avoid bugs or find them faster
  - is easier to maintain

- Clean and comprehensible code
  - helps you understand the code better
  - helps you to avoid bugs or find them faster
  - is easier to maintain
  - is the best documentation of what your analysis did.

- Clean and comprehensible code
  - helps you understand the code better
  - helps you to avoid bugs or find them faster
  - is easier to maintain
  - is the best documentation of what your analysis did.
- Today we disussed tools and principles to keep you code clean and clear

- Clean and comprehensible code
  - helps you understand the code better
  - helps you to avoid bugs or find them faster
  - is easier to maintain
  - is the best documentation of what your analysis did.
- Today we disussed tools and principles to keep you code clean and clear
- In upcoming lectures we discuss tools to keep you code and results consistent and reproducible.

Thanks for listening.
Questions?