

# Best Practices

Nicola Chiapolini

Physik-Institut  
University of Zurich

June 8, 2015

Based on talk by Valentin Haenel <https://github.com/esc/best-practices-talk>



This work is licensed under the [Creative Commons Attribution-ShareAlike 3.0 License](https://creativecommons.org/licenses/by-sa/3.0/).

# Introduction

## Introduction

- ▶ We write code regularly
- ▶ We have not been formally trained

## Best Practices

- ▶ evolved from experience
- ▶ increase productivity
- ▶ decrease stress
- ▶ still evolve with tools and languages

## Development Methodologies

- ▶ e.g. Agile Programming or Test Driven Development
- ▶ lots of buzzwords
- ▶ still many helpful ideas

# Outline

Introduction

Style and Documentation

Special Python Statements

KIS(S) & DRY

Refactoring

Development Methodologies

# Outline

Introduction

Style and Documentation

Special Python Statements

KIS(S) & DRY

Refactoring

Development Methodologies

## Coding Style

- ▶ readability counts
- ▶ explicit is better than implicit
- ▶ give variables *intention revealing* names
  - ▶ For example: `numbers` instead of `n`
  - ▶ For example: `numbers` instead of `list_of_float_numbers`
  - ▶ See also: [Ottingers Rules for Naming](#)

### Example

```
def my_product(numbers):  
    """ Compute the product of a sequence of numbers. """  
    total = 1  
    for item in numbers:  
        total *= item  
    return total
```

## Formatting Code

- ▶ use coding conventions
- ▶ conventions specify:
  - ▶ variable naming
  - ▶ indentation
  - ▶ import
  - ▶ maximum line length
  - ▶ blank lines, whitespace, comments
- ▶ e.g: [PEP-8](#)
- ▶ OR use a consistent style (especially when collaborating)

### Tools

- ▶ [pylint](#)
- ▶ [pep8](#)
- ▶ [flake8](#)

# Documenting Code: Docstrings

## Example

```
def my_product(numbers):  
    """ Compute the product of a sequence of numbers. """
```

- ▶ at least a single line
- ▶ also for yourself
- ▶ is on-line help too
  
- ▶ Document arguments and return objects, including types
- ▶ For complex algorithms, document every line, and include equations in docstring
- ▶ Use the [numpy docstring conventions](#)

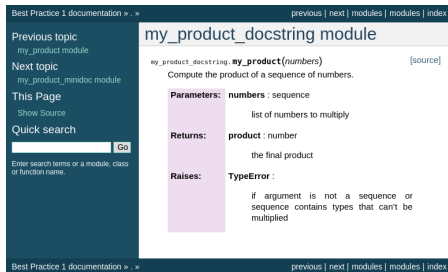


## Example Docstring

```
def my_product(numbers):  
    """ Compute the product of a sequence of numbers.  
  
    Parameters  
    -----  
    numbers : sequence  
        list of numbers to multiply  
  
    Returns  
    -----  
    product : number  
        the final product  
  
    Raises  
    -----  
    TypeError  
        if argument is not a sequence or sequence contains  
        types that can't be multiplied  
    """
```

# Documenting Code

- ▶ tools generate website from docstrings
  - ▶ [pydoc](#)
  - ▶ [epydoc](#)
  - ▶ [sphinx](#)
- ▶ when project gets bigger
  - ▶ how-to
  - ▶ FAQ
  - ▶ quick-start



Best Practice 1 documentation » » previous | next | modules | modules | index

## my\_product\_docstring module

my\_product\_docstring.**my\_product**(numbers) [\[source\]](#)

Compute the product of a sequence of numbers.

**Parameters:** **numbers** : sequence  
list of numbers to multiply

**Returns:** **product** : number  
the final product

**Raises:** **TypeError** :  
if argument is not a sequence or sequence contains types that can't be multiplied

Best Practice 1 documentation » » previous | next | modules | modules | index

# Outline

Introduction

Style and Documentation

**Special Python Statements**

KIS(S) & DRY

Refactoring

Development Methodologies

## import

- ▶ Don't use the *star import*: `from module import *`
  - ▶ hard to read
  - ▶ modules may overwrite each other
  - ▶ Where does this function come from?
  - ▶ will import *everything* in a module
  - ▶ ...unless you have a very good reason: e.g. `pylab`, `interactive`
- ▶ Put all imports at the beginning of the file...
- ▶ ...unless you have a very good reason

### Example

```
import my_product as mp
mp.my_product([1,2,3])

from my_product import my_product
my_product([1,2,3])
```

# Exceptions

- ▶ use `try except` and `raise`
- ▶ often better than `if` (e.g. `IndexError`)

## Example

```
try:
    my_product(1,2,3)
except TypeError:
    print "'my_product' expects a sequence"
    raise TypeError
```

- ▶ don't use *special* return values:  
1, 0, False, None
- ▶ Fail early, fail often
- ▶ use [built-in Exceptions](#)

# Outline

Introduction

Style and Documentation

Special Python Statements

**KIS(S) & DRY**

Refactoring

Development Methodologies

# Keep it Simple (Stupid) – KIS(S) Principle

Keep it Simple

# Keep it Simple (Stupid) – KIS(S) Principle

Keep it Simple



## Don't Repeat Yourself (DRY)

- ▶ No copy & paste!
- ▶ Not just lines code, but knowledge of all sorts
- ▶ Do not express the same piece of knowledge in two places...
- ▶ ...or you will have to update it everywhere
- ▶ It is not a question of *if* this may fail, but *when*

## Don't Repeat Yourself (DRY): Types

### Example

- ▶ Version number in source code, website, readme, package filename
- ▶ Copy-and-paste a snippet, instead of refactoring it into a function
- ▶ Repeated implementation of utility methods
  - ▶ because you don't remember
  - ▶ because you don't know the libraries

```
numpy.prod([1,2,3])
```

- ▶ because developers don't talk to each other

- ▶ If you detect duplication: refactor mercilessly!

## Don't Repeat Yourself (DRY): Types

### Example

- ▶ Version number in source code, website, readme, package filename
- ▶ Copy-and-paste a snippet, instead of refactoring it into a function
- ▶ Repeated implementation of utility methods
  - ▶ because you don't remember
  - ▶ because you don't know the libraries

```
numpy.prod([1,2,3])
```

- ▶ **because developers don't talk to each other**
- 
- ▶ If you detect duplication: refactor mercilessly!

# Outline

Introduction

Style and Documentation

Special Python Statements

KIS(S) & DRY

**Refactoring**

Development Methodologies

# Refactoring

- ▶ re-organise your code without changing its function
- ▶ rethink earlier design decisions
- ▶ break large code blocks apart
- ▶ rename and restructure code
  
- ▶ will improve the readability and modularity
- ▶ will usually reduce the lines of code

# Common Refactoring Operations

- ▶ Rename class/method/module/package/function
- ▶ Move class/method/module/package/function
- ▶ Encapsulate code in method/function
- ▶ Change method/function signature
- ▶ Organise imports (remove unused and sort)
  
- ▶ Always refactor one step at a time, and ensure code still works
  - ▶ version control
  - ▶ unit tests

## Refactoring Example

```
def my_func(numbers):  
    """ Difference between sum and product of sequence. """  
    total = 0  
    for item in numbers:  
        total += item  
    total2 = 1  
    for item in numbers:  
        total2 *= item  
    return total2 - total
```

- ▶ split into functions
- ▶ use libraries/built-ins
- ▶ fix bug

## Refactoring Example

```
from my_math import my_product, my_sum

def my_func(numbers):
    """ Difference between sum and product of sequence. """
    sum_value = my_sum(numbers)
    product_value = my_product(numbers)
    return product_value - sum_value
```

- ▶ split into functions
- ▶ use libraries/built-ins
- ▶ fix bug



## Refactoring Example

```
from numpy import prod, sum

def my_func(numbers):
    """ Difference between sum and product of sequence. """
    sum_value = sum(numbers)
    product_value = prod(numbers)
    return product_value - sum_value
```

- ▶ split into functions
- ▶ use libraries/built-ins
- ▶ fix bug

## Refactoring Example

```
from numpy import prod, sum

def my_func(numbers):
    """ Difference between sum and product of sequence. """
    sum_value = sum(numbers)
    product_value = prod(numbers)
    return sum_value - product_value
```

- ▶ split into functions
- ▶ use libraries/built-ins
- ▶ fix bug

# Outline

Introduction

Style and Documentation

Special Python Statements

KIS(S) & DRY

Refactoring

**Development Methodologies**

# What is a Development Methodology?

## Consists of:

- ▶ attitude, style and approach towards development
- ▶ tools and models to support approach

## Help answer questions like:

- ▶ How far ahead should I plan?
- ▶ What should I prioritise?
- ▶ When do I write tests and documentation?

Right methodology depends on scenario.

# What is a Development Methodology?

## Consists of:

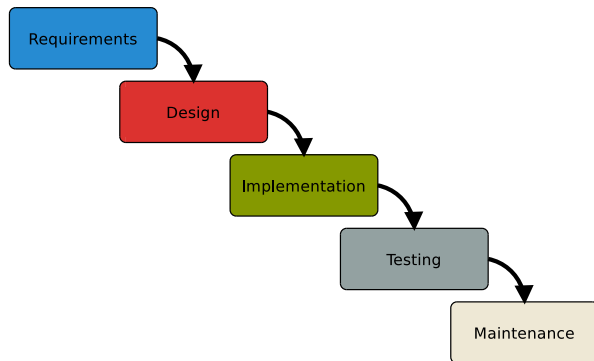
- ▶ attitude, style and approach towards development
- ▶ tools and models to support approach

## Help answer questions like:

- ▶ How far ahead should I plan?
- ▶ What should I prioritise?
- ▶ When do I write tests and documentation?

Right methodology depends on scenario.

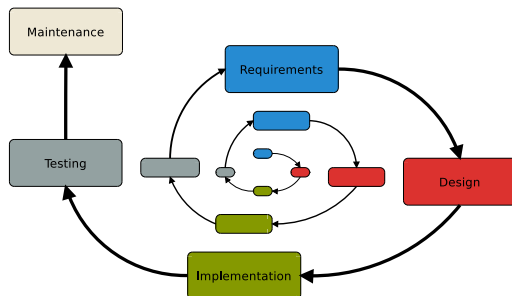
# The Waterfall Model, Royce 1970



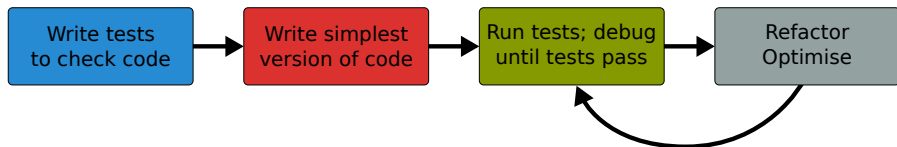
- ▶ sequential
- ▶ from manufacturing and construction

## Agile Methods (late 90's)

- ▶ minimal planning, small development iterations
- ▶ design/implement/test on a modular level
- ▶ frequent input from team/customer/boss/professor
- ▶ very adaptive, since nothing is set in stone



# Test Driven Development (TDD)



- ▶ Define unit tests first!
- ▶ Develop one unit at a time!
- ▶ more tomorrow