

Best Practices, Development Methodologies, and the Zen of Python

Valentin Haenel

Freelance Consultant and Software Developer

<http://haenel.co>

1 Sept 2013

Version: 2013-09-Zurich-4-gf9e5fa8

<https://github.com/esc/best-practices-talk>



This work is licensed under the *Creative Commons Attribution-ShareAlike 3.0 License*.

Introduction

- Many scientists write code regularly but few have been formally trained to do so
- **Best practices** evolved from programmer's folk wisdom
- They increase productivity and decrease stress
- **Development methodologies**, such as Agile Programming and Test Driven Development, are established in the software engineering industry
- We can learn a lot from them to improve our coding skills
- When programming in Python: Always bear in mind the **Zen of Python**

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Coding Style

- Readability counts
- Explicit is better than implicit
- Beautiful is better than ugly
- Give your variables *intention revealing* names
 - For example: `numbers` instead of `nu`
 - 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

- Format code to coding conventions
- for example: **PEP-8**
- OR use a consistent style (especially when collaborating)
- Conventions Specify:
 - variable naming convention
 - Indentation
 - import
 - maximum line length
 - blank lines, whitespace, comments
- Use automated tools to check adherence (aka static checking):
 - **pylint**
 - **pychecker**
 - **pep8**
 - **pyflakes**
 - **flake8** (combination of pep8 and pyflakes)

Documenting Code

- Minimum requirement: at least a single line docstring
- Not only for others, but also for yourself!
- Serves as on-line help in the interpreter
- Document arguments and return objects, including types
- Use the **numpy docstring conventions**
- Use tools to automatically generate website from docstrings
 - **pydoc**
 - **epydoc**
 - **sphinx**
- For complex algorithms, document every line, and include equations in docstring
- When your project gets bigger: provide a *how-to*, *FAQ* or *quick-start* on your website

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  
    """
```

Example Autogenerated Website

Table of Contents

[Everything](#)

Modules

[my_product_docstring](#)

[\[hide private\]](#)

Everything

All Functions

[my_product_docstring.my](#)

All Variables

[my_product_docstring._p](#)

[\[hide private\]](#)

Home Trees Indices Help	
Module <code>my_product_docstring</code> [hide private] (frames) no frames	
Module <code>my_product_docstring</code> source code	
Functions [hide private]	
	source code
<code>my_product</code> (numbers)	Compute the product of a sequence of numbers.
Variables [hide private]	
	<code>_package_</code> = None
Function Details [hide private]	
<code>my_product(numbers)</code> source code	
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	
Home Trees Indices Help	
Generated by Epydoc 3.0.1 on Thu Sep 1 13:10:11 2011	
http://epydoc.sourceforge.net	

Using Exceptions

- Use the `try except` statements to detect anomalous behaviour:

Example

```
try:
    my_product(['abc', 1])
except TypeError:
    print "'my_product' failed due to a 'TypeError'"
```

- Allow you to recover or fail gracefully
- Resist the temptation to use *special* return values: they will backfire!
 - `(-1, 0, False, None)`
- Fail early, fail often
- Errors should never pass silently...
- Unless explicitly silenced

Appropriate Exceptions

- Python has a **built-in Exception hierarchy**
- These will suit your needs most of the time, If not, subclass them

Exception

```
+-- StandardError
    +-- ArithmeticError
        +-- FloatingPointError
        +-- OverflowError
        +-- ZeroDivisionError
    +-- AssertionError
    +-- IndexError
    +-- TypeError
    +-- ValueError
```

import Pitfalls

- Don't use the *star import*: `import *`
 - Code is hard to read
 - Modules may overwrite each other
 - Where does this function come from?
 - You will import *everything* in a module
 - ...unless you are using the interpreter interactively
- Put all imports at the beginning of the file...
- ...unless you have a very good reason to do otherwise

import foobar as fb **VS** from foo import bar

Example

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

- + origin of my_product known
- - slightly more to type
- - fails only on call (late)

Example

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

- + slightly less to type
- + fails on import (early)
- - must look at import for origin

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Write and Run Unit Tests

- We wish to automate testing of our software
- Instead of testing the whole system we test *units*

Definition of a *Unit*

- The smallest testable piece of code
- Example: `my_product`

Available Packages

- In python we have several packages available:
 - `unittest`
 - `nosetest`
 - `py.test`
 - `Mock`
- Tests increase the confidence that your code works correctly, not only for yourself but also for your reviewers
- Tests are the only way to trust your code
- It might take you a while to get used to writing them, but it will pay off quite rapidly

Example Unit-Tests

Example

```
import nose.tools as nt
from my_product import my_product

def test_my_product():
    """ Test my_product() on simple case. """
    nt.assertEqual(my_product([1, 2, 3]), 6)

def test_raise_type_error():
    """ Test that my_product raises a type error. """
    nt.assert_raises(TypeError, my_product, ['abc', 1, 2, 3])
```

Running the Tests

```
zsh» nosetests
```

```
.F
```

```
=====
```

```
FAIL: Test that my_product raises a type error.
```

```
-----
```

```
Traceback (most recent call last):
```

```
File "/usr/lib/pymodules/python2.6/nose/case.py", line 183, in runTest
    self.test(*self.arg)
```

```
File "/home/valentin/git-working/best-practices-talk/\
    code/my_product_test.py", line 10, in test_raise_type_error
    nt.assert_raises(TypeError, my_product, ['abc', 1, 2, 3])
```

```
AssertionError: TypeError not raised
```

```
-----
```

```
Ran 2 tests in 0.011s
```

```
FAILED (failures=1)
```

Whats going on?

```
>>> my_product(['abc', 1, 2, 3])  
      'abcbcabcbcabcbcabcb'
```

A Sneak Preview of Test-Driven Development (TDD)

```
from numbers import Number

def my_product(numbers):
    """ Compute the product of a sequence of numbers. """
    total = 1
    for item in numbers:
        if not isinstance(item, Number):
            raise TypeError("%r unsupported by 'my_product'"
                             "%type(item))"
        total *= item
    return total
```

But make sure that it works!

Example

```
from my_product_fixed import my_product
```

```
zsh» nosetests my_product_fixed_test.py
```

```
..
```

```
-----  
Ran 2 tests in 0.001s
```

```
OK
```

Goals and Benefits

Goals

- check code works
- check design works
- catch regression

Benefits

- Easier to test the whole, if the units work
- Can modify parts, and be sure the rest still works
- Provide examples of how to use code

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Motivation to use Version Control

Problem 1

"Help! my code worked yesterday, but I can't recall what I changed."

- Version control is a method to track and retrieve modifications in source code

Problem 2

"We would like to work together, but we don't know how!"

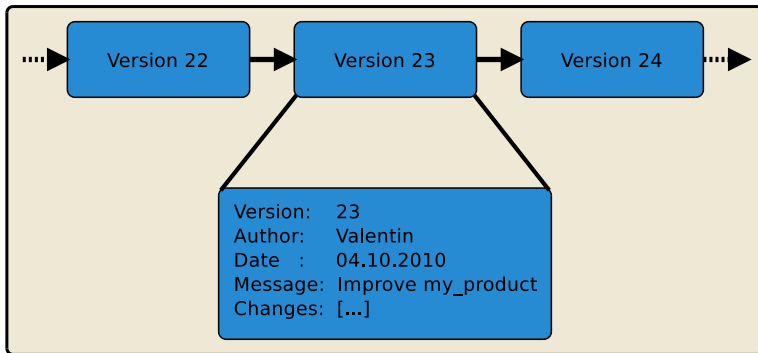
- Concurrent editing by several developers is possible via merging

Features

- Checkpoint significant improvements, for example releases
- Document developer effort
 - Who changed what, when and why?
- Use version control for anything that's text
 - Code
 - Thesis/Papers
 - (Love) letters
- Easy collaboration across the globe

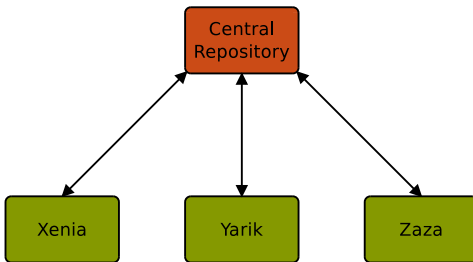
Vocabulary

- Modifications to code are called *commits*
- Commits are stored in a *repository*
- Adding commits is called *committing*



Centralised Version Control

- All developers connect to a single resource over the network
- Any interaction (history, previous versions, committing) require network access



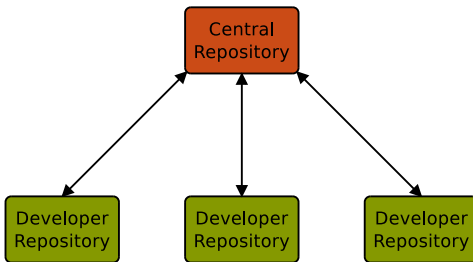
- Example systems: **Subversion (svn)**, **Concurrent Version System (cvs)**

Distributed Version Control

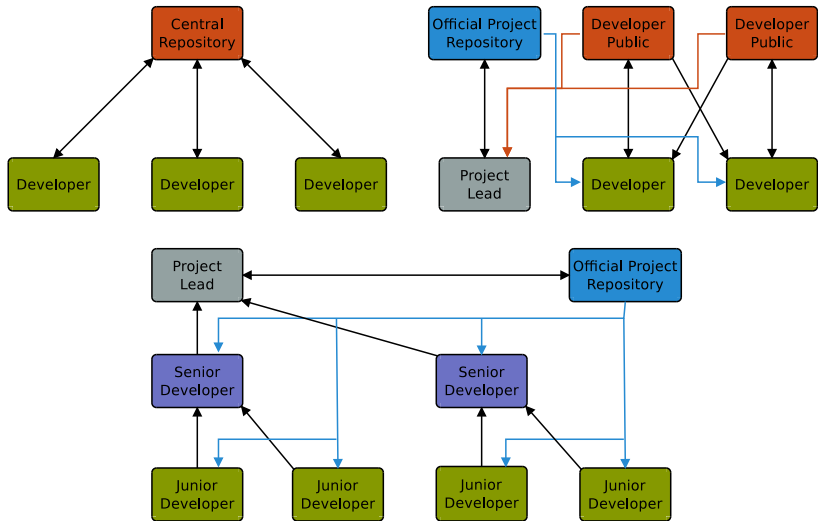
- Several copies of the repository may exist all over the place
- Network access only required when synchronising repositories
- Much more flexible than centralised
- Widely regarded as state-of-the-art
- Example systems: **git**, **Mercurial (hg)**, **Bazaar (bzt)**

Distributed like Centralised

- ... except that each developer has a *complete* copy of the entire repository



Distributed Supports any Workflow :-)



What we will use...



Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Refactor Continuously

- As a program evolves it may become necessary to rethink earlier decisions and adapt the code accordingly
- Re-organisation of your code without changing its function
- Increase modularity by breaking large code blocks apart
- Rename and restructure code to increase readability and reveal intention
- Always refactor one step at a time, and use the tests to check code still works
- Now is better than never
- Although never is often better than *right* now

Common Refactoring Operations

- Rename class/method/module/package/function
- Move class/method/module/package/function
- Encapsulate code in method/function
- Change method/function signature
- Organize imports (remove unused and sort)

- Generally you will improve the readability and modularity of your code
- Usually refactoring will reduce the lines of code

Refactoring Example

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

Split into functions, and use built-ins

```
from my_product import my_product

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

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Do not Repeat Yourself (DRY Principle)

- When developing software, avoid duplication
- No cut&paste!
- Not just lines code, but knowledge of all sorts
- Do not express the same piece of knowledge in two places
- If you need to update this knowledge you will have to update it everywhere
- It is not a question of *how* this may fail, but instead a question of *when*
- Categories of Duplication:
 - Imposed Duplication
 - Inadvertent Duplication
 - Impatient Duplication
 - Interdeveloper Duplication
- If you detect duplication in code thats already written, refactor mercilessly!

Imposed Duplication

- When duplication seems to be forced on us
- We feel like there is no other solution
- The environment or programming language seems to require duplication

Example

- Duplication of a program version number in:
 - Source code
 - Website
 - Licence
 - README
 - Distribution package
- Result: Increasing version number consistently becomes difficult

Inadvertent Duplication

- When duplication happens by accident
- You don't realize that you are repeating yourself

Example

- Variable name: `list_of_numbers` instead of just `numbers`
- Type information duplicated in variable name
- What happens if the set of possible types grows or shrinks?
- Side effect: Type information incorrect, function may operate on any sequence such as tuples

Impatient Duplication

- Duplication due to sheer laziness
- Reasons:
 - End-of-day
 - Deadline
 - Insert `pretext` here

Example

- Copy-and-paste a snippet, instead of refactoring it into a function
 - What happens if the original code contains a bug?
 - What happens if the original code needs to be changed?
-
- By far the easiest category to avoid, but requires discipline and willingness
 - Be patient, invest time now to save time later! (especially when facing oh so important deadlines)

Interdeveloper Duplication

- Repeated implementation by more than one developer
- Usually concerns utility methods
- Often caused by lack of communication
- Or lack of a module to contain utilities
- Or lack of library knowledge

Interdeveloper Duplication Example

- Product function may already exist in some library
- (Though I admit this may also be classified as impatient duplication)

Example

```
import numpy

def my_product_refactor(numbers):
    """ Compute the product of a sequence of numbers. """
    return numpy.prod(numbers)
```

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - **Keep it Simple**
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

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

- Resist the urge to over-engineer
- Write only what you need now
- Simple is better than complex
- Complex is better than complicated
- Special cases aren't special enough to break the rules
- Although practicality beats purity

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

What is a Development Methodology?

Consists of:

- An attitude that informs the style and approach towards development
- A set of tools and models to support that particular approach

Help answer the following questions:

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

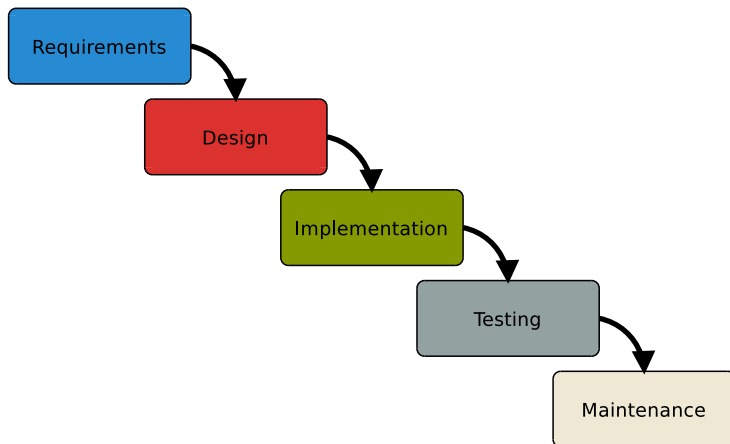
Scenarios

- Lone student/scientist



- Small team of scientists, working on a common library
- Speed of development more important than execution speed
- Often need to try out different ideas quickly:
 - rapid prototyping of a proposed algorithm
 - re-use/modify existing code

An Example: The Waterfall Model, Royce 1970



- Sequential software development process
- Originates in the manufacturing and construction industries
- Rigid, inflexible model—focusing on one stage at a time

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

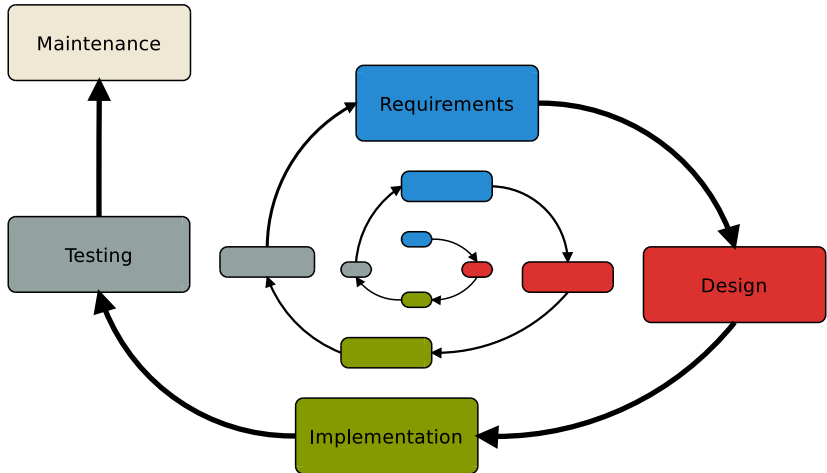
Agile Methods

- Agile methods emerged during the late 90's
- Generic name for set of more specific paradigms
- Set of *best practices*
- Particularly suited for:
 - Small teams (Fewer than 10 people)
 - Unpredictable or rapidly changing requirements...
 - ... isn't this what science is all about?

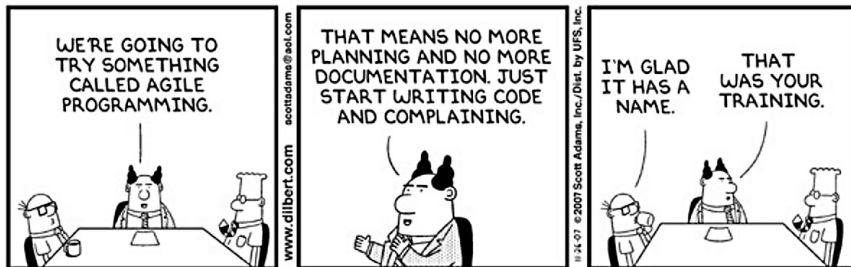
Prominent Features of Agile methods

- Minimal planning, small development iterations
- Design/implement/test on a modular level
- Rely heavily on testing
- Promote collaboration and teamwork, including frequent input from customer/boss/professor
- Very adaptive, since nothing is set in stone

The Agile Spiral



Agile methods

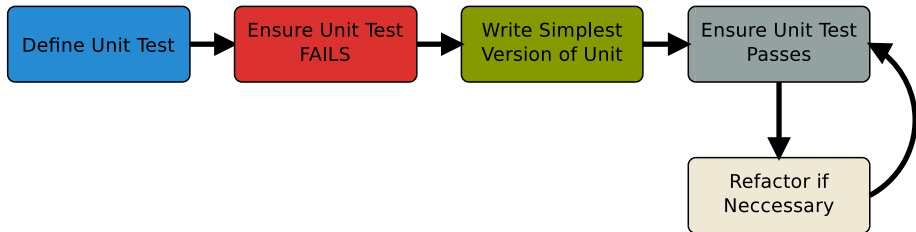


© Scott Adams, Inc./Dist. by UFS, Inc.

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - **Test Driven Development**
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Test Driven Development (TDD)



- Define unit tests first!
- Develop one unit at a time!

Benefits of TDD

- Encourages simple designs and inspires confidence
- No one ever *forgets* to write the unit tests
- Helps you design a good API, since you are forced to use it when testing (*dog fooding*)
- Perhaps you may want to even write the documentation first?

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

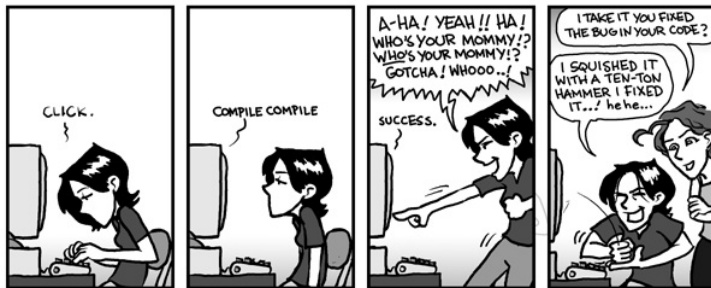
Dealing with Bugs — The Agile Way

- Write a unit test to expose the bug
- Isolate the bug using a debugger
- Fix the code, and ensure the test passes
- Use the test to catch the bug should it reappear (regression)

Debugger

- A program to run your code one step at a time, and giving you the ability to inspect the current state
- For example:
 - `pdb`
 - `ipdb` (IPython enhanced `pdb`)
 - `winpdb`
 - `puddb`
 - `pydb`|`pydbgr`

Dealing with Bugs?



phd.stanford.edu/

Design by Contract

- Functions carry their specifications around with them:
 - Keeping specification and implementation together makes both easier to understand
 - ...and improves the odds that programmers will keep them in sync
- A function is defined by:
 - pre-conditions: what must be true in order for it to work correctly
 - post-conditions: what it guarantees will be true if pre-conditions are met
- Pre- and post-conditions constrain how the function can evolve:
 - can only ever relax pre-conditions (i.e., take a wider range of input)...
 - ...or tighten post-conditions (i.e., produce a narrower range of output)
 - tightening pre-conditions, or relaxing post-conditions, would violate the function's contract with its callers

Defensive Programming

- Specify pre- and post-conditions using assertion:
 - `assert len(numbers) > 0`
 - `raise AssertionError`
- Use assertions liberally
- Program as if the rest of the world is out to get you!
- Fail early, fail often, fail better!
- The less distance there is between the error and you detecting it, the easier it will be to find and fix
- It's never too late to do it right
 - Every time you fix a bug, put in an assertion and a comment
 - If you made the error, the right code can't be obvious
 - You should protect yourself against someone "simplifying" the bug back in

Pair Programming

- Two developers, one computer
- Two roles: driver and navigator
- Driver sits at keyboard
 - Can focus on the tactical aspects
 - See only the “road” ahead
- Navigator observes and instructs
 - Can concentrate on the “map”
 - Pay attention to the big picture
- Switch roles every so often!
- In a team: switch pairs every so often!

Pair Programming — Benefits

- Know-How is shared/transferred:
 - Specifics of the system
 - Tool usage (editor, interpreter, debugger, version control)
 - Coding style, idioms, knowledge of library
- Less likely to:
 - Surf the web, read personal email
 - Be interrupted by others
 - Cheat themselves (being impatient, taking shortcuts)
- Pairs produce code which:¹
 - Is shorter
 - Incorporates better designs
 - Contains fewer defects...
 - ... $1+1 > 2$!

¹Cockburn, Alistair, Williams, Laurie (2000). *The Costs and Benefits of Pair Programming*

Optimization for Speed — My Point of View

- Readable code is usually better than fast code
- Programmer/Scientist time is more valuable than computer time
- Don't optimize early, ensure code works, has tests and is documented...
- *before* starting to optimize
- Only optimize if it is absolutely necessary
- Only optimize your bottlenecks
- ...and identify these using a profiler

Profilers and Viewers

Profiler

- A tool to measure and provide statistics on the execution of code.
 - `timeit`
 - `cProfile`
 - `line profiler`
 - Memory profiler

Viewer

- Viewers display the profiler output, usually a call-graph
 - `gprof2dot`
 - `run snake run`
 - `kcacheogrind`

Quick Example

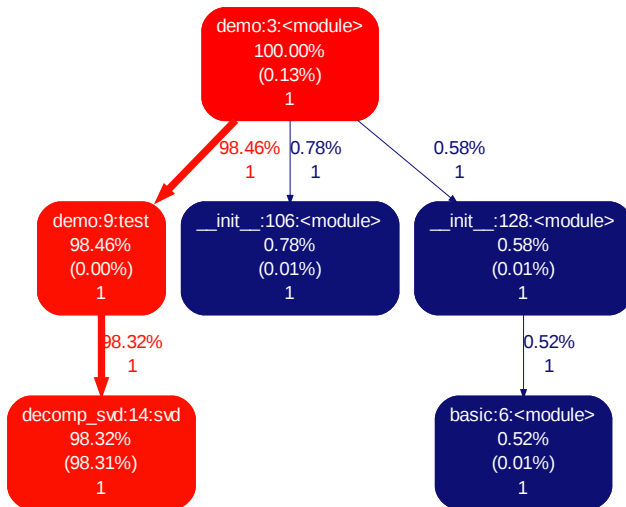
Call the Profiler

```
zsh» python -m cProfile -o profile-stats \  
./wiki2beamer-0.9.2 slides.wiki > /dev/null
```

Generate the Call-Graph

```
zsh» gprof2dot.py -f pstats profile-stats |  
dot -Tpdf -o profile_stats.svg
```

Call-Graph



Prototyping

- Ever tried to hit a moving target?
- If you are unsure how to implement something, write a prototype
- Hack together a proof of concept quickly
- No tests, no documentation, keep it simple (stupid)
- Use this to explore the feasibility of your idea
- When you are ready, scrap the prototype and start with the unit tests
- In the face of ambiguity, refuse the temptation to guess

Quality Assurance

- The techniques I have mentioned above help to assure high quality of the software
- Quality is not just testing:
 - Trying to improve the quality of software by doing more testing is like trying to lose weight by weighing yourself more often
- Quality is designed in (For example, by using the DRY and KISS principles)
- Quality is monitored and maintained through the whole software life cycle

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

The Zen of Python

```
>>>import this
```

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
```

```
Explicit is better than implicit.
```

```
Readability counts.
```

```
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.
```

```
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.
```

Outline

- 1 Introduction
- 2 Best Practices
 - Style and Documentation
 - Unit Tests
 - Version Control
 - Refactoring
 - Do not Repeat Yourself
 - Keep it Simple
- 3 Development Methodologies
 - Definition and Motivation
 - Agile Methods
 - Test Driven Development
 - Additional techniques
- 4 Zen of Python
- 5 Conclusion

Results

- Every scientific result (especially if important) should be independently reproduced at least internally before publication. (German Research Council 1999)
- Increasing pressure to make the source code (and data?) used in publications available
- With unit tested code you need not be embarrassed to publish your code
- Using version control allows you to share and collaborate easily
- In this day and age there is **absolutely no excuse** to not use them!
- If you can afford it, hire a developer :-)

The Last Slide

- Slides based on:
 - Material by Pietro Berkes and Tiziano Zito
 - *The Pragmatic Programmer* by Hunt and Thomas,
 - *The Course on Software Carpentry* by Greg Wilson
- Open source tools used to make this presentation:
 - Wiki2beamer
 - L^AT_EXbeamer
 - Dia
 - Pygments
 - Minted
 - Solarized theme for pygments

Questions ?

<https://github.com/esc/best-practices-talk>

<http://krzz.de/u>