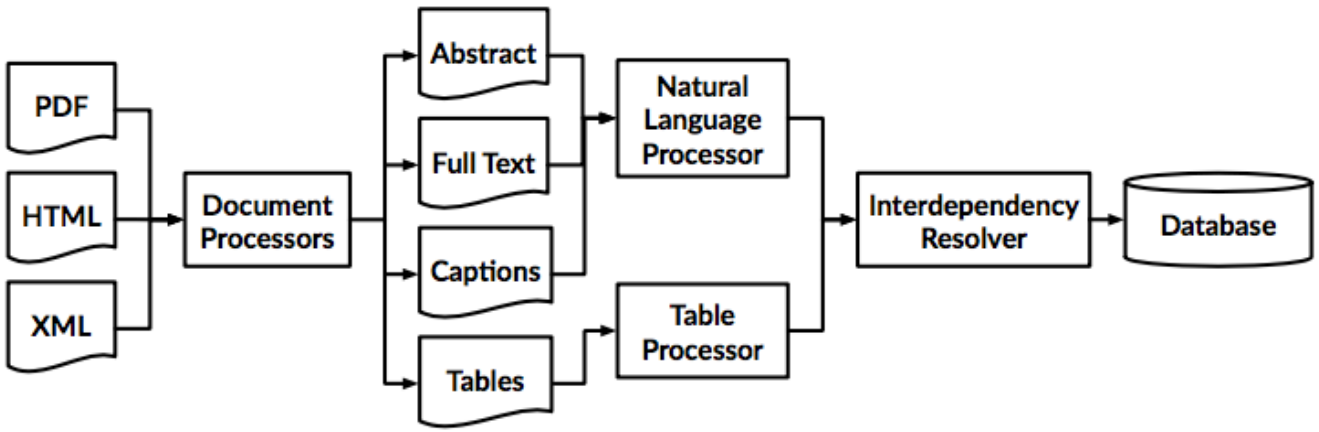
ChemDataExtractor: An Introduction for New Developers

Callum Court
Molecular Engineering Group,
Department of Physics,
University of Cambridge

Overview

ChemDataExtractor

- **A comprehensive toolkit for the automated extraction of chemical information from scientific documents.**

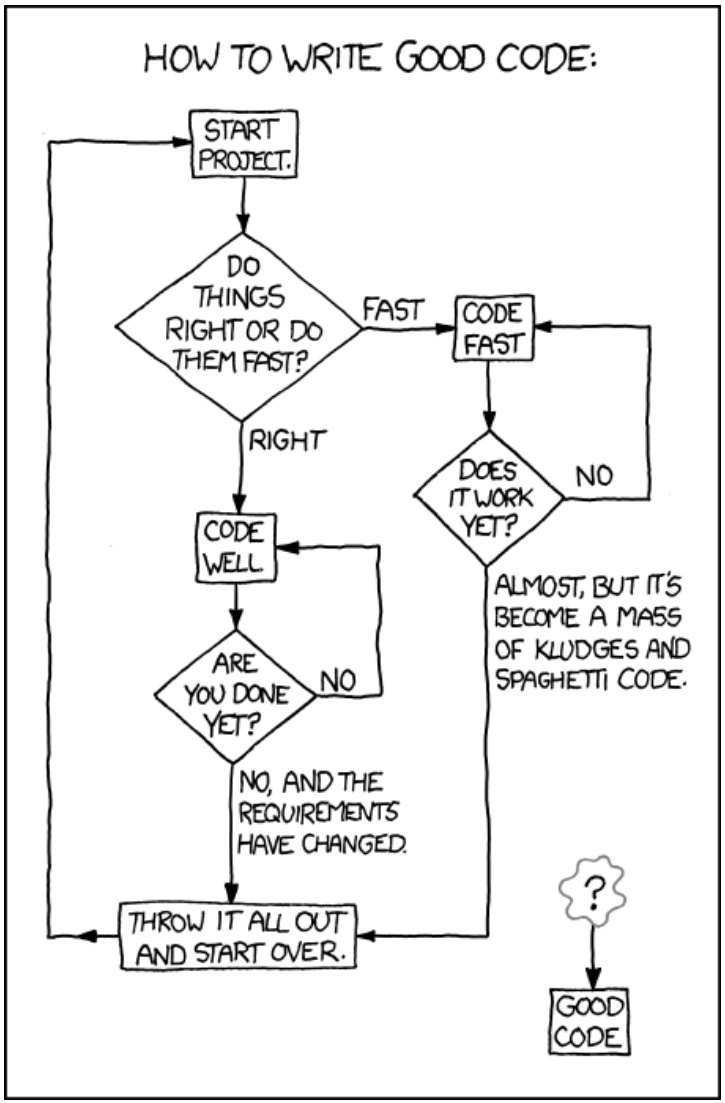


The ChemDataExtractor Developers Group

- The number of researchers working on ChemDataExtractor is reaching critical mass.
- Resulting in a number of unstable development versions.
- All source code is now unified under a single stable development version.
- The rest of these slides will illustrate how to contribute your code to the ChemDataExtractor project.

1. Contributing

Contributing



Contributing (1)

- The current unstable release is available from the **CambridgeMolecularEngineering** Github page.
- You will likely need to make many changes to ChemDataExtractor for it to work well with your particular use case.
- All new developers are given a branch of the **PRIVATE** development version.

Contributing (2) - Getting Started

- To get set up developing with ChemDataExtractor you will need to do the following:
  1. Create a student Github account with your @cam email address that provides you with ***FREE and UNLIMITED*** private repositories.
  2. Email me (**cc889**) with your name, CRSID and GitHub username.
- You will then gain access to the private GitHub repository and Slack group.

Contributing (3) - Creating a Development Branch

- Clone the chemdataextractor-development repository.
- Then create your new development branch.

Creating a New Branch

```
$ git checkout [your CRSID]
$ git push origin [your CRSID]
```

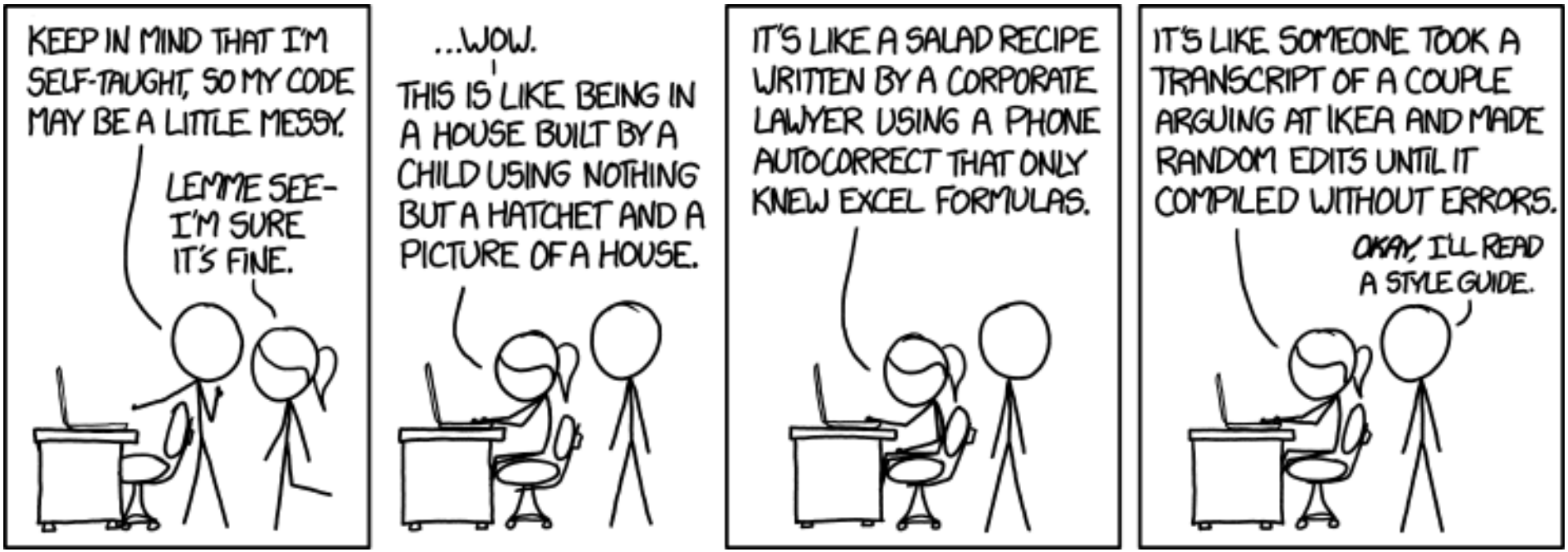
- You will then be able to edit, commit and push your source code as you please!

Pushing to Master

- You are encouraged to push your code to the master branch anytime you make **material** changes to the code.
- All new code must be fully documented and tested before it will be allowed to merge.
- To push your code, create a new pull request outlining your changes and these will be code reviewed.
- Major changes to the core of ChemDataExtractor should be discussed well in advance with Jacqui, Ed and myself.

2. Coding Standards

Coding Standards



Style Guidelines

- Please follow the **PEP8** style guide as closely as possible.
- The most important rule is ***BE CONSISTENT***, it makes reading and reviewing code easier for everyone involved.
- Some legacy code does not follow the guidelines, feel free to correct any offending code and submit a pull request with the changes.
- There are useful Linting tools, such as **autopep8**, that automatically format code to pre-defined guidelines.

Boilerplate

- At the start of every new file you create, please add the following boilerplate:

Boilerplate Example

```
# -*- coding: utf-8 -*-
"""
<file location> # e.g. chemdataextractor.my_package.my_file.py
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

<File description>. # e.g. Generic boilerplate example

<Author> (<email>). # e.g. Callum Court (cc889@cam.ac.uk)

"""
```

Docstrings

- At the very least, all functions should have a Docstring containing a description of its functionality, inputs, outputs and exceptions.

Docstring Example

```
def from_string(self, fstring, fname=None, readers=None):
        """Create a Document from a byte string containing the contents of a file.

        Usage::
            contents = open('paper.html', 'rb').read()
            doc = Document.from_string(contents)
        .. note::
            This method expects a byte string

        :param bytes fstring: A byte string containing the contents of a file.
        :param string fname: (Optional) The filename
        :param list readers: (Optional) List of readers to use.

        :raises: ReaderError: If specified readers are not found
        """
```

Comments

- When you make changes to pre-existing code, please add a comment containing your CRSID, this helps others to identify your changes in later versions.
- **TODO** items should also contain your CRSID.

Comments Example

```
...
# Added by cc889 (11/10/18)
# TODO(cc889): Test new functionality

def my_new_function(self, input):
        ...
```

3. Testing

Testing


PC WEENIES
YOU WILL BE PLEASED TO KNOW THAT MY CODE FINALLY PASSES YOUR STRINGENT UNIT TESTS, PAM.
REALLY? THAT'S GREAT, APCHUCK!
SO WHAT WAS YOUR DEBUGGING STRATEGY?
I MODIFIED YOUR UNIT TESTS.
WWW.PCWEENIES.COM
KRISHNA M. SADASIVAM

Testing Your Code

- Tests are small scripts written to ensure that a new bit of code is behaving as it should.
- It's very common for someone to add a few lines of code, only for it to unexpectedly break some functionality elsewhere in the code.
- The inclusion of automated tests helps you to find these cases early and drastically reduce the time spent troubleshooting.

Unit Tests (1)

- ChemDataExtactor Unit Tests are built using the python **unittest** package.
- Pre-existing tests are in the **src/tests/** directory
- For each new piece of functionality you create, you must also create a test class that contains logic for ensuring the code behaves as it should
- You may need to create a new test file if one does not already exist, this should be named **src/tests/test_[package]_[file].py**.

Unit Tests (2)

Unit Tests Example

```
import unittest
from lxml import etree

from chemdataextractor.doc.text import Sentence, Paragraph
from chemdataextractor.parse.mp import mp_phrase

class TestParseMp(unittest.TestCase):
    def test_mp1(self):

        # Declaration
        s = Sentence('Colorless solid; mp 77.2 −77.5 C.')
        expected = '<mp><value>77.2−77.5</value><units> C </units></mp>'

        # Testing
        result = next(mp_phrase.scan(s.tagged_tokens))[0]

        # Assertion
        self.assertEqual(expected, etree.tostring(result, encoding='unicode'))
```

Unit Tests (3)

- This is just one simple test of the function.
- You should include multiple different tests, with many different cases to prove that the function works as expected.
- **One single test of a simple case is not enough for your code to pass through the code review.**

Running Your Tests

- Run your unit tests using the **pytest** package, pointing it to a directory (or file) you wish to test.
- This will create a neat output showing wether or not the tests passed and where any failures occurred.

Using pytest

```
$ pip install pytest

$ pytest /tests
...
...

========================= 350/350 Tests Succeeded =========================
```

Tips

- **Golden Rule**: Put time in now, save time later.
- Where possible write tests in parallel with your code development.
- You might even enjoy it (!)

4. Advanced Topics

Advanced Topics

- The GitHub repository contains an **Introduction.md** file describing all this information, as well as covering advanced topics such as:
  1. IDE's
  2. Web-scraping
  3. Document Readers
  4. Writing Parsers
  5. Regular Expressions
- If you need help, please use the **Slack group** as a way to ask questions.

Summary

- Send email providing me with your CRSID and GitHub details
- Checkout your own development branch
- Edit to your hearts content!
- Ensure code is fully tested and documented before pushing to master

Happy Coding!

Any Questions?