Software development toolbox Documentation

Release 0.0

Radovan Bast

April 22, 2016

Contents

CHAPTER 1

Frequently asked questions

1.1 Network

There is access to Eduroam in the course room (in fact in large parts of the campus). If you do not have access to Eduroam you will get a temporary wireless access for the course week.

1.2 Microwaves for lunchboxes

There are several microwave possibilities with plates and forks and knives etc. in close vicinity of the course room. Please remember to wash your dishes or to put them into the dishwasher.

CHAPTER 2

Pre-study week

During the pre-study week you should acquire basics in at least one interpreted language in case you have never used one (we recommend Python or Julia) and basics in at least one compiled language in case you have never used one (we recommend C or C++ or Fortran). But if you prefer Ruby to Python then there is nothing wrong with that. And if you are interested in learning Haskell or Clojure or F# as contrast to "traditional" languages for scientific programs, then please do - this course is in principle language-independent.

The reason why we recommend to learn an interpreted and a compiled language during the pre-study week is that we will not teach programming languages but we will teach tools and work-flows for an efficient collaborative programming. We believe that developers who use compiled languages can benefit from interpreted languages and vice versa. Typically you will anyway end up using several programming languages.

We also recommend to study the basics of version control using Git. This will be treated during the lectures as well but it will help to already have a basic idea.

Here is some recommended material for the pre-study week:

Linux/unix shell

- http://swcarpentry.github.io/shell-novice/
- http://linuxcommand.org/tlcl.php
- http://www.tldp.org/LDP/abs/html/index.html
- http://mywiki.wooledge.org/BashGuide

Git

- http://swcarpentry.github.io/git-novice/
- https://git-scm.com/book/
- https://try.github.io
- http://pcottle.github.io/learnGitBranching/
- https://guides.github.com/introduction/flow/index.html

Python

- http://swcarpentry.github.io/python-novice-inflammation/
- http://hplgit.github.io/scipro-primer/slides/index.html
- https://docs.python.org/3/tutorial/
- http://docs.python-guide.org

New languages

- http://exercism.io

Make and CMake

- http://swcarpentry.github.io/make-novice/
- https://www.youtube.com/watch?v=TqjtN8NGtl4

Test-driven development

- http://katyhuff.github.io/python-testing/

C

- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-087-practical-programming-in-c-january-iap-2010/index.htm

Fortran

- http://www.csee.umbc.edu/~squire/fortranclass/summary.shtml
- http://www.fortran90.org/src/best-practices.html

Documentation

- http://jacobian.org/writing/great-documentation/

CHAPTER 3

Software that you should install prior to the course

All exercises will be done on your laptop. You can also run them on your remote desktop via your laptop. We will not have access to other (local) computing resources.

3.1 Alternative 1: You install the software directly on your laptop or remote desktop

Disclaimer: These instructions are written by a Linux user. Instructions for Windows and Mac OS X are untested (please submit corrections via pull requests).

We will need:

- Shell (bash or other shell that you like better)
- Text editor (vi or vim or emacs or nano or atom or your favourite editor; if you haven't used any of these before, pick the one you can exit without killing the terminal; you can exit vi and vim with ":q!", emacs with "CTRL-X CTRL-C", and nano with "CTRL-X")
- Python
- Python packages (Sphinx, Jupyter notebook, pytest); we recommend to install these either using Anaconda or using Virtualenv
- Git
- Compilers: gfortran, gcc, g++ (depending on whether you use Fortran or C or C++)
- GDB
- Make
- CMake: http://www.cmake.org
- Valgrind
- Meld or Diffuse

For Anaconda, please use the 2.7 version: https://www.continuum.io/downloads

If you prefer Virtualenv over Anaconda, please follow http://docs.python-guide.org/en/latest/dev/virtualenvs/. Note that you should not try to install both.

For Mac OS X we recommend installing packages via Homebrew: http://brew.sh (use `$ brew search <package>`). But if you like MacPorts better, that should work, too. If you are not a sudoer, Homebrew is a better option than MacPorts. Or so I heard.

On Linux we recommend to install cmake, gfortran, gcc, g++, and git via standard package installers (apt-get or yum or pacman or your favourite installer). Sphinx can be installed via standard package installers although in the long run it is convenient to install Python packages using Virtualenv.

For troubleshooting on Windows we recommend to use this good resource: https://github.com/swcarpentry/workshop-template/wiki/Configuration-Problems-and-Solutions.

3.2 How you can verify that the installation worked

[I really do not know how this looks on Windows]

Bash

```
$ bash --version
```

Should give you a version (like here) and not an error (don't worry if the version is different on your system):

```
GNU bash, version 4.3.42(1)-release (x86_64-unknown-linux-gnu)
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Text editor

Open a file and edit it. If this works, all is good.

Python

Open a Python shell. It should look like this (version might be different; Python 2 is good enough):

```
$ python

Python 3.5.1 (default, Dec  7 2015, 12:58:09)
[GCC 5.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You get out of it with CTRL-D.

Sphinx

```
$ sphinx-quickstart --version
```

Should produce (don't worry about exact version, just make sure you don't see an error):

```
Sphinx v1.3.4
```

Jupyter notebook

```
$ jupyter-notebook --version
```

Should produce (don't worry about exact version, just make sure you don't see an error):

```
4.1.0
```

pytest

```
$ py.test --version
```

Should produce (don't worry about exact version, just make sure you don't see an error):

```
This is pytest version 2.8.5, imported from /foo
```

Git

```
$ git --version
```

Should give you a version (like here) and not an error (don't worry if the version is different on your system):

```
git version 2.7.0
```

Before you start using any Git commands, We strongly suggest switching the global editor to the one you know how to exit. This should do the trick:

```
$ git config --global core.editor emacs # or vim or something else
```

GFortran

```
$ gfortran --version
```

Should give you a version (like here) and not an error (don't worry if the version is different on your system):

```
GNU Fortran (GCC) 5.3.0
Copyright (C) 2015 Free Software Foundation, Inc.

GNU Fortran comes with NO WARRANTY, to the extent permitted by law.
You may redistribute copies of GNU Fortran
under the terms of the GNU General Public License.
For more information about these matters, see the file named COPYING
```

GCC

Check output of `gcc --version`.

G++

Check output of `g++ --version`.

GDB

```
$ gdb --version
```

Should give you a version (like here) and not an error (don't worry if the version is different on your system):

```
GNU gdb (GDB) 7.10.1
Copyright (C) 2015 Free Software Foundation, Inc.
```

Make

```
$ make --version
```

Should give you a version (like here) and not an error (don't worry if the version is different on your system):

```
GNU Make 4.1
Built for x86_64-unknown-linux-gnu
Copyright (C) 1988-2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

CMake

```
$ cmake --version
```

Should give you a version (like here) and not an error (don't worry if the version is different on your system):

```
cmake version 3.4.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

Valgrind

```
$ valgrind --version
```

Should give you a version (like here) and not an error (don't worry if the version is different on your system):

```
valgrind-3.11.0
```

Meld or Diffuse

To test it create two files which are similar and then compare them with Meld or Diffuse:

```
$ meld file1 file2
```

3.3 Alternative 2: You code in the cloud (in your browser)

Use this fantastic service https://c9.io and create a workspace for this course. A workspace is an Ubuntu container via Docker in which you can edit files, install and run software.

You can install (almost) all the software we need with:

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install sphinx jupyter pytest
$ sudo apt-get install fortran cmake
```

CHAPTER 4

Timetable

4.1 Monday, Jan 25, 2016

- 10:00 - 10:30 Course overview and practical information (Radovan)
- 10:30 - 11:30 Complexity in software development (Jonas)
- 13:00 - 14:00 Functional programming (Jonas)
- 14:00 - 17:00 Working with Git 1/3 (Radovan) [Intro] [Branches]

4.2 Tuesday, Jan 26, 2016

- 09:30 - 11:00 Working with Git 2/3 (Radovan) [Conflicts] [Distributed]
- 11:00 - 12:00 Mixed Martial Arts (Jonas)
- 13:00 - 14:30 Working with Git 3/3 (Radovan) [GitHub] [Design] [Arch]
- 14:30 - 17:00 Exercise session (Git)

4.3 Wednesday, Jan 27, 2016

- 09:30 - 10:00 Working with Jupyter notebook (Radovan)
- 10:00 - 11:00 Profiling and code optimization (Radovan)
- 11:00 - 12:00 Modern code documentation (Radovan)
- 13:00 - 14:00 Test-driven development (Radovan)
- 14:00 - 17:00 Exercise session (TDD, profiling, and documentation)

4.4 Thursday, Jan 28, 2016

- 09:30 - 11:00 Building software with Make (Michael)
- 11:00 - 12:00 Debugging toolbox (Michael)
- 13:00 - 14:30 Building software with CMake (Radovan) [Basics] [Advanced]

- 14:30 - 17:00 Exercise session (Make, CMake and debugging)

4.5 Friday, Jan 29, 2016

- 10:00 - 11:00 Software licensing (Erik)
- 11:00 - 11:20 Real life example: Code development in DIRAC and Dalton (Radovan)
- 11:20 - 11:40 Real life example: Code review and continuous integration in GROMACS (Rossen)
- 11:40 - 12:00 Concluding remarks and practical information (Radovan)

CHAPTER 5

Exercises

5.1 Get comfortable with Git

You can work on the exercises "out of order" - in the order that is most interesting/relevant for you.

5.1.1 Basic init-add-commit workflow

Initialize an empty Git repository, add some source code or text and commit few changes. Use `git status` a lot. Test `git log`, `git grep`, `git diff`. Experiment with the staging area with `git add` and verify how `git diff` behaves with staged changes. Create files that you want ignored by Git. Make Git ignore these files. Create branches, switch between them, merge them, delete them.

5.1.2 Git branching game

Try to solve basic "Main" and "Remote" exercises in http://pcottle.github.io/learnGitBranching/. You decide how far you want to get and which topics are most relevant for your work.

5.1.3 Practice working with remotes (on a local machine)

- Create a normal Git repository on your laptop (repo A).
- Create, add, and commit a README file or an example source file or script.
- Clone it into a bare repository (repo B).
- Clone the bare into another non-bare repository (repo C), everything still on your computer.
- Have a look at `git remote -v` in repo C.
- Have a look at `git remote -v` in repo B.
- Have a look at `git remote -v` in repo A.
- Add the bare repo B as remote in A.
- Exercise communicating changes between the two non-bare clones (A and C).
- Verify that `origin` is just a label by pushing directly to the full path.
- Create a GitHub project (without auto-creating `README`, `LICENSE`, or `.gitignore`).

- Change `origin` to now point to GitHub and push the entire `master` branch from one our your local repos into it.

5.1.4 Collaborative GitHub workflow

https://github.com/bast/forking-workflow-exercise

5.1.5 Git bisect exercise

https://github.com/bast/bisect-me

5.1.6 Rebasing and squashing commits

https://github.com/bast/git-rebase-squash-exercise

5.1.7 Bonus exercise

Reimplement `git clone` using shell scripting or using your favourite language.

5.2 Test-driven development

We will do this exercise in pairs. Not only we will learn how to do test-driven development, but we will also exercise collaborative GitHub workflow, and get to know two great services for automated testing (Travis) and code coverage analysis (Coveralls).

First find a partner who speaks the same programming language as you. Then proceed as follows:

- Create a GitHub project for this exercise (both of you create one).
- Sign in to https://travis-ci.org and https://coveralls.io with your GitHub account and enable there your new GitHub project.
- Create two or three unit tests for functions which do not exist yet.
- Do not implement the functions, only their tests and stubs of the functions.

Example (Python; the function get_word_lengths currently fails):

```
def get_word_lengths(s):
    """
    Returns a list of integers representing
    the word lengths in string s.
    """
    return None


def test_get_word_lengths():
    text = "Three tomatoes are walking down the street"
    assert get_word_lengths(text) == [5, 8, 3, 7, 4, 3, 6]
```

Then:

- Check that the test fails (since the function is not implemented/finished).

- Commit the function and its test.
- Create a .travis.yml file based on provided examples (below) and commit it.
- Push the tests and function stubs to GitHub and verify that the tests fail on Travis.

Now your programming partner forks your repository and you fork hers/his. Then:

- Fix the function/routine until the test(s) pass(es).
- Commit and push the working function/routine.
- Check and discuss the test history on https://travis-ci.org.
- Check and discuss the test coverage on https://coveralls.io.
- Iterate and refine.

When you are finished, submit your work as pull request and your programming partner will review and possibly accept the changes.

Examples that you can use as a starting point:

- Python example using pytest
- C/C++ example using Google Test
- Fortran example using pFUnit

5.3 Modern code documentation

5.3.1 Part 1: Sphinx-based documentation on Read the Docs

In this exercise we will implement a Sphinx-based documentation, host it on GitHub and deploy it to https://readthedocs.org. This is exactly how this page that you read right now arrives to your browser (the sources are here: https://github.com/bast/software-development-toolbox).

- Set up a virtual environment according to http://docs.python-guide.org/en/latest/dev/virtualenvs/.
- Install Sphinx to the virtual environment.
- Run `sphinx-quickstart` (http://sphinx-doc.org/tutorial.html).
- Build the html and check it locally on your computer and in your browser.
- Make some changes to it and build them locally.
- Create a new GitHub project for it.
- Push the documentation sources to the new GitHub project.
- Create an account at https://readthedocs.org.
- Import the Github project you just created to Read the Docs.
- Create a post commit hook in GitHub so that changes automatically refresh the Read the Docs pages.
- Test the post commit hook by making and pushing changes to the documentation sources and verify that the documentation refreshes after your changes.

5.3.2 Part 2: Create an example project website and host it on GitHub Pages

Create an example project website (from GitHub Pages templates or on your own) and host it on GitHub Pages (https://pages.github.com). If you use Doxygen, try to host Doxygen-generated documentation on GitHub Pages.

5.4 CMake

5.4.1 Create a CMake framework for a project and practice debugging with Valgrind

In this exercise we will CMake-ify a project. This is interesting for people who use Makefiles or Autotools.

You can use the exercise time to practice CMake on your own project(s) but we also provide a mockup project: https://github.com/juselius/vat-69.git

Your task is to:

- **Create a build system using CMake:**
  - Build a shared library
  - Build and link the main program
  - Create an installer so the program can be installed properly (GNU standards)
  - Compile a parallel version with OpenMP
- Find all bugs using Valgrind and fix them
- Find all parallelization bugs using Helgrind (part of Valgrind; this exercise point will only work with the Intel compiler; skip this when using GNU)

5.4.2 Create a small CMake-built project (C or C++ or Fortran)

Define the project version as a CMake variable. In this project try to get the configure-time Git hash and the project version into the output of the code.

CHAPTER 6

Project week

During the project week you will use what you have learned during the lectures and hands-on exercises.

Ideally this will be to introduce version control, testing, documentation framework, good practices and good intentions for your own project(s). If you use makefiles then the project can also involve moving to CMake (we highly recommend such a step).

You should choose and submit a short project abstract (one or two paragraphs) in pdf format before the end of day 4 so that we have the possibility to discuss the projects on day 5.

The project work concludes with a report summarizing the work done. You don't have to write a novel, 3-5 pages should do it. Describe the situation before and after. Describe what you did and how everything works together in your project(s). It is OK to write this report online - then other developers have the possibility to follow your good example and you also document routines and workflows for new developers that want to join your project(s).