Introducing Software Carpentry

Alan O'Cais, JSC
April 2018

software carpentry

JÜLICH
FORSCHUNGSZENTRUM

The Mission

The Carpentries make researchers in science, engineering, and medicine more productive by teaching them basic lab skills for scientific computing

The Problem

- Scientists spend more and more time building and using software
- *Most are primarily self-taught*
- Hard to measure how *well* they do things
- But anecdotal evidence suggests "not very"

The Carpentries Solution

- Scientists teaching scientists
- Two days of hands-on learning

the Unix shell	⇒ automate repetitive tasks
Git and GitHub	⇒ track and share work
Python or R	⇒ build modular code
SQL	⇒ manage data

Advertise the tool, teach the thinking

Why Workshops?

- Scientists don't know what questions to ask
- Or how to recognize a useful answer when they find one
- Most online tutorials are aimed at commercial developers, not researchers
- **And many focus on HPC but ignore pre-requisite skills**

Outcomes

- 10-20% improvement in productivity is common
- 10X isn't rare
- Do the old things faster
- Tackle new problems
- Ready for HPC, the cloud, big data, ...
- Start *doing* open science

The Details

- Materials are all open access
- Instructors are volunteers
- Host site pays for instructor travel and accommodation
- Administrative fee to cover central costs if we're helping organize

Principles of Computational Thinking

software carpentry

JÜLICH
FORSCHUNGSZENTRUM

Seven Big Ideas

1. It's all data.
2. Data is meaningless without interpretation.
3. Programming is about creating abstractions.
4. Models for computers, views for people.
5. Paranoia makes us productive.
6. Algorithms beat hardware.
7. The tool shapes the hand.

Principle #1

It's All Data

- Papers, observations, and images are all stored as 1's and 0's.
- Source code is just text files
    - So it can be manipulated like text.
- A program in memory is just bytes
    - Manipulating those bytes is no different from manipulating characters or pixels.

Principle #2

Data Is Meaningless Without Interpretation

- 01100100011000010111100001100001 is:
  - the word "data"
  - or the integer 1,684,108,385
  - or the number 1.6635613602263159e+22
  - or a bluish-gray pixel that's slightly transparent
  - Et cetera
- Because *computer don't understand: they obey*

Principle #3

Programming Is About Creating Abstractions

- Short-term memory can only hold 7±2 item.
- Must put details into groups (of groups...)
- Most features of programming languages exist to help do this.

Principle #3

Programming Is About Creating Abstractions

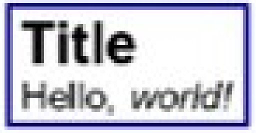
1. Separate *interface* (what something does) from *implementation* (how it works).
2. Value *clarity* over *cleverness*.

Principle #4

Models For Computers, Views For People

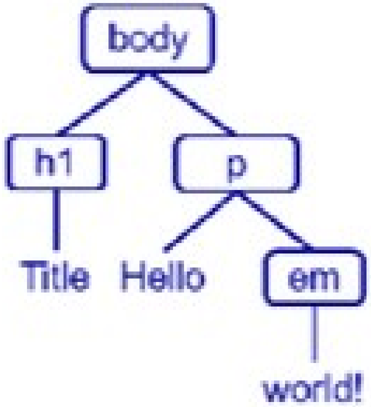
- A *model* is a representation that is easy for a computer to operate on.
- A *view* is a display that people can understand.
- Store models - show views.

Principle #4



Title

Hello, *world!*



```
<body>
<h1>Title</h1>
<p>Hello, <em>world!</em></p>
</body>
```

Principle #5

Paranoia makes us productive.

- Best way to improve productivity is to improve quality.
- Write tests to clarify meaning as well as to catch errors.
- Automate, automate, automate.

Principle #6

Algorithms beat hardware.

Data Size	O(log n)	O(n)	O(n^2)	O(2^n)
1	1	1	1	1
10	2.3	10	100	1024
100	4.5	100	10,000	1.26×10^{30}

Principle #7

The tool shapes the hand.

- Knowing how tools work gives you new ideas about how to use them.
- And about what new tools you could create.

Software Carpentry: Lessons Learned

software carpentry

JÜLICH

FORSCHUNGSZENTRUM

Not an Overnight Success

Los Alamos National Laboratory, July 1998

If you were born then, you can drive now.

Why We Exist

HPC, the cloud, big data

the other 92%

Lesson #1

Most researchers think programming is a tax they have to pay to do science.

"If I wanted to be a computer scientist,
I would have picked a different major in undergrad."

Lesson #2

They don't care about reproducibility.

- Five million papers published 1990–2000.
- 100 retracted for computational reasons.
- So odds of retraction = 1 in 50,000.
- Average paper takes eight months to produce.
- Reproducibility worth *115 seconds per paper*.

Lesson #3

They care *a lot* about productivity.

- And about being able to tackle new problems.
- And about their careers.

Lesson #4

The curriculum is full.

- "What do I drop to make room for more computing: quantum or thermo?"
- 5 minutes per lecture ⇒ 4 courses in a degree
- Have to fit in *around* the curriculum until we achieve critical mass

What Winning Looks Like

# Reviewers	% Papers
2	10%
3	40%
4	40%
5	10%
P(at least one reviewer is a believer)	50%
P(single reviewer is a believer)	18.3%

We only have to change the mind of 1 scientist in 5

Lesson #5

It's all in the details.

Two days
Live coding
Group signup

Charge a fee
Sticky notes
Peer instructors

Lesson #6

Incentives, incentives, incentives.

Save the world

Self-defense

Make new friends

Teach to learn

Boost their careers

Lesson #7

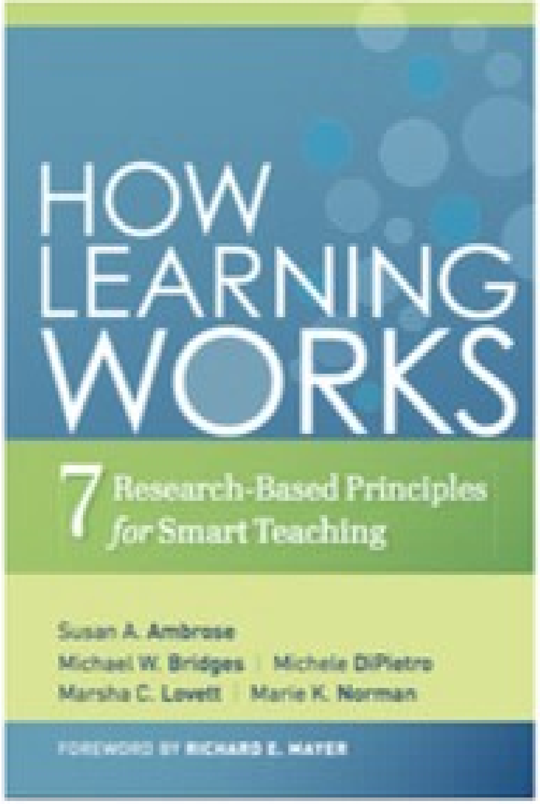
There's a lot we don't know.

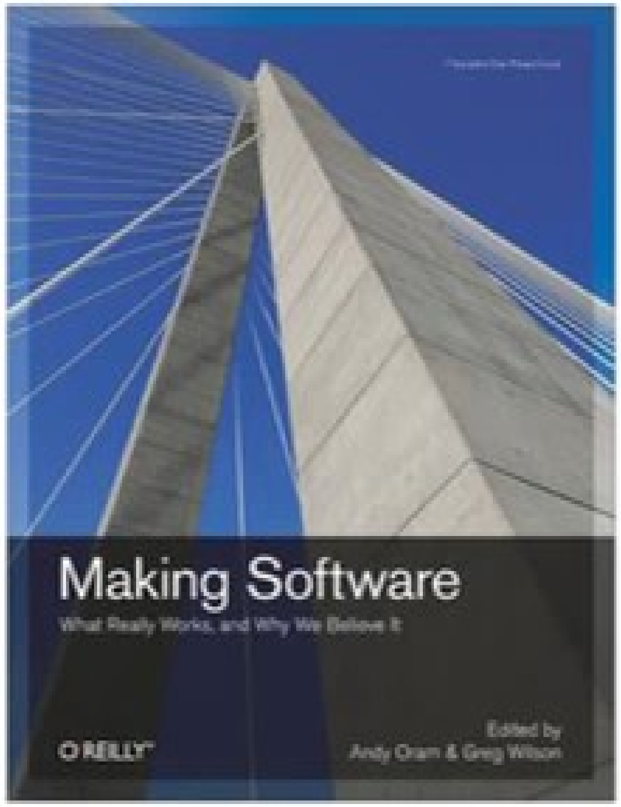
- How to measure programmers' productivity?
- How to measure scientists' productivity?
- The unknowns *don't* cancel out

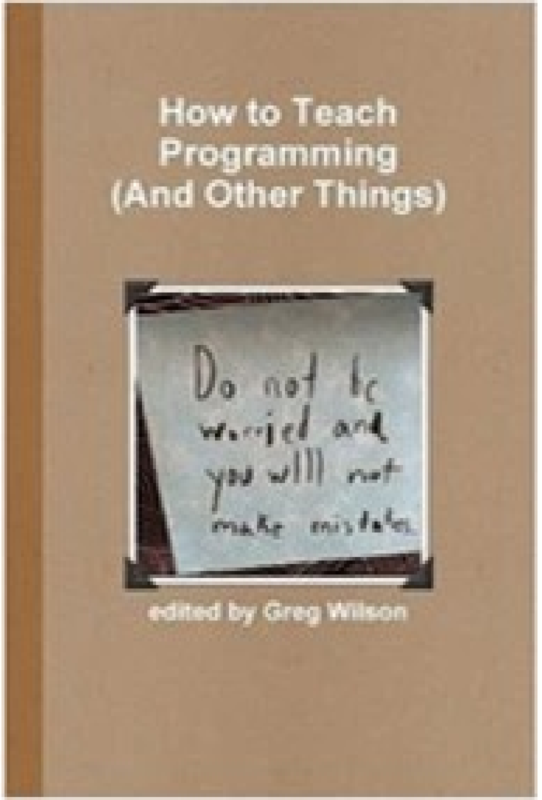
Our biggest failing is lack of systematic assessment

Lesson #8

There's a lot we do know.


HOW LEARNING WORKS
7 Research-Based Principles for Smart Teaching
Susan A. Ambrose
Michael W. Bridges | Michele DiPietro
Marsha C. Lovett | Marie K. Norman
FOREWORD BY RICHARD E. MAYER




How to Teach Programming (And Other Things)
Do not be worried and you will not make mistakes
edited by Greg Wilson

For Example

- Subgoals improve performance
- Practice works best for facts, worked examples for skills
- Peer instruction beats lecture
- Media-first increases retention

Read http://computinged.wordpress.com/

Lesson #9

Most people would rather fail than change.

Most scientists treat research on teaching and programming like most politicians treat research on climate change.

Lesson #10

Open isn't just for science.

- Our lessons have had over 150 contributors
- We can write them the way we write software and encyclopedias

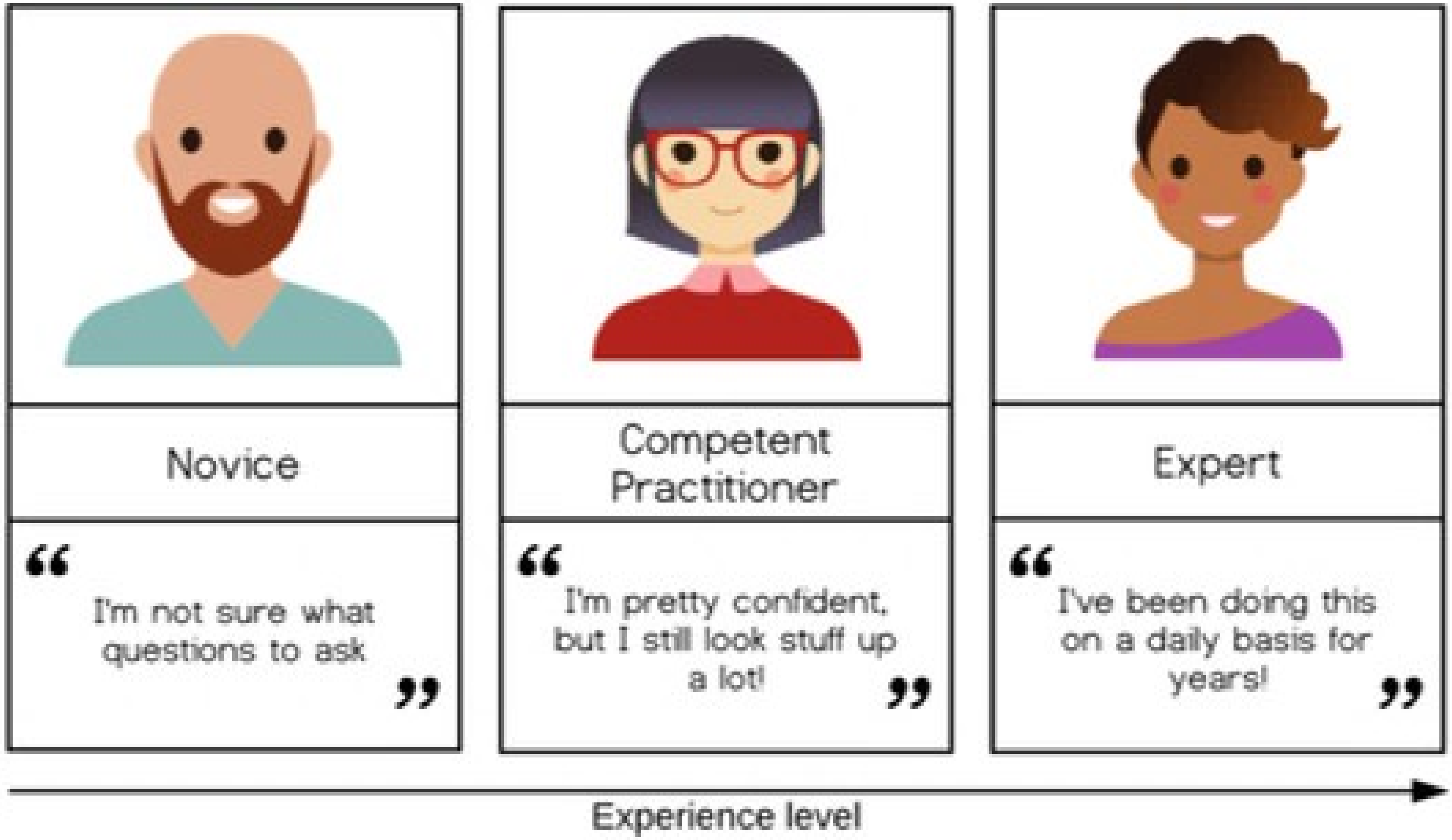
Open collaboration is the real revolution.

What can we take from this: *Teaching Tips*

software carpentry

JÜLICH
FORSCHUNGSZENTRUM

Skill levels



Novice

“ I'm not sure what questions to ask ”

Competent Practitioner

“ I'm pretty confident, but I still look stuff up a lot! ”

Expert

“ I've been doing this on a daily basis for years! ”

Experience level

Who Teaches and Why?

- Carpentries instructors are scientists in many career stages, from many fields
- Most are not computer scientists or software engineers
- "Conscious competence": still new enough to relate to beginners
- Many follow learner → helper → instructor path

Who Teaches and Why?

- Carpentries instructors are scientists in many career stages, from many fields
- Most are not computer scientists or software engineers
- "Conscious competence": still new enough to relate to beginners
- Many follow learner → helper → instructor path

Motivation

Motivation is the best predictor of learning.

1. Explain how these skills help your own research
2. Have learners sign up in groups
3. Follow learners' questions off the lesson
4. Have helpers give individual assistance

Demotivation

Avoid crushing their enthusiasm.

1. Enforce the code of conduct.
2. Avoid the passive dismissive "just".
3. Avoid cognitive overload.
4. If a problem can't be fixed quickly, have the person pair up.

Active Learning

Active learning beats passive observation.

1. Have learners type along as you teach.
2. Don't go more than 10 minutes without hands-on work.

Feedback

Everyone needs to know where they are.

1. Get real-time feedback ("OK/not OK") via colored sticky notes.
2. Get short written feedback ("minute cards") at every break.
3. *Respond to the feedback* even (especially) if it means teaching less.

Live Coding

No slides.

1. Start with a blank window — just like they will.
2. Having to type stops you from racing ahead.
3. Seeing you make mistakes gives them permission to.
4. Seeing you diagnose and fix mistakes shows them how to.

Pacing

People can't concentrate for more than an hour.

- Each major topic is 4 or 5 half-hour chunks over half a day.
- Get them out of their seats at each break.

Helpers

Never teach alone.

1. Former learners / local volunteers / the other instructor(s).
2. Help learners with setup and challenges, take notes on Etherpad, ...
3. Provide feedback to instructors.

Collaboration

Never learn alone.

1. Pair early, pair often.
2. Use Etherpad for note-taking and chat.
3. Use Git if/when learners are comfortable with it.

Assessment

Know your audience.

1. Pre-workshop survey drives workshop planning.
2. Challenges during workshops for formative assessment.
3. Post-workshop survey of learners...
4. ...and debriefing for instructors.

Instructor Training

Two-day class with the following overall goals:

1. Introduce evidence-based best-practices of teaching.
2. Teach you how to create a positive environment for learners.
3. Provide opportunities for you to practice and build your teaching skills.
4. Help you become integrated into the community.
5. Prepare you to use these teaching skills in workshops.

What can we take from this:
Hosting a Workshop

software carpentry

JÜLICH
FORSCHUNGSZENTRUM

Format

- 40 learners + 2 instructors per room
- Plus as many (local) helpers as possible
- Two (or more) rooms in parallel lets us stream people by prior experience
- Dinner-style seating, good WiFi, lots of power plugs, *unlocked* washrooms, ...

Flying Solo

- All materials are open access including a workshop template
- You can run a workshop on your own whenever you want without a fee
- Or use materials in other courses
- Must have at least one certified instructor and cover core topics to use the Carpentries name and logo
    - See FAQ for details

What can we take from this: *Creating a lesson*

software carpentry

JÜLICH
FORSCHUNGSZENTRUM

What

Create a new lesson for Software Carpentry

https://github.com/swcarpentry/styles

https://github.com/swcarpentry/lesson-example

Why a Template?

1. Simplify contribution
2. Ensure uniform appearance and metadata

How

1. Use GitHub Import to create a new repository with material from styles
2. Clone to desktop
3. Edit according to rules in lesson-example
4. Check

Why?

- Authors may work on many lessons
- But a user can only fork a repo once on GitHub

Template vs. Example

- styles has CSS, tools, etc. that may be updated centrally
- lesson-example is explanations that *shouldn't* be merged into lessons over and over again
- Separate repositories are less confusing than two "main" branches in one repository

1. Create Repository

- *Not* a fork of styles
- Use GitHub Import with https://github.com/swcarpentry/styles as the source URL
- Name it topic-level or topic-level-something
  - E.g., shell-novice
  - Or python-novice-inflammation
- Can be owned by anyone

2. Clone to Desktop

```
$ git clone -b gh-pages git@github.com:user/some-lesson.git
```

- Make sure you're *not* already in a Git repository
- `-b gh-pages` to put it in the `gh-pages` branch

3. Edit and Check

- See the README for general instructions
  - And lesson layout for details
  - There are notes on design
  - And an FAQ (additions welcome)

What's in the Template?

https://github.com/swcarpentry/styles

- Page layout templates
- CSS and images
- *Validation tool*
    - Please run this before pushing changes

What's in the Example?

https://github.com/swcarpentry/lesson-example

- Example lesson files (home page, topics, etc.)
- Description of required files and formatting rules

Updating

- We occasionally update the CSS, icons, etc.

```
$ git remote add template
https://github.com/swcarpentry/styles.git
$ git pull template gh-pages
```

- Call the remote template rather than upstream

Source Formats

- IPython Notebooks are difficult to diff and merge
- Other formats (e.g., reStructured Text) are only used by one community

Template Contents

Sub-directories for formatting

- _layouts: page templates
- _includes: included HTML snippets
- Named to be consistent with workshop-template

Required Files

- index.md: lesson's home page
- discussion.md: general discussion and pointers
- instructors.md: instructor's guide
- reference.md: reference guide for learners

Required Files

nn-topic.md: topics within lesson

- E.g., 01-select.md
- Each should be 10-15 minutes long

These are for instructors and offline reference, ***not*** *to be shown to learners during teaching*

Required Files

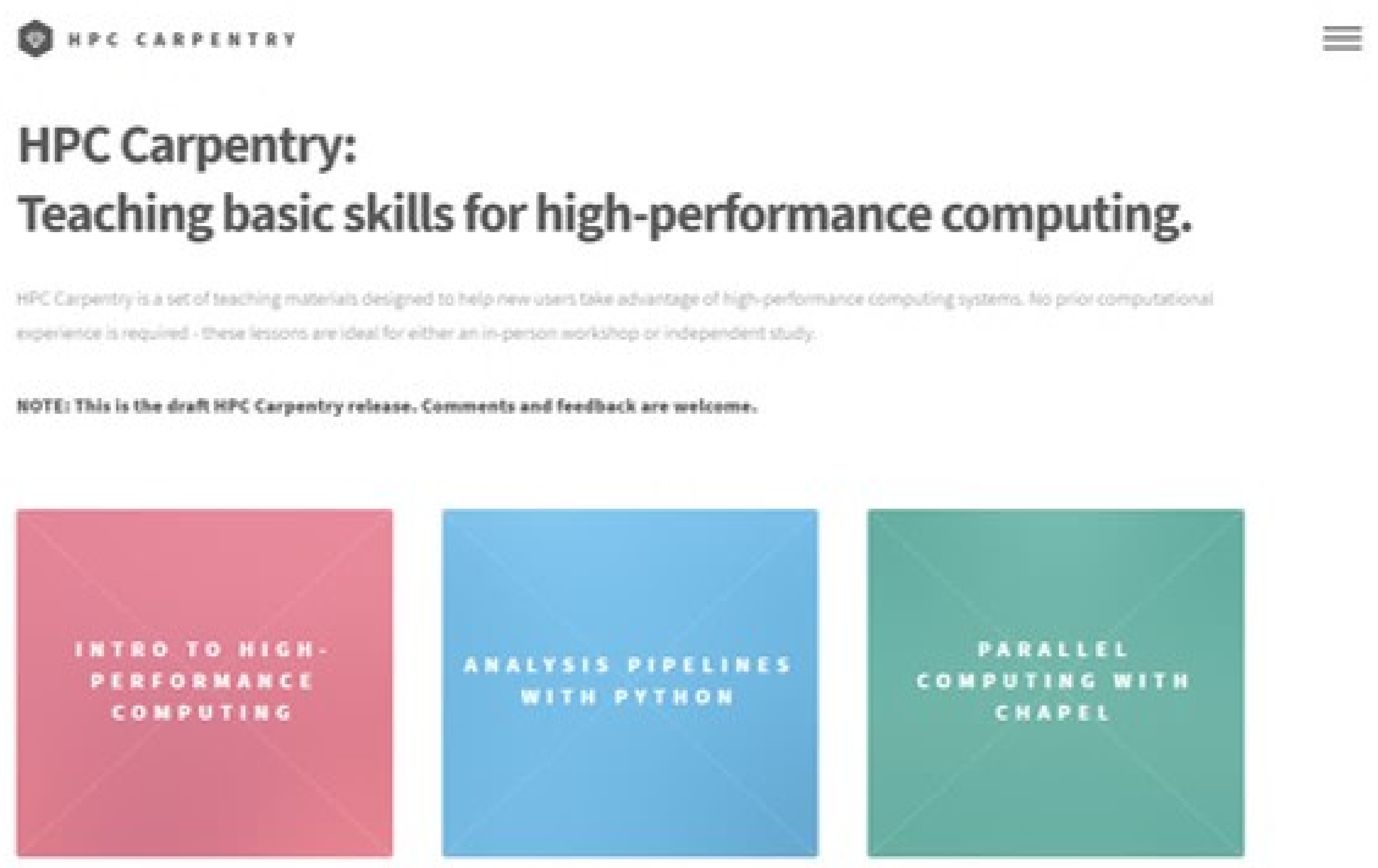
Sub-directories for lesson files

- code: source code
- data: data files
- fig: figures

Existing Lessons

- See http://software-carpentry.org/lessons.html
- Please let them know when you start to work on another one so we can advertise it

HPC Carpentry



HPC Carpentry:
Teaching basic skills for high-performance computing.

HPC Carpentry is a set of teaching materials designed to help new users take advantage of high-performance computing systems. No prior computational experience is required - these lessons are ideal for either an in-person workshop or independent study.

NOTE: This is the draft HPC Carpentry release. Comments and feedback are welcome.

INTRO TO HIGH-PERFORMANCE COMPUTING

ANALYSIS PIPELINES WITH PYTHON

PARALLEL COMPUTING WITH CHAPEL

HPC Carpentry

What has been done?

- Two HPC novice lessons in the wild
- BoF session at SC17

Site specifics in HPC space make lesson collaboration that much harder

CSA could look at application-specific lessons for HPC

Thank you for listening!

30 May - 1 June, 2018, University College, Dublin

CARPENTRYCON 2018

Building Locally, Connecting Globally

This slide deck is based on
https://github.com/swcarpentry/slideshows