Jupyter notebooks - A tool to write and share executable notebooks and data visualization

The goal of this lesson is to teach learners the user interface of JupyterLab, how Jupyter notebooks work, and what some common and powerful use cases are.

For many learners, notebooks are not a new concept, and our goal will be to demonstrate and discuss and guide towards good practices for **reproducibility, collaboration, and reusability**.

⚙ Prerequisites

- A reasonably recent version of Jupyter and JupyterLab, which is installed as part of the Conda environment for CodeRefinery workshops.
- Optional (we only demonstrate this): A reasonably recent version of Git to be able to work with the JupyterLab Git integration.
- If you wish to follow in the terminal and are new to the command line, we recorded a short shell crash course.

Jupyter Notebooks

❗ Objectives

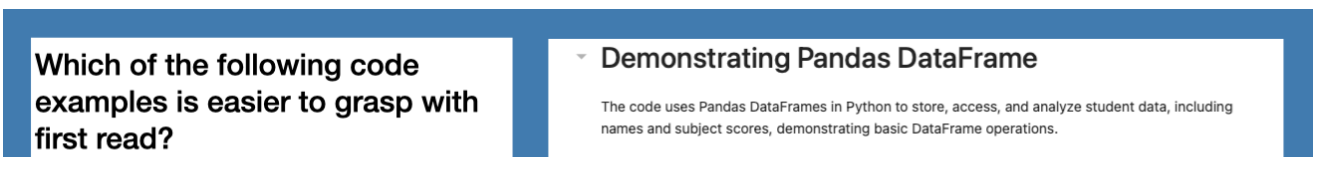
- Get an idea of the purpose of Jupyter.
- See some inspirational Jupyter Notebooks.

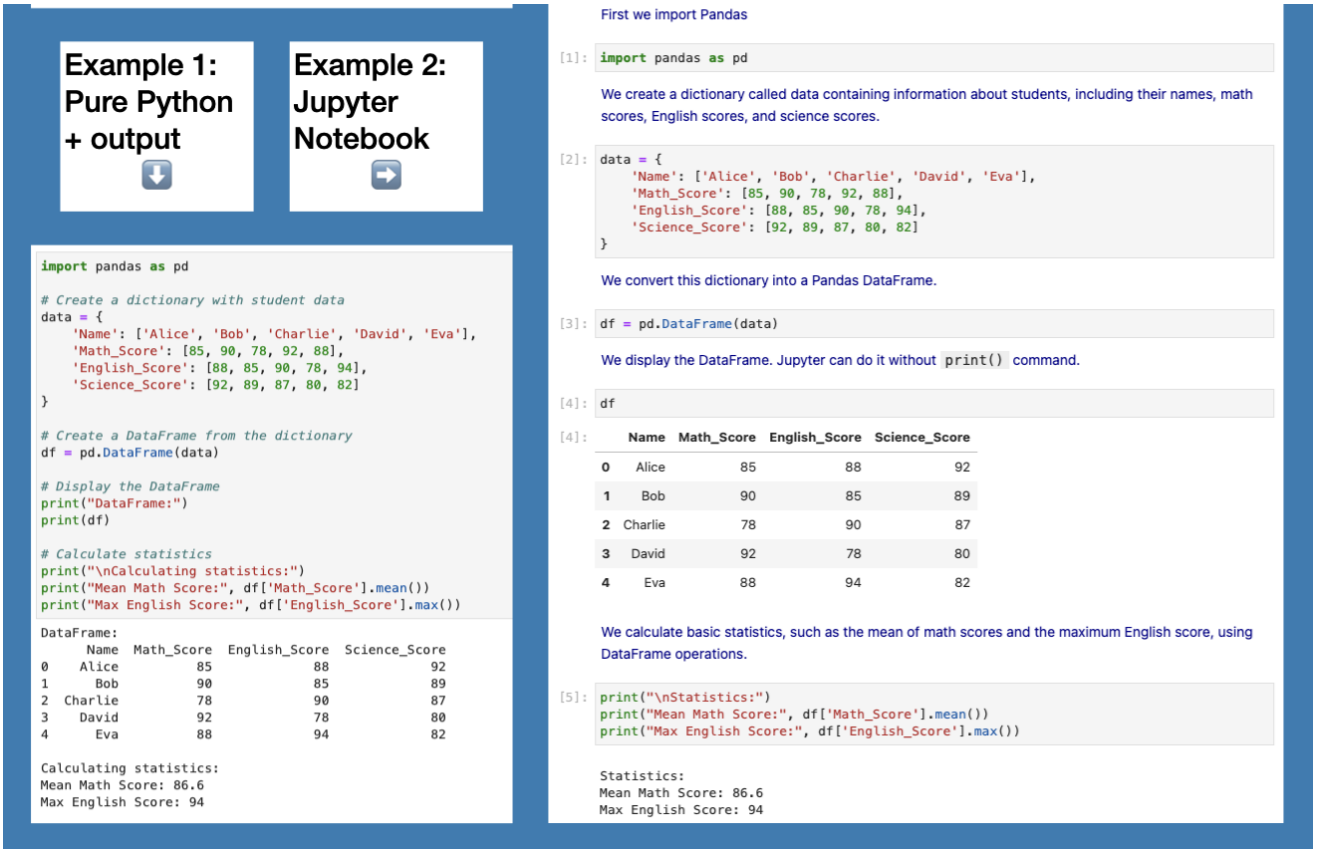
Instructor note

- 10 min teaching
- 0 min exercises

Motivation for Jupyter Notebooks

Jupyter Notebook is a tool for creating and sharing documents that contain live code, equations, visualizations, and narrative text.





Jupyter Notebooks can contain much more than just regular code. (CC-BY).

- **Code, text, equations, figures, plots**, etc. are interleaved, creating a *computational narrative*.
- *"an environment in which users execute code, see what happens, modify and repeat in a kind of iterative conversation between researcher and data"*
- The name "Jupyter" derives from Julia+Python+R, but today Jupyter kernels exist for dozens of programming languages.

Jupyter Notebook-ish functionalities are widely used

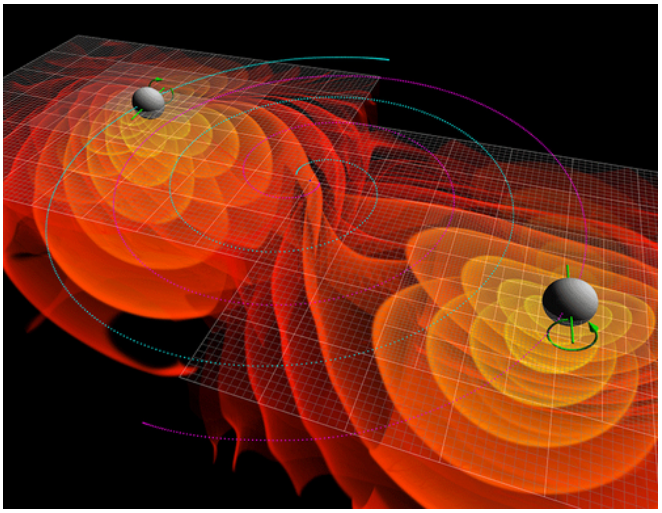
Many tools can have Jupyter-like behaviour: code and markdown cells

- VSCode
- Google Colab
- GitHub Codespaces

Case examples

Jupyter Notebooks make it feasible to share your code: you can explain your ideas and anyone can run the code eg. in Binder.

Gravitational wave discovery

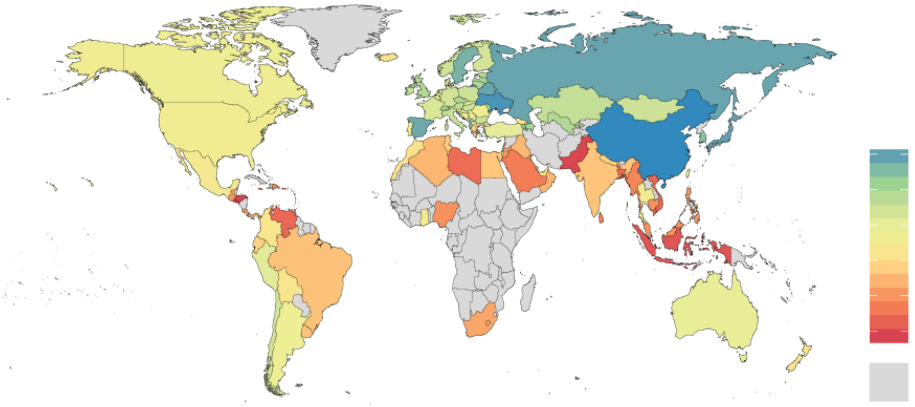


Discovery of gravitational waves.

As a case example, let us have a look at the analysis published together with the discovery of gravitational waves. This page lists the available analyses and presents several options to browse them.

- A quick look at short segments of data can be found at https://github.com/losc-tutorial/quickview
- The notebook can be opened and interactively explored using Binder by clicking the "launch Binder" button.
- How does the Binder instance know which Python packages to load? It takes the information from files like `requirements.txt` (Python) or `environment.yml` or `runtime.txt` (R).

Activity inequality study



Stanford Activity Inequality Study.

Researchers in the Stanford Activity Inequality Study measured daily activity from cell phone tracking data for over 700,000 users in different countries across the world.

- All data and notebooks are available at https://github.com/timalthoff/activityinequality
- Even without a "launch binder" button, the notebooks can still be launched on Binder (you may see an error "missing R kernel" because a file `runtime.txt` is missing - more about that later)

- Do you see any potential problems in recreating e.g. fig3bc?

More examples

For further inspiration, head over to the Gallery of interesting Jupyter Notebooks.

Use cases

- Really good for **linear workflows** (e.g. read data, filter data, do some statistics, plot the results)
- Experimenting with new ideas, testing new libraries/databases
- As an *interactive* development environment for code, data analysis, and visualization
- Interactive work on HPC clusters
- Sharing and explaining code to colleagues
- Teaching (programming, experimental/theoretical science)
- Learning from other notebooks
- Keeping track of interactive sessions, like a **digital lab notebook**
- **Supplementary information with published articles**
- Slide presentations using Reveal.js

Pitfalls

- Programs with non-linear code flow
- Large codebase (however it can make sense to use Jupyter as interface to the large codebase and import the codebase as a module)
- You cannot easily write a notebook directly in your text editor (but you can do that with R Markdown)
- Notebooks can be **version controlled** (nbdime helps with that), but there are **still limitations**.
- Notebooks **aren't named by default** and tend to **acquire a bunch of unrelated stuff**. Be careful with organization!
- See also https://scicomp.aalto.fi/scicomp/jupyter-pitfalls/.

Good practices

- Rename notebooks from "Untitled.ipynb".
- Run all cells before sharing/saving to verify that the results you see on your computer were not due to cells being run out of order (we will try this later).

JupyterLab and notebook interface

Objectives

- Learn to navigate JupyterLab user interface.
- Discuss integrated development environments.
- Get an overview of useful keyboard shortcuts.

- Learn about command/edit modes and markdown/code cells.

Instructor note

- 15 min teaching
- 0 min exercises

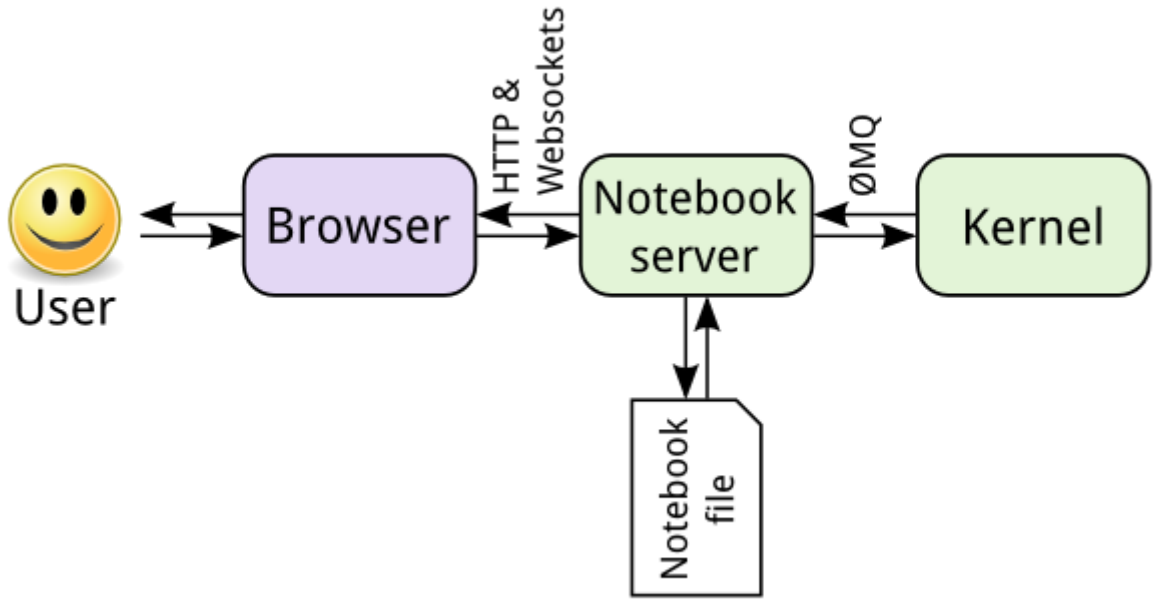
JupyterLab is the next-generation user interface for Jupyter Notebooks and is intended to replace the conventional interface. It is a highly modular and customizable interface.

How to start JupyterLab

Let's have a look at how it works. We go to terminal, and type:

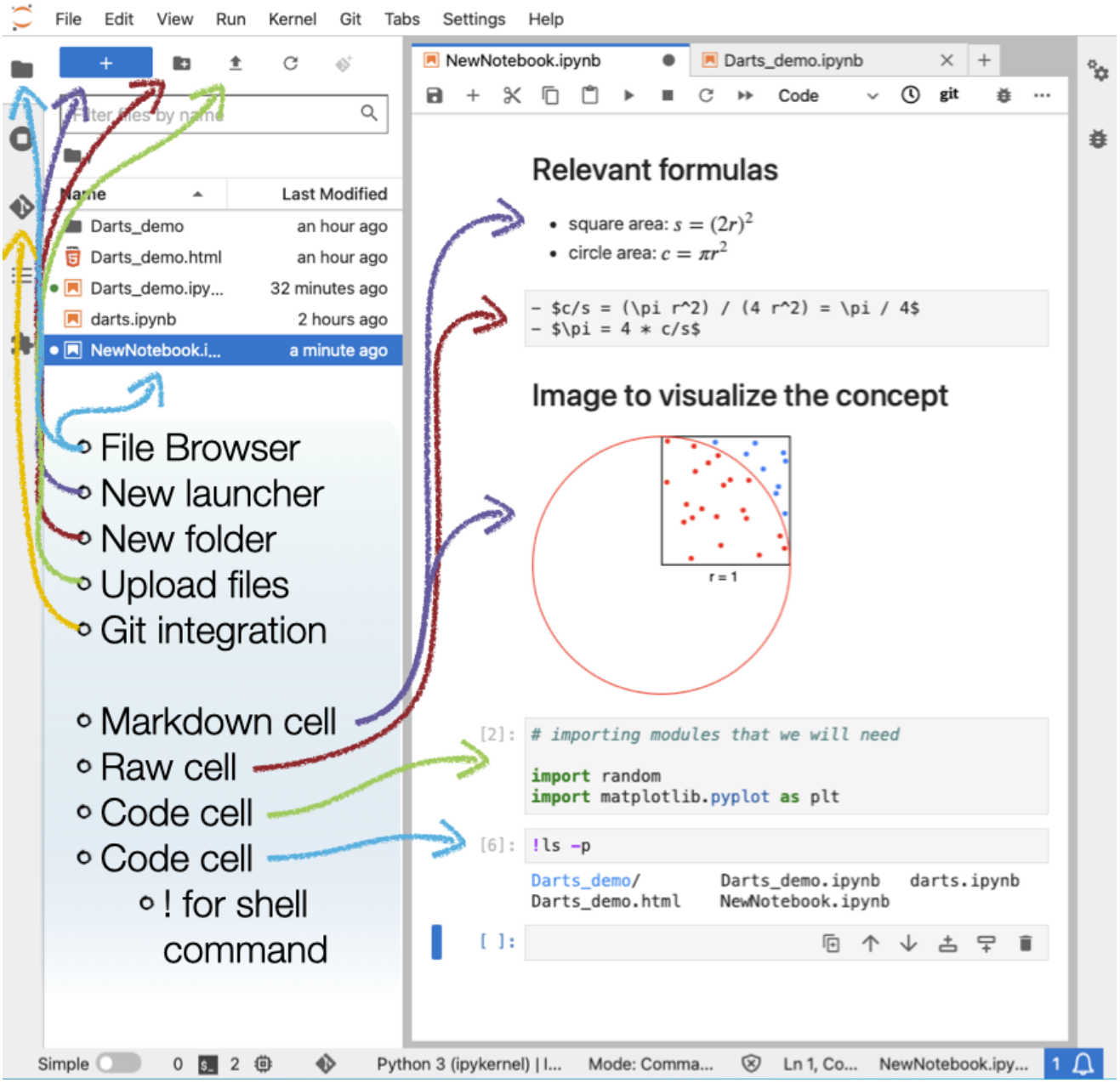
```
$ mkdir jupyterlab-demo
$ cd jupyterlab-demo
$ jupyter-lab
```

Components: the big picture



Components of a Jupyter notebook.

Navigation



Jupyter Lab user interface has a left side toolbar – that has a file browser and a git tab. The file browser shows the file that Jupyter Lab was opened in. The right side shows the files that are open. You can have split view by dragging from the top tab view. (CC-BY)

- Left-hand menu (toggle it with `Ctrl(⌘)-b`):
  - File browser
    - New launcher
    - New folder
    - Upload files
  - Running terminals and kernels
  - Command palette
  - Cell inspector
  - Open tabs
  - Git integration (if `jupyterlab-git` extension installed)
  - GitHub browser (if `jupyterlab-github` extension installed)
- Fully-fledged terminal

- Text editor for source code in different languages
- Code console to run code interactively in a kernel with rich output and linear order
- Modular interface
  - Notebooks, terminals, consoles, data files etc can be moved around
- Classical notebook style is available under the Help menu

Cells

- **Markdown cells** contain formatted text written in Markdown
- **Code cells** contain code to be interpreted by the *kernel* (Python, R, Julia, Octave/Matlab...)

Markdown cells

```
## Second level heading

This cell contains simple
[markdown](https://daringfireball.net/projects/markdown/syntax), a simple language for
writing text that can be automatically converted to other formats, e.g. HTML, LaTeX or
any of a number of others.

**Bold**, *italics*, **_combined_**, ~~strikethrough~~, `inline code`.

* bullet points

or

1. numbered
3. lists

**Equations:**
inline $e^{i\pi} + 1 = 0$
or on new line
$$e^{i\pi} + 1 = 0$$

Images:

![Jupyter logo](https://jupyter.org/assets/homepage/main-logo.svg)

Links:
[One of many markdown cheat-sheets](https://github.com/adam-p/markdown-
here/wiki/Markdown-Cheatsheet#emphasis)
```

Code cells

```
# a code cell can run statements of code.
# when you run this cell, the output is sent
# from the web page to a back-end process, run
# and the results are displayed to you
print("hello world")
```

Command and edit modes

- To add contents to a cell, you need to enter *edit mode* by pressing `Enter` or double-clicking on a cell
- To navigate between cells, create new cells, etc., you need to enter *command* mode by pressing `Escape` key or executing the current cell.

Keyboard shortcuts

Some shortcuts only work in Command or Edit mode. It can also happen that these shortcuts interfere with browser shortcuts.

Cell shortcuts

Shortcut	Effect
`Enter`	Enter Edit mode
`Escape` or `Ctrl` - `m`	Enter Command mode
`Ctrl` - `Enter`	Run the cell
`Shift` - `Enter`	Run the cell and select the cell below
`Alt` - `Enter`	Run the cell and insert a new cell below
`m` and `y`	Toggle between Markdown and Code cells
`d-d`	Delete a cell
`z`	Undo deleting
`a/b`	Insert cells above/below current cell
`x/c/v`	Cut/copy/paste cells
`Up/Down` or `k/j`	Select previous/next cells

Notebook shortcuts

Shortcut	Effect
`Ctrl(⌘)` - `s`	Save notebook
`Shift` - `Ctrl(⌘)` - `s`	Save notebook as
`Ctrl` -q	Close notebook
`Ctrl(⌘)` - `b`	Toggle left-hand menu
`Shift` - `Ctrl(⌘)` - `c`	Open command palette
`Shift` - `Ctrl(⌘)` -d	Toggle single-document mode

Tools for writing, testing and debugging code

What tools do you use for writing, testing, and debugging code?

Some people prefer **terminal-based text editors** for writing code (e.g. Vi/Vim, Nano, Emacs, etc.).

Others prefer **integrated development environments (IDEs)**, which can bring "everything" one needs for productive programming to one's fingertips.a

Yet others prefer **code editors**, which are light-weight IDEs.

Terminal editor

- Good command line skills are needed for effectively using terminal editors
- Continue using Emacs and Vim, if you are already proficient
- Supports multiple programming languages

IDE

- If you are working with large code bases, then you should definitely checkout the IDE suitable for your programming language
- IDEs offer rich support for Debugging and Code refactoring
- Focused on a single language

Code editor

- If you use multiple programming languages then code editors offer good support
- Both IDE and code editors share common features such as code completion, hints, highlighting sections of code
- Supports multiple programming languages

A first computational notebook

Objectives

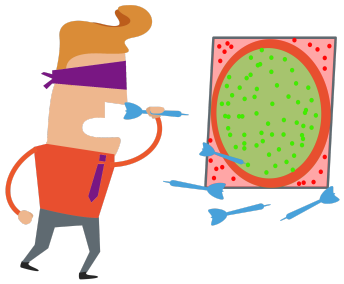
- Get started with notebooks for analysis.
- Get a feeling for the importance of execution order.

Instructor note

- 5 min teaching
- 20 min exercises

Creating a computational narrative

Let's create our first real computational narrative in a Jupyter notebook (adapted from Python and R data analysis course at Aalto Science IT).



Throwing darts to compute pi.

Imagine you are on a desert island and wish to compute pi. You have a computer with you with Python installed but no math libraries and no Wikipedia.

Here is one way of doing it - "throwing darts" by generating random points within a square area and checking whether the points fall within the unit circle.

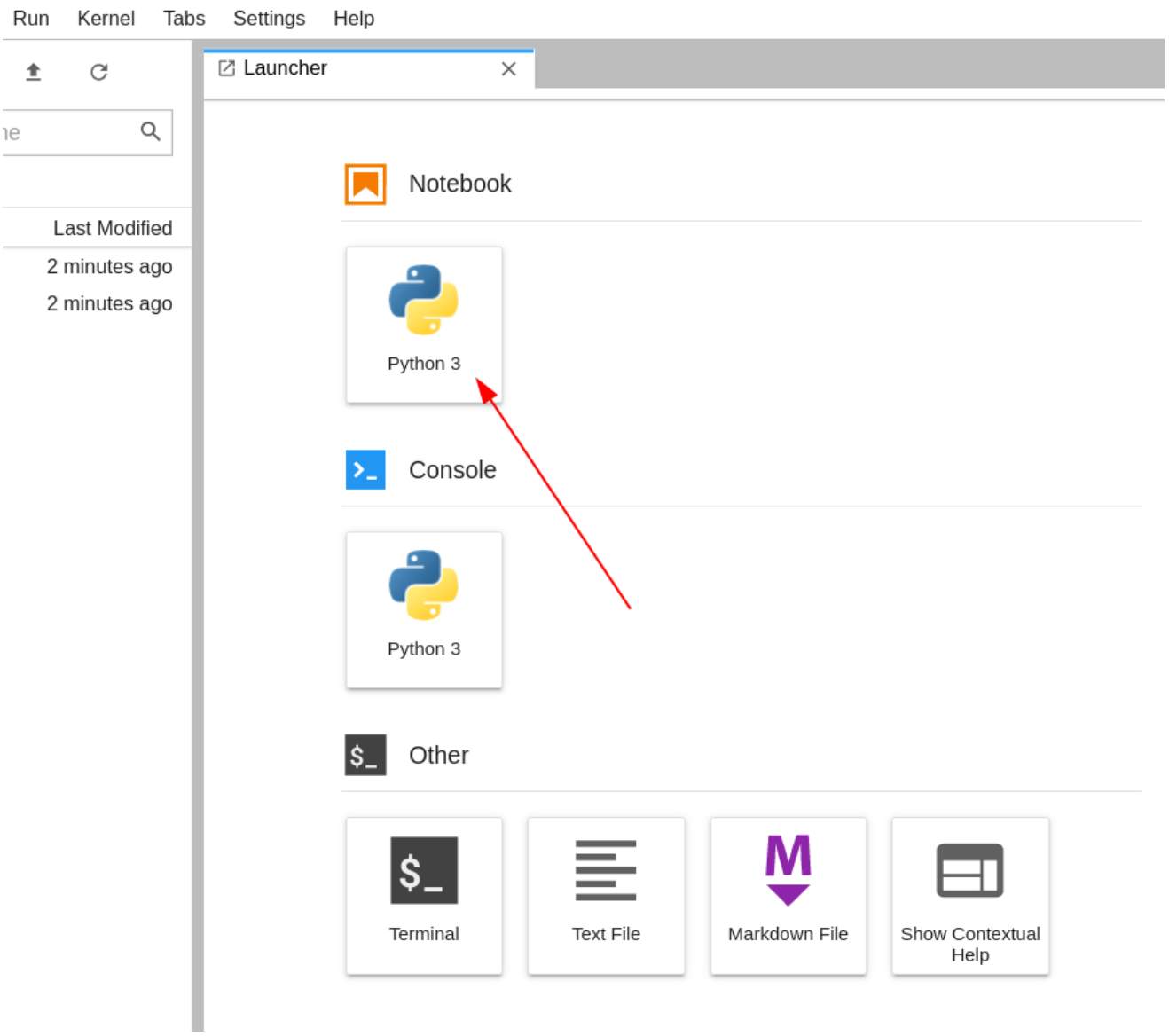
Launching JupyterLab notebook

In your terminal **first create a folder or navigate to a folder** where you would like the new notebook to appear.

After you have created a new folder and moved to it, launch JupyterLab:

```
$ jupyter-lab
```

This opens JupyterLab in your browser. Click on the Python 3 tile.



JupyterLab opened in the browser. Click on the Python 3 tile.

If you prefer to select in which browser to open JupyterLab, use:

```
$ jupyter-lab --no-browser
```

An example computational notebook

💬 Hint: Opening a webpage inside JupyterLab

If you would like to copy-paste content from this webpage into your Jupyter notebook, a cool way of doing it is to open this page inside an IFrame:

```
from IPython.display import IFrame
IFrame(src="https://coderefinery.github.io/jupyter/first-notebook/", width='100%',
height='500px')
```

✍️ Exercise/demonstration: Calculating pi using Monte Carlo methods

This can be either done as a 20 minute exercise or as a type-along demo.

Each numbered item will be a new cell. Press SHIFT+ENTER to run a cell and create a new cell below. With the cell selected, press ESCAPE to go into command mode. Use shortcuts `M` and `Y` to change cells to markdown and code, respectively.

1. Create a new notebook, name it, and add a heading (markdown cell).

```
# Calculating pi using Monte Carlo methods
```

2. Document the relevant formulas in a new cell (markdown cell):

```
## Relevant formulas

- square area: $s = (2 r)^2$
- circle area: $c = \pi r^2$
- $c/s = (\pi r^2) / (4 r^2) = \pi / 4$
- $\pi = 4 * c/s$
```

3. Add an image to explain the concept (markdown cell):

```
## Image to visualize the concept

![Darts]
(https://raw.githubusercontent.com/coderefinery/jupyter/main/example/darts.svg)
```

4. Import two modules that we will need (code cell):

```
# importing modules that we will need

import random
import matplotlib.pyplot as plt
```

5. Initialize the number of points (code cell):

```
# initializing the number of "throws"

num_points = 1000
```

6. “Throw darts” (code cell):

```
# here we "throw darts" and count the number of hits

points = []
hits = 0
for _ in range(num_points):
    x, y = random.random(), random.random()
    if x*x + y*y < 1.0:
        hits += 1
        points.append((x, y, "red"))
    else:
        points.append((x, y, "blue"))
```

7. Plot results (code cell):

```
# unzip points into 3 lists
x, y, colors = zip(*points)

# define figure dimensions
fig, ax = plt.subplots()
fig.set_size_inches(6.0, 6.0)

# plot results
ax.scatter(x, y, c=colors)
```

8. Compute the estimate for pi (code cell):

```
# compute and print the estimate

fraction = hits / num_points
4 * fraction
```

Here is the notebook:
https://github.com/coderefinery/jupyter/blob/main/example/darts.ipynb (static version, later we will learn how to share notebooks which are dynamic and can be modified).

Instructor note

Demonstrate out-of-order execution problems and how to avoid them:

- Add a cell at the end of the notebook which redefines `num_points`
- Then run the cell which computes the pi estimate
- Then demonstrate “run all cells”

Notebooks in other languages

It is possible to use Jupyter for other programming languages than Python (list of supported kernels). However, if you write R or Julia code, instead of installing a kernel, we recommend to use their corresponding notebook solutions which are optimized for these languages:

- R Markdown for R
- Pluto.jl for Julia

Discussion

What do we get from this?

- With code separate from everything else, you might just send one number or a plot to your supervisor/collaborator for checking.
- With a notebook as a narratives, you send everything in a **consistent story**.
- A reader may still just read the introduction and conclusion, but they can easily see more - *and try changes themselves* - if they want.

💬 Where should we add comments?

We can comment code either in **Markdown cells** or in the code cell as **code comments**.

What advantages do you see of commenting in Markdown cells and what advantages can you list for writing code comments in code cells?

❗ Keypoints

- Notebooks provide an intuitive way to perform interactive computational work.
- Allows fast feedback in your test-code-refactor loop.
- Cells can be executed in any order, beware of out-of-order execution bugs!

Notebooks and version control

❗ Objectives

- Demonstrate two tools which make version control of notebooks easier.

Instructor note

- 10 min teaching
- 5 min demo

Jupyter Notebooks are stored in JSON format. With this format it can be a bit difficult to compare and merge changes which are introduced through the notebook interface.

Packages and JupyterLab extensions to simplify version control

Several packages and JupyterLab extensions have been developed to make it easier to interact with Git and GitHub:

- nbdime (notebook "diff" and "merge") provides "content-aware" diffing and merging.
  - Adds a Git button to the notebook interface.
  - `git diff` and `git merge` shell commands can use nbdime's diff and merge for notebook files, but leave Git's behavior unchanged for non-notebook files.
- jupyterlab-git is a JupyterLab extension for version control using Git.
  - Adds a Git tab to the left-side menu bar for version control inside JupyterLab.
- JupyterLab GitHub is a JupyterLab extension for accessing GitHub repositories.
  - Adds a GitHub tab to the left-side menu bar where you can browse and open notebooks from your GitHub repositories.

All three extensions can be used from within the JupyterLab interface and our Conda environment provides jupyterlab-git and nbdime. To install additional extensions, please consult the official documentation about installing and managing JupyterLab extensions.

Comparing Jupyter Notebooks on GitHub

For this you really want to enable Rich Jupyter Notebook Diffs on GitHub:

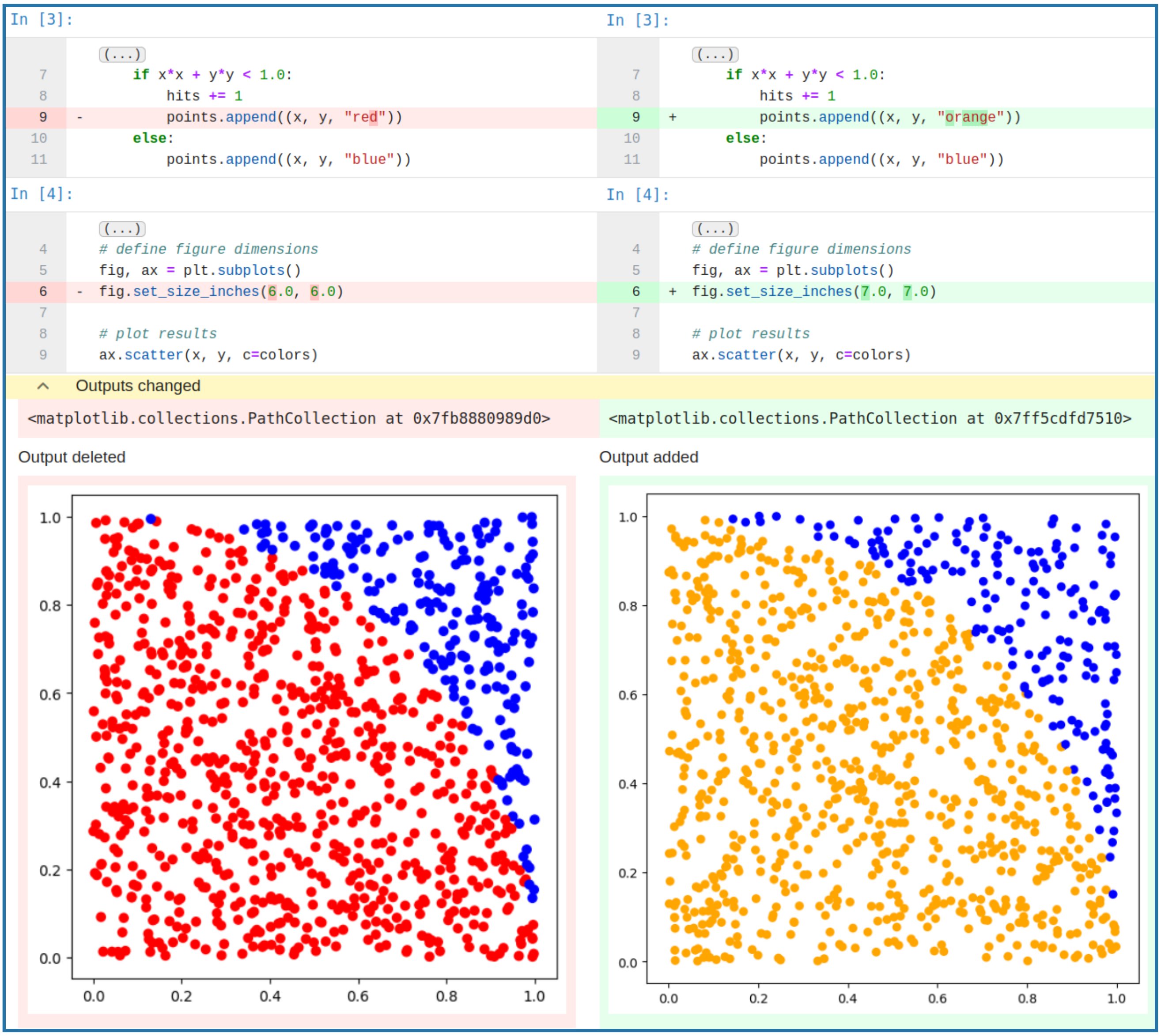
- On GitHub click on your avatar/image (top right).
- Click on "Feature preview".
- Enable "Rich Jupyter Notebook Diffs".

To demonstrate the difference we have created a small change and you can try to compare the effect yourself by enabling/disabling the feature:
https://github.com/coderefinery/jupyter/compare/5ff55b8..fce21e6

Here is the diff **without "Rich Jupyter Notebook Diffs"**:

A small change in a notebook viewed on GitHub without “Rich Jupyter Notebook Diffs” (truncated).

Here is the same change, but this time **with “Rich Jupyter Notebook Diffs” enabled**:



The same change viewed on GitHub, this time with “Rich Jupyter Notebook Diffs” enabled.

Comparing changes locally without jupyterlab-git/nbdime

Instructor note

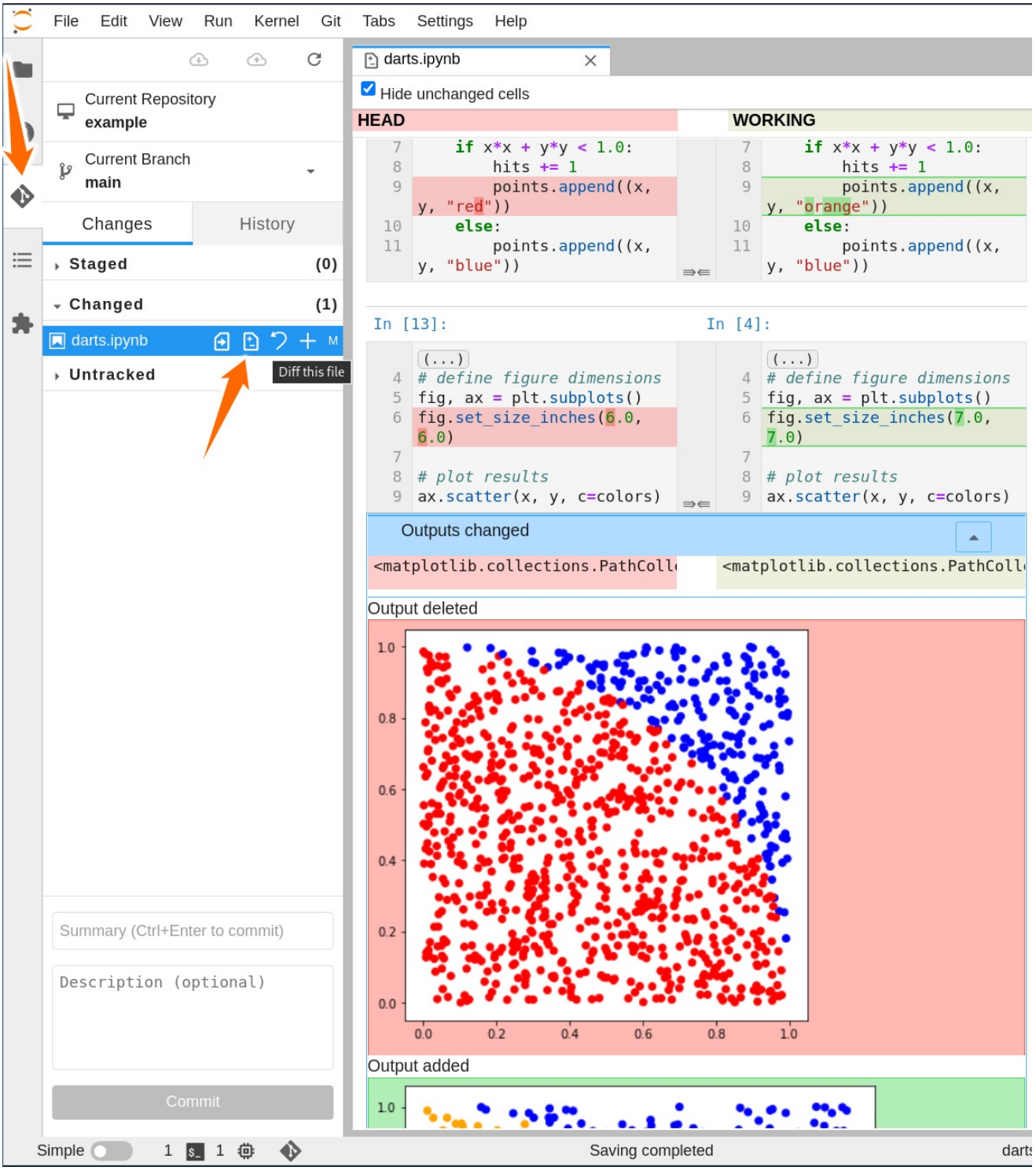
- Create a new folder
- Initialize a new Git repository (which is anyway good to demonstrate)
- Copy the “darts” notebook into it (from the previous episode)
- Add `.ipynb_checkpoints/` to `.gitignore`
- Stage and commit the file before trying the changes below

✍ Instructor demonstrates a plain git diff

1. To understand the problem, the instructor first shows the example notebook and then the source code in JSON format.
2. Then we introduce a simple change to the example notebook, for instance changing colors (change “red” and “blue” to something else) and also changing dimensions in `fig.set_size_inches(6.0, 6.0)`.
3. Run all cells.
4. We save the change (save icon) and in the JupyterLab terminal try a “normal” `git diff` and see that this is not very useful. Discuss why.

Comparing changes with jupyterlab-git/nbdime

Let us inspect the same changes using jupyterlab-git (which uses nbdime). This is more convenient since it highlights only the changes that we have made:



Comparing changes with jupyterlab-git/nbdime. Click on the Git tab, then on the plus-minus symbol.

Using nbdime on the command line

You can configure your (command line) Git to always use nbdime when comparing and merging notebooks:

```
$ nbdime config-git --enable --global
```

Now when you do git diff or git merge with notebooks, you should see a nice diff view. For more information please see the corresponding documentation.

See also

- nbdev developed by fast.ai is a notebook-driven development platform which includes support for git-friendly Jupyter notebooks
- Verdant is a JupyterLab extension that automatically records history of all experiments you run in a Jupyter notebook, and stores them in an `.ipyhistory` JSON file.

Sharing notebooks

Objectives

- Learn how to share notebooks with colleagues and the community?

Instructor note

- 5 min teaching
- 20 min exercises

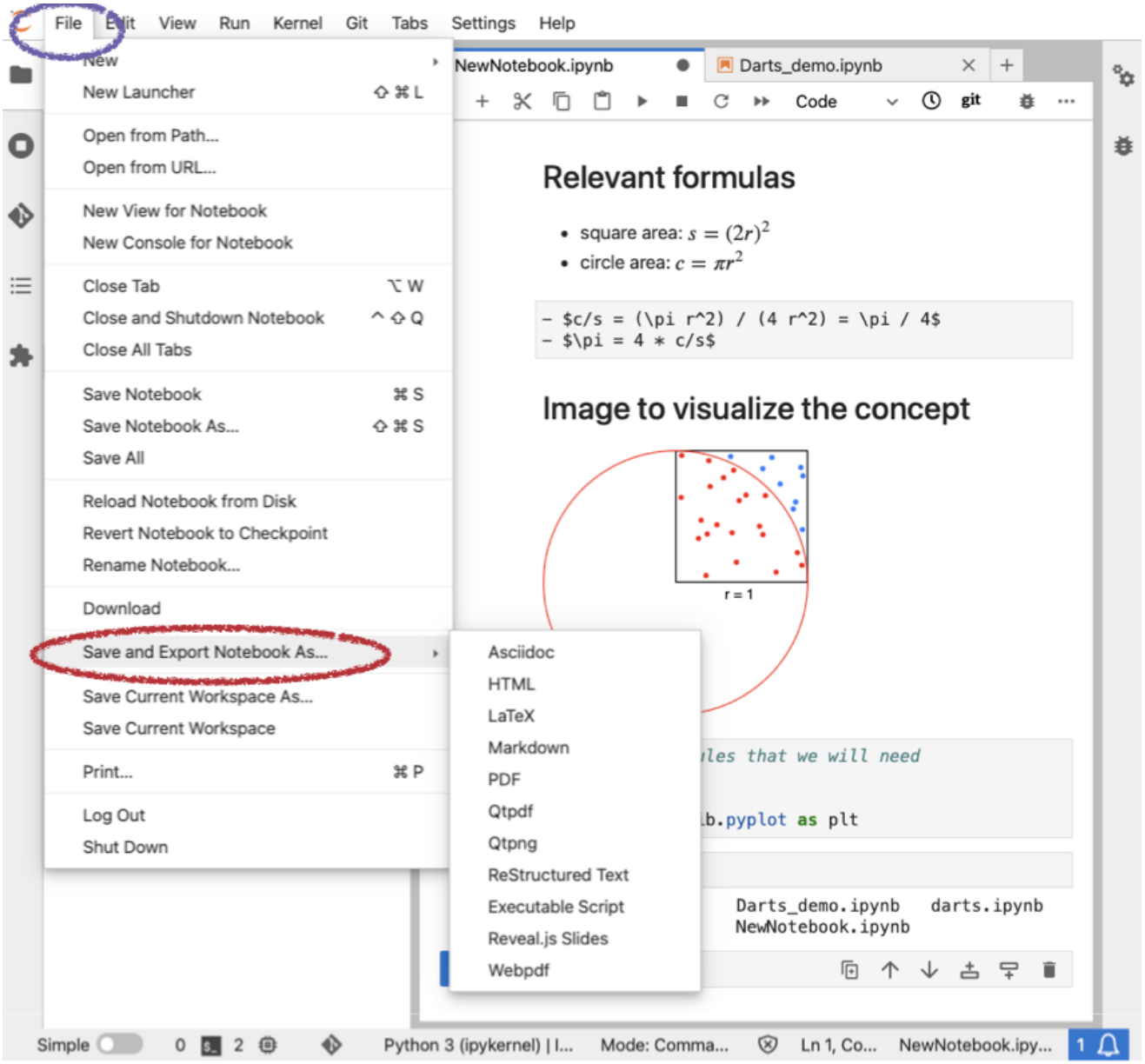
💬 Nudge your brain: When have you shared your code?

- To yourself between two computers?
- To a large audience eg. on a webpage?
- Coding together with a colleague?

Different ways to share a notebook

- You can enter a URL, GitHub repo or username, or GIST ID in nbviewer and view a rendered Jupyter notebook
- Read the Docs can render Jupyter Notebooks via the nbsphinx package
- Binder creates live notebooks based on a GitHub repository
- EGI Notebooks (see also https://egi-notebooks.readthedocs.io)
- JupyterLab supports sharing and collaborative editing of notebooks via Google Drive. Recently it also added support for Shared editing with collaborative notebook model.
- JupyterLite creates a Jupyterlab environment in the browser and can be hosted as a GitHub page.
- Notedown, Jupinx and DocOnce can take Markdown or Sphinx files and generate Jupyter Notebooks
- Voilà allows you to convert a Jupyter Notebook into an interactive dashboard
- The `jupyter nbconvert` tool can convert a (`.ipynb`) notebook file to:

- python code (`.py` file)
- an HTML file
- a LaTeX file
- a PDF file
- a slide-show in the browser



You can export Jupyter Notebooks to various formats. Some might need extra installations.

Commercial offers with free plans

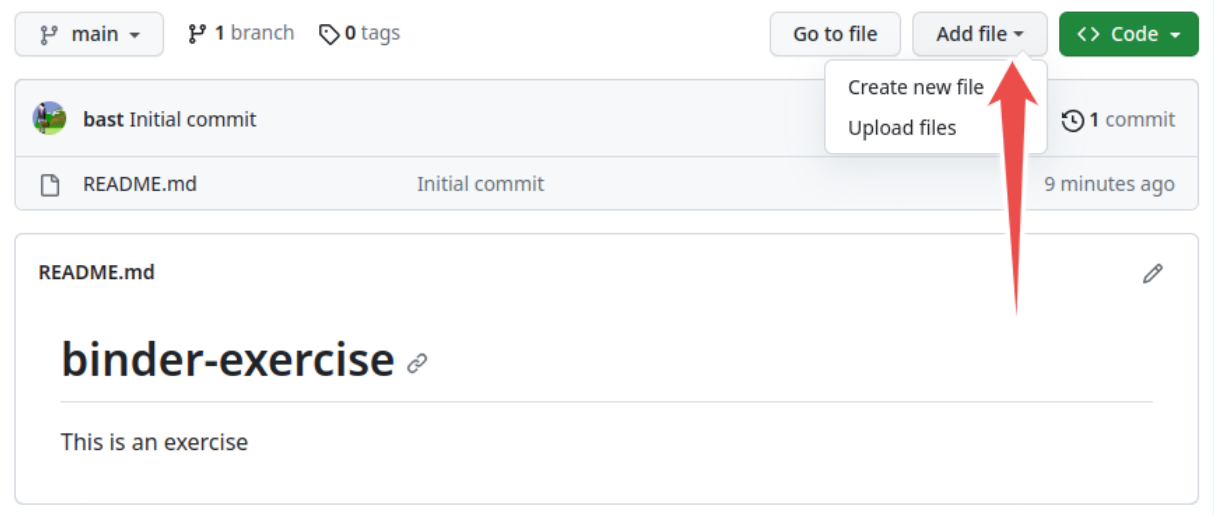
These platforms can be used free of charge but have paid subscriptions for faster access to cloud resources:

- CoCalc (formerly SageMathCloud) allows collaborative editing of notebooks in the cloud
- Google's Colaboratory lets you work on notebooks in the cloud, and you can read and write to notebook files on Drive
- Microsoft Azure Notebooks also offers free notebooks in the cloud
- Deepnote allows real-time collaboration

Sharing dynamic notebooks on Binder

✍ Exercise (20 min): Making your notebooks reproducible by anyone via Binder

- Create a new GitHub repository and **click on "Add a README file"**: https://github.com/new
- This exercise can be done entirely through the GitHub web interface (but using the terminal is of course also OK). You can use the "Add file" button to upload files:



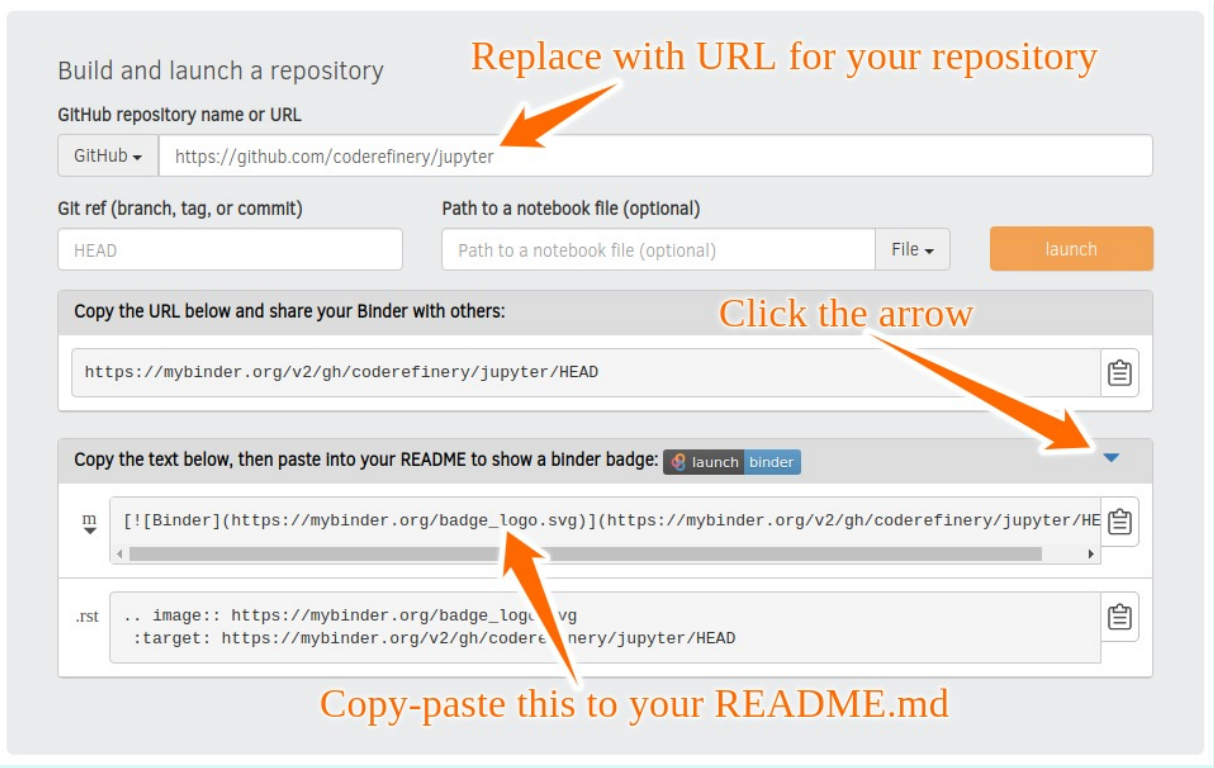
Screenshot of Binder web interface.

Jupyter Notebooks | R Markdown/R Studio project

- Upload the notebook which we have created earlier to this repository. If you got stuck earlier, you can download this notebook (right-click, "Save as ..."). You can also try this with a different notebook.
- Add also a `requirements.txt` file which contains (adapt this if your notebook has other dependencies):

```
matplotlib==3.4.1
```

- Visit https://mybinder.org:



Screenshot of Binder web interface.

- Copy-paste the markdown text for the mybinder badge into a README.md file in your notebook repository.
- Check that your notebook repository now has a "launch binder" badge in your `README.md` file on GitHub.
- Try clicking the button and see how your repository is launched on Binder (can take a minute or two). Your notebooks can now be expored and executed in the cloud.
- Enjoy being fully reproducible! Even better would be to get a DOI to your notebook and point Binder to the DOI.

❗ More examples with Binder:

- Binder documentation
- Collection of example repositories

Optional exercises

Importance of tracking dependencies

✍ (Optional) Exercise: what happens without requirements.txt?

Let's look at the same activity inequality repository.

- Start the repository in Binder using this link.
- `fig3/fig3bc.ipynb` is a Python notebook, so it works in Binder. Most others are in R, which also works in Binder. But how?
- Try to run the notebook. What happens?

- Most likely the run breaks down immediately in the first cell:

```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="whitegrid")
from itertools import cycle
```

We get a long list of `ModuleNotFoundError` messages. This is because the required Python packages have not been installed and can not be imported. The missing packages include, at least, `pandas` and `matplotlib` mentioned in the error message.

- To install the missing requirements, add a new code cell to the beginning of the notebook with the contents

```
!python3 -m pip install pandas matplotlib
```

and run the notebook again. What happens now?

- Again, the run breaks due to missing packages. This time the culprit is the `seaborn` package. Modify the first cell to also install it with

```
!python3 -m pip install pandas matplotlib seaborn
```

and try to run the notebook for the third time. Does it finally work? What could have been done differently by the developer?

- A good way to make a notebook more usable is to create a `requirements.txt` file containing the necessary packages to run the notebook and add it next to the notebook in the repository.
- In this case, the `requirements.txt` could look like this

```
pandas
matplotlib
seaborn
```

and to make sure the packages are installed, one could add a code cell to the beginning of original notebook with the line:

```
!python3 -m pip install -r requirements.txt
```

To make sure that the notebook will continue to work also in few months, you might want to specify also the version in the `requirements.txt` file.

Sharing an interactive notebook on Binder

✍ (Optional) Exercise: share an interactive (ipywidgets) notebook via Binder

- Take the solution from the exercise “Widgets for interactive data fitting” in the Examples of Jupyter features episode and paste it into a notebook.
- Push the notebook to a GitHub/GitLab repository.
- Create a `requirements.txt` file in your notebook repository, e.g.:

```
ipywidgets==7.4.2
numpy==1.16.4
matplotlib==3.1.0
```

- Try to deploy this example via Binder in the same way as the above exercise.

Summary

Recommendations for longer notebooks

Create a table of contents on top

You can do that using Markdown. This produces a nice overview for longer notebooks.
Example: https://stackoverflow.com/a/39817243

Jupyter Book

- https://jupyterbook.org
- You can cite and cross-reference
- You can toggle cell visibility
- … and a lot more

Discussion points

💬 Use cases and reproducibility

- If you are already using Jupyter, what tasks do you use it for?
- If you are new to Jupyter, do you see any possible use cases?
- Do you think Jupyter Notebooks can help tackle the problem of irreproducible results?

💬 Are Jupyter notebooks “workflows”?

- A reproducible workflow documents a “pipeline”
- Also a Jupyter notebook can be a data processing and visualization pipeline

- Are Jupyter notebooks “workflows”?

Interesting blog posts and articles

- A. Guzharina, “We Downloaded 10,000,000 Jupyter Notebooks From Github – This Is What We Learned”
- J. M. Perkel, “Reactive, reproducible, collaborative: computational notebooks evolve”, Nature 593, 156-157 (2021)

Similar tools for other languages

- R: R Shiny, R Markdown
- JavaScript: Observable
- Julia: Pluto

Avoiding repetitive code

It all started with a short and simple notebook but how to organize as projects and notebooks grow?

Let’s imagine we wrote this function `fancy_plot` for a hexagonal 2D histogram plot (please try it in your notebook):

```
import seaborn as sns

# to get some random numbers
from numpy.random import default_rng


# this one is simple but let us imagine something very lengthy
def fancy_plot(x_values, y_values, color):
    """
    Fancy function creating fancy plots.
    """
    sns.jointplot(x=x_values, y=y_values, kind="hex", color=color)


rng = default_rng()

x_values = rng.standard_normal(500)
y_values = rng.standard_normal(500)

# call our function
fancy_plot(x_values, y_values, "#4cb391")


other_x_values = rng.standard_normal(500)
other_y_values = rng.standard_normal(500)

# call our function again, this time with other data
fancy_plot(other_x_values, other_y_values, "#fc9272")
```

Now we would like to use this function in 5 other notebooks without duplicating it over all of the notebooks (imagine the function is very lengthy).

It can be useful to create a Python file `myplotfunctions.py` in the same folder as the notebooks (you can change the name) and place this code into `myplotfunctions.py`:

```
import seaborn as sns


# this one is simple but let us imagine something very lengthy
def fancy_plot(x_values, y_values, color):
    """
    Fancy function creating fancy plots.
    """
    sns.jointplot(x=x_values, y=y_values, kind="hex", color=color)
```

Now we can simplify our notebook:

```
# to get some random numbers
from numpy.random import default_rng

from myplotfunctions import fancy_plot


rng = default_rng()

x_values = rng.standard_normal(500)
y_values = rng.standard_normal(500)

# call our function
fancy_plot(x_values, y_values, "#4cb391")


other_x_values = rng.standard_normal(500)
other_y_values = rng.standard_normal(500)

# call our function again, this time with other data
fancy_plot(other_x_values, other_y_values, "#fc9272")
```

Shell commands, magics and widgets

❓ Questions

- Are there any other features besides code, text and output?

❗ Objectives

- Learn how to access help.
- Learn how to use magics and shell commands.
- Learn how to use widgets.

Extra features

Access to help

We can get help on an object using a question mark:

```
import numpy as np
np.sum?
```

Or two question marks to also see the source code:

```
np.sum??
```

List all names in a module matching pattern:

```
?np.*sum*
```

Output:

```
np.cumsum
np.einsum
np.einsum_path
np.nancumsum
np.nansum
np.sum
```

`%quickref` shows a quick reference card of features and shortcuts:

```
%quickref
```

Shell commands

- You can run shell commands by prepending with "!"
  - On Windows, GitBash needs to have the following option enabled: `Use Git and the optional Unix tools from the Windows Command Prompt`
- Make sure your cell command doesn't require interaction

```
!echo "hello"
```

Output:

```
hello
```

We can also capture the output of a shell command:

```
notebooks = !ls *.ipynb
```

- Common linux shell commands are also available as *magics*: %ls, %pwd, %mkdir, %cp, %mv, %cd, *etc.*.

- Using shell commands can be useful when testing a new idea but **for reproducible notebooks be careful with shell commands** (will they also work on a different computer?).

Magics

Magics are a simple command language which significantly extend the power of Jupyter.

There are two kinds of magics:

- **Line magics**: commands prepended by one % character and whose arguments only extend to the end of the current line.
- **Cell magics**: use two percent characters as a marker (%%), receive as argument the whole cell (must be used as the first line in a cell)

`%lsmagic` lists all available line and cell magics:

```
%lsmagic
```

Question mark shows help:

```
%sx?
```

Additional magics can also be installed or created.

Widgets

Widgets add more interactivity to Notebooks, allowing one to visualize and control changes in data, parameters etc.

```
from ipywidgets import interact
```

> The `ipywidgets` package is included in the standard CodeRefinery conda environment, but if you run into problems getting widgets to work please refer to the official installation instructions.

Use `interact` as a function

```
def f(x, y, s):
    return (x, y, s)

interact(f, x=True, y=1.0, s="Hello");
```

Use `interact` as a decorator

```
@interact(x=True, y=1.0, s="Hello")
def g(x, y, s):
    return (x, y, s)
```

💬 Does it not work? Extensions need to be installed.

The widgets interface have to be installed. JupyterLab is modular, and some parts need to be installed as an extension. In general, copy and paste the command into a shell (the JupyterLab shell works fine). See the installation instructions.

After installation, you need to reload the page to make it active (and if you installed it with pip or conda, restart the whole JupyterLab server)

✍️ A few useful magic commands

Using the computing-pi notebook, practice using a few magic commands. Remember that cell magics need to be on the first line of the cell.

1. In the cell with the for-loop over `num_points` (throwing darts), add the `%%timeit` cell magic and run the cell.
2. In the same cell, try instead the `%%prun` cell profiling magic.
3. Try introducing a bug in the code (e.g., use an incorrect variable name: `points.append((x, y2, True))`)
   - run the cell
   - after the exception occurs, run the `%debug` magic in a new cell to enter an interactive debugger
   - type `h` for a help menu, and `help <keyword>` for help on keyword
   - type `p x` to print the value of `x`
   - exit the debugger by typing `q`
4. Have a look at the output of `%lsmagic`, and use a question mark and double question mark to see help for a magic command that raises your interest.

✍️ Playing around with a widget

Widgets can be used to interactively explore or analyze data.

1. We return to the pi approximation example and create a new cell where we reuse code that we have written earlier but this time we place the code into functions. This "hides" details and allows us to reuse the functions later or in other notebooks:

```
import random
from ipywidgets import interact, widgets

%matplotlib inline
from matplotlib import pyplot


def throw_darts(num_points):
    points = []
    hits = 0
    for _ in range(num_points):
        x, y = random.random(), random.random()
        if x*x + y*y < 1.0:
            hits += 1
            points.append((x, y, True))
        else:
            points.append((x, y, False))
    fraction = hits / num_points
    pi = 4 * fraction
    return pi, points


def create_plot(points):
    x, y, colors = zip(*points)
    pyplot.scatter(x, y, c=colors)


def experiment(num_points):
    pi, points = throw_darts(num_points)
    create_plot(points)
    print("approximation:", pi)
```

2. Try to call the `experiment` function with e.g. `num_points` set to 2000.
3. Add a cell where we will make it possible to vary the number of points interactively:

```
interact(experiment, num_points=widgets.IntSlider(min=100, max=10000, step=100,
value=1000))
```

   If you run into `Error displaying widget: model not found`, you may need to refresh the page.

4. Drag the slider back and forth and observe the results.
5. Can you think of other interesting uses of widgets?

💬 RShiny is a nice R alternative/solution à la ipywidgets

RShiny is a nice R alternative/solution a la ipywidgets which can be interesting for R developers.

See for instance their gallery of examples.

Keypoints

- Jupyter notebooks have a number of extra features that can come in handy.

Examples of Jupyter features

❓ Questions

- Mixed examples/exercises to practice various aspects of using Jupyter

Objectives

- Learn more advanced usage of widgets.
- Learn how to profile code and install a new line-profiler tool.
- Practice some data analysis using pandas dataframes.
- Learn how to define your own magic command.
- Learn how to parallelize Python code using ipyparallel.
- Learn how to mix Python with R in the same noteobook.

Widgets for interactive data fitting

✍ Widgets for interactive data fitting

Widgets are fun, but they can also be useful. Here's an example showing how you can fit noisy data interactively.

1. Execute the cell below. It fits a 5th order polynomial to a gaussian function with some random noise
2. Use the `@interact` decorator around the last two code lines such that you can visualize fits with polynomial orders `n` ranging from, say, 3 to 30:

```
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

def gaussian(x, a, b, c):
    return a * np.exp(-b * (x-c)**2)

def noisy_gaussian():
    # gaussian array y in interval -5 <= x <= 5
    nx = 100
    x = np.linspace(-5.0, 5.0, nx)
    y = gaussian(x, a=2.0, b=0.5, c=1.5)
    noise = np.random.normal(0.0, 0.2, nx)
    y += noise
    return x, y

def fit(x, y, n):
    pfit = np.polyfit(x, y, n)
    yfit = np.polyval(pfit, x)
    return yfit

def plot(x, y, yfit):
    plt.plot(x, y, "r", label="Data")
    plt.plot(x, yfit, "b", label="Fit")
    plt.legend()
    plt.ylim(-0.5, 2.5)
    plt.show()

x, y = noisy_gaussian()
yfit = fit(x, y, n=5)  # fit a 5th order polynomial to it
plot(x, y, yfit)
```

✔ Solution

```
import numpy as np

from ipywidgets import interact

import matplotlib.pyplot as plt
%matplotlib inline

def gaussian(x, a, b, c):
    return a * np.exp(-b * (x-c)**2)

def noisy_gaussian():
    # gaussian array y in interval -5 <= x <= 5
    nx = 100
    x = np.linspace(-5.0, 5.0, nx)
    y = gaussian(x, a=2.0, b=0.5, c=1.5)
    noise = np.random.normal(0.0, 0.2, nx)
    y += noise
    return x, y

def fit(x, y, n):
    pfit = np.polyfit(x, y, n)
    yfit = np.polyval(pfit, x)
    return yfit

def plot(x, y, yfit):
    plt.plot(x, y, "r", label="Data")
    plt.plot(x, yfit, "b", label="Fit")
    plt.legend()
    plt.ylim(-0.5, 2.5)
    plt.show()

x, y = noisy_gaussian()

@interact
def slider(n=(3, 30)):
    yfit = fit(x, y, n)
    plot(x, y, yfit)
```

Cell profiling

✍ Cell profiling

This exercise is about cell profiling, but you will get practice in working with magics and cells.

1. Copy-paste the following code into a cell:

```
import numpy as np
import matplotlib.pyplot as plt

def step():
    import random
    return 1. if random.random() > .5 else -1.

def walk(n):
    x = np.zeros(n)
    dx = 1. / n
    for i in range(n - 1):
        x_new = x[i] + dx * step()
        if x_new > 5e-3:
            x[i + 1] = 0.
        else:
            x[i + 1] = x_new
    return x

n = 100000
x = walk(n)
```

2. Split up the functions over 4 cells (either via Edit menu or keyboard shortcut `Ctrl-Shift-minus`).
3. Plot the random walk trajectory using `plt.plot(x)`.
4. Time the execution of `walk()` with a line magic.
5. Run the prun cell profiler.
6. Can you spot a little mistake which is slowing down the code?
7. In the next exercise you will install a line profiler which will more easily expose the performance mistake.

✔ Solution

Split the code over multiple cells (e.g. using `Ctrl-Shift-minus`)

```
import numpy as np
```

```
def step():
    import random
    return 1. if random.random() > .5 else -1.
```

```
def walk(n):
    x = np.zeros(n)
    dx = 1. / n
    for i in range(n - 1):
        x_new = x[i] + dx * step()
        if x_new > 5e-3:
            x[i + 1] = 0.
        else:
            x[i + 1] = x_new
    return x
```

Initialize `n` and call `walk()`:

```
n = 100000
x = walk(n)
```

Plot the random walk

```
import matplotlib.pyplot as plt
plt.plot(x);
```

Time the execution using the `%timeit` line magic, and capture the output:

```
t1 = %timeit -o walk(n)
```

Best result

```
t1.best
```

Run with the `%%prun` cell profiler

```
%%prun
walk(n)
```

Installing a magic command for line profiling

✍ Installing a magic command for line profiling

Magics can be installed using `pip` and loaded like plugins using the `%load_ext` magic. You will now install a line-profiler to get more detailed profile, and hopefully find insight to speed up the code from the previous exercise.

1. If you haven't solved the previous exercise, copy paste the following code into a cell and run it:

```
import numpy as np
import matplotlib.pyplot as plt

def step():
    import random
    return 1. if random.random() > .5 else -1.

def walk(n):
    x = np.zeros(n)
    dx = 1. / n
    for i in range(n - 1):
        x_new = x[i] + dx * step()
        if x_new > 5e-3:
            x[i + 1] = 0.
        else:
            x[i + 1] = x_new
    return x

n = 100000
x = walk(n)
```

2. Then install the line profiler using `!pip install line_profiler`.
3. Next load it using `%load_ext line_profiler`.
4. Have a look at the new magic command that has been enabled with `%lprun?`
5. In a new cell, run the line profiler on the `walk` and `step` functions in the way described on the help page.
6. Inspect the output. Can you more easily see the mistake now?

✔ Solution

Copy-paste the code into a cell

Install the line profiler

```
!pip install line_profiler
```

Load the IPython extension

```
%load_ext line_profiler
```

See help:

```
%lprun?
```

Use the line profiler on the `walk` function:

```
%lprun -f walk walk(10000)
```

Aha, most time is spent on the line calling the `step()` function. Run line profiler on `step`:

```
%lprun -f step walk(10000)
```

```
...
     8                                           def step():
     9      9999       7488.0      0.7     52.3      import random
    10      9999       6840.0      0.7     47.7      return 1. if
random.random()
...
```

Aha! Lot's of time is spent on importing the `random` module inside the `step` function which is called thousands of times. Move the import statement to outside the function!

Data analysis with pandas dataframes

✍ Data analysis with pandas dataframes

Data science and data analysis are key use cases of Jupyter. In this exercise you will familiarize yourself with dataframes and various inbuilt analysis methods in the high-level `pandas` data exploration library. A dataset containing information on Nobel prizes will be viewed with the file browser.

1. Start by navigating in the File Browser to the `data/` subfolder, and double-click on the `nobels.csv` dataset. This will open JupyterLab's inbuilt data browser.
2. Have a look at the data, column names, etc.
3. In a your own notebook, import the `pandas` module and load the dataset into a *dataframe*:

```
import pandas as pd
nobel = pd.read_csv("data/nobels.csv")
```

4. The "share" column of the dataframe contains the number of Nobel recipients that shared the prize. Have a look at the statistics of this column using

```
nobel["share"].describe()
```

5. The `describe()` method is smart about data types. Try this:

```
nobel["bornCountryCode"].describe()
```

```
- What country has received the largest number of Nobel prizes, and how many?
- How many countries are represented in the dataset?
```

6. Now analyze the age of prize recipients. You first need to convert the "born" column to datetime format:

```
nobel["born"] = pd.to_datetime(nobel["born"],
                               errors ='coerce')
```

7. Next subtract the birth date from the year of receiving the prize and insert it into a new column "age":

```
nobel["age"] = nobel["year"] - nobel["born"].dt.year
```

- Now print the "surname" and "age" of first 10 entries using the `head()` method.

8. Now plot results in two different ways:

```
nobel["age"].plot.hist(bins=[20,30,40,50,60,70,80,90,100], alpha=0.6);
nobel.boxplot(column="age", by="category")
```

9. Which Nobel laureates have been Swedish? See if you can use the `nobel.loc[CONDITION]` statement to extract the relevant rows from the `nobel` dataframe using the appropriate condition.
10. Finally, try the powerful `groupby()` method to analyze the number of Nobel prizes per country, and visualize it with the high-level `seaborn` plotting library.

- First add a column "number" to the `nobel` dataframe containing 1's (to enable the counting below).
- Then extract any 4 countries (replace below) and create a subset of the dataframe:

```
countries = np.array([COUNTRY1, COUNTRY2, COUNTRY3, COUNTRY4])
nobel2 = nobel.loc[nobel['bornCountry'].isin(countries)]
```

- Next use `groupby()` and `sum()`, and inspect the resulting dataframe:

```
nobels_by_country = nobel2.groupby(['bornCountry',"category"], sort=True).sum()
```

- Next use the `pivot_table` method to reshape the dataframe to a spreadsheet-like structure, and display the result:

```
table = nobel2.pivot_table(values="number", index="bornCountry",
columns="category", aggfunc=np.sum)
```

- Finally visualize using a heatmap:

```
import seaborn as sns
sns.heatmap(table,linewidths=.5);
```

- Have a look at the help page for `sns.heatmap` and see if you can find an input parameter which annotates each cell in the plot with the count number.

✔ Solution

```
import numpy as np
import pandas as pd
nobel = pd.read_csv("data/nobels.csv")
```

```
nobel["share"].describe()
```

```
nobel["bornCountryCode"].describe()
```

- USA has received 275 prizes.
- 76 countries have received at least one prize.

```
nobel["born"] = pd.to_datetime(nobel["born"], errors ='coerce')
```

Add column

```
nobel["age"] = nobel["year"] - nobel["born"].dt.year
```

Print surname and age

```
nobel[["surname","age"]].head(10)
```

```
nobel["age"].plot.hist(bins=[20,30,40,50,60,70,80,90,100],alpha=0.6);
```

```
nobel.boxplot(column="age", by="category")
```

Which Nobel laureates have been Swedish?

```
nobel.loc[nobel["bornCountry"] == "Sweden"]
```

Finally, try the powerful `groupby()` method. Add extra column with number of Nobel prizes per row (needed for statistics)

```
nobel["number"] = 1.0
```

Pick a few countries to analyze further

```
countries = np.array(["Sweden", "United Kingdom", "France", "Denmark"])
nobel2 = nobel.loc[nobel['bornCountry'].isin(countries)]
```

```
table = nobel2.pivot_table(values="number", index="bornCountry",
                           columns="category", aggfunc=np.sum)
table
```

```
import seaborn as sns
sns.heatmap(table,linewidths=.5, annot=True);
```

Defining your own custom magic command

✍ Defining your own custom magic command

It is possible to create new magic commands using the `@register_cell_magic` decorator from the `IPython.core` library. Here you will create a cell magic command that compiles C++ code and executes it. This exercise requires that you have the GNU `g++` compiler installed on your computer.

> This example has been adapted from the IPython Minibook, by Cyrille Rossant, Packt Publishing, 2015.

1. First import `register_cell_magic`

```
from IPython.core.magic import register_cell_magic
```

2. Next copy-paste the following code into a cell, and execute it to register the new cell magic command:

```
@register_cell_magic
def cpp(line, cell):
    """Compile, execute C++ code, and return the standard output."""

    # We first retrieve the current IPython interpreter instance.
    ip = get_ipython()
    # We define the source and executable filenames.
    source_filename = '_temp.cpp'
    program_filename = '_temp'
    # We write the code to the C++ file.
    with open(source_filename, 'w') as f:
        f.write(cell)
    # We compile the C++ code into an executable.
    compile = ip.getoutput("g++ {0:s} -o {1:s}".format(
        source_filename, program_filename))
    # We execute the executable and return the output.
    output = ip.getoutput('./{0:s}'.format(program_filename))
    print('\n'.join(output))
```

- You can now start using the magic using `%%cpp`.

3. Write some C++ code into a cell and try executing it.
4. To be able to use the magic in another notebook, you need to add the following function at the end and then write the cell to a file in your PYTHONPATH. If the file is called `cpp_ext.py`, you can then load it by `%load_ext cpp_ext`.

```
def load_ipython_extension(ipython):
    ipython.register_magic_function(cpp,'cell')
```

✓ Solution

```
from IPython.core.magic import register_cell_magic
```

Add `load_ipython_extension` function, and write cell to file called `cpp_ext.py`:

```
%%writefile cpp_ext.py
def cpp(line, cell):
    """Compile, execute C++ code, and return the standard output."""

    # We first retrieve the current IPython interpreter instance.
    ip = get_ipython()
    # We define the source and executable filenames.
    source_filename = '_temp.cpp'
    program_filename = '_temp'
    # We write the code to the C++ file.
    with open(source_filename, 'w') as f:
        f.write(cell)
    # We compile the C++ code into an executable.
    compile = ip.getoutput("g++ {0:s} -o {1:s}".format(
        source_filename, program_filename))
    # We execute the executable and return the output.
    output = ip.getoutput('./{0:s}'.format(program_filename))
    print('\n'.join(output))

def load_ipython_extension(ipython):
    ipython.register_magic_function(cpp,'cell')
```

Load extension:

```
%load_ext cpp_ext
```

Get help on the cpp magic:

```
%%cpp?
```

Hello World program in C++

```
%%cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, World!";
    return 0;
}
```

Parallel Python with ipyparallel

✍ Parallel Python with ipyparallel

Traditionally, Python is considered to not support parallel programming very well (see "GIL"), and "proper" parallel programming should be left to "heavy-duty" languages like Fortran or C/C++ where OpenMP and MPI can be utilised.

However, IPython now supports many different styles of parallelism which can be useful to researchers. In particular, `ipyparallel` enables all types of parallel applications to be developed, executed, debugged, and monitored interactively. Possible use cases of `ipyparallel` include:

- Quickly parallelize algorithms that are embarrassingly parallel using a number of simple approaches.
- Run a set of tasks on a set of CPUs using dynamic load balancing.
- Develop, test and debug new parallel algorithms (that may use MPI) interactively.
- Analyze and visualize large datasets (that could be remote and/or distributed) interactively using IPython

This exercise is just to get started, for a thorough treatment see the official documentation and this detailed tutorial.

1. First install `ipyparallel` using `conda` or `pip`. Open a terminal window inside JupyterLab and do the installation.
2. After installing `ipyparallel`, you need to start an "IPython cluster". Do this in the terminal with `ipcluster start`.
3. Then import `ipyparallel` in your notebook, initialize a `Client` instance, and create *DirectView* object for direct execution on the engines:

```
import ipyparallel as ipp
client = ipp.Client()
print("Number of ipyparallel engines:", len(client.ids))
dview = client[:]
```

4. You have now started the parallel engines. To run something simple on each one of them, try the `apply_sync()` method:

```
dview.apply_sync(lambda : "Hello, World")
```

5. A serial evaluation of squares of integers can be seen in the code snippet below.

```
serial_result = list(map(lambda x:x**2, range(30)))
```

- Convert this to a parallel calculation on the engines using the `map_sync()` method of the DirectView instance. Time both serial and parallel versions using `%%timeit -n 1`.

6. You will now parallelize the evaluation of pi using a Monte Carlo method. First load modules, and export the `random` module to the engines:

```
from random import random
from math import pi
dview['random'] = random
```

Then execute the following code in a cell. The function `mcpi` is a Monte Carlo method to calculate π. Time the execution of this function using `%timeit -n 1` and a sample size of 10 million (`int(1e7)`).

```
def mcpi(nsamples):
    s = 0
    for i in range(nsamples):
        x = random()
        y = random()
        if x*x + y*y <= 1:
            s+=1
    return 4.*s/nsamples
```

Now take the incomplete function below which takes a `DirectView` object and a number of samples, divides the number of samples between the engines, and calls `mcpi()` with a subset of the samples on each engine. Complete the function (by replacing the `____` fields), call it with 10^7 samples, time it and compare with the serial call to `mcpi()`.

```
def multi_mcpi(dview, nsamples):
    # get total number target engines
    p = len(____.targets)
    if nsamples % p:
        # ensure even divisibility
        nsamples += p - (nsamples%p)

    subsamples = ____//p

    ar = view.apply(mcpi, ____)
    return sum(ar)/____
```

Final note: While parallelizing Python code is often worth it, there are other ways to get higher performance out of Python code. In particular, fast numerical packages like Numpy should be used, and significant speedup can be obtained with just-in-time compilation with Numba and/or C-extensions from Cython.

✓ Solution

Open terminal, run `ipcluster start` and wait a few seconds for the engines to start. Import module, create client and DirectView object:

```
import ipyparallel as ipp
client = ipp.Client()
dview = client[:]
dview
```

```
<DirectView [0, 1, 2, 3]>
```

Time the serial evaluation of the squaring lambda function:

```
%%timeit -n 1
serial_result = list(map(lambda x:x**2, range(30)))
```

Use the `map_sync` method of the DirectView instance:

```
%%timeit -n 1
parallel_result = list(dview.map_sync(lambda x:x**2, range(30)))
```

There probably won't be any speedup due to the communication overhead.

Focus instead on computing pi. Import modules, export `random` module to engines:

```
from random import random
from math import pi
dview['random'] = random
```

```
def mcpi(nsamples):
    s = 0
    for i in range(nsamples):
        x = random()
        y = random()
        if x*x + y*y <= 1:
            s+=1
    return 4.*s/nsamples
```

```
%%timeit -n 1
mcpi(int(1e7))
```

```
3.05 s ± 97.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Function for splitting up the samples and dispatching the chunks to the engines:

```
def multi_mcpi(view, nsamples):
    p = len(view.targets)
    if nsamples % p:
        # ensure even divisibility
        nsamples += p - (nsamples%p)

    subsamples = nsamples//p

    ar = view.apply(mcpi, subsamples)
    return sum(ar)/p
```

```
%%timeit -n 1
multi_mcpi(dview, int(1e7))
```

```
1.71 s ± 30.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Some speedup is seen!

Mixing Python and R

✍ Mixing Python and R

Your goal now is to define a pandas dataframe, and pass it into an R cell and plot it with an R plotting library.

1. First you need to install the necessary packages:

```
!conda install -c r r-essentials
!conda install -y rpy2
```

2. To run R from the Python kernel we need to load the rpy2 extension:

```
%load_ext rpy2.ipython
```

3. Run the following code in a code cell and plot it with the basic plot method of pandas dataframes:

```
import pandas as pd
df = pd.DataFrame({
    'cups_of_coffee': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    'productivity': [2, 5, 6, 8, 9, 8, 0, 1, 0, -1]
})
```

4. Now take the following R code, and use the `%%R` magic command to pass in and plot the pandas dataframe defined above (to find out how, use `%%R?`):

```
library(ggplot2)
ggplot(df, aes(x=cups_of_coffee, y=productivity)) + geom_line()
```

5. Play around with the flags for height, width, units and resolution to get a good looking graph.

✔ Solution

```
import pandas as pd
df = pd.DataFrame({
 'cups_of_coffee': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 'productivity': [2, 5, 6, 8, 9, 8, 0, 1, 0, -1]
})
```

```
%load_ext rpy2.ipython
```

```
%%R -i df -w 6 -h 4 --units cm -r 200
# the first line says 'import df and make default figure size 5 by 5 inches
# with resolution 200. You can change the units to px, cm, etc. as you wish.
library(ggplot2)
ggplot(df, aes(x=cups_of_coffee, y=productivity)) + geom_line();
```

Word-count analysis with widgets

✍ Word-count analysis with widgets

This exercise uses the word-count project from earlier lessons.

1. Have a look under the `data/` directory. You will see four .dat files containing word-count statistics from books. You can try opening one.
2. Open the Launcher, and open a new Text File.
3. Copy-paste the code below to the text file, and save it to a file `zipf.py` (note how syntax highlighting gets activated).

```
def load_word_counts(filename):
    """
    Load a list of (word, count, percentage) tuples from a file where each
    line is of the form "word count percentage". Lines starting with # are
    ignored.
    """
    counts = []
    with open(filename, "r") as input_fd:
            for line in input_fd:
          if not line.startswith("#"):
                    fields = line.split()
               counts.append((fields[0], int(fields[1]), float(fields[2])))
    return counts

def top_n_word(counts, n):
    """
    Given a list of (word, count, percentage) tuples,
    return the top n word counts.
    """
    limited_counts = counts[0:n]
    count_data = [count for (_, count, _) in limited_counts]
    return count_data

def zipf_analysis(input_file, n=10):
    counts = load_word_counts(input_file)
    top_n = top_n_word(counts, n)
    return top_n
```

4. Import the new zipf module, and have a look at the docstring for one of the functions:

```
import zipf
zipf.top_n_word?
```

5. Run the `zipf_analysis()` function for a processed datafile. Plot the output, and compare with a 1/N function, using the following code:

```
import matplotlib.pyplot as plt
%matplotlib inline

nmax = 10
z = zipf.zipf_analysis("data/isles.dat", nmax)
n = range(1,nmax+1)
z_norm = [i/z[0] for i in z]
plt.plot(n,z_norm)
inv_n = [1.0/i for i in n]
plt.plot(n, inv_n)
```

6. Add an interactive widget to analyze Zipf's law, using for example this code:

```
from ipywidgets import interact
import matplotlib.pyplot as plt
%matplotlib inline

nmax = 10
@interact(p=-1.0)
def zipf_plot(p):
    plt.clf()
    n = range(1,nmax+1)
    for f in ["data/isles.dat", "data/last.dat", "data/abyss.dat",
"data/sierra.dat"]:
        z = zipf.zipf_analysis(f, nmax)
        z_norm = [i/z[0] for i in z]
        plt.plot(n,z_norm)
    inv_n = [i**p for i in n]
    plt.plot(n, inv_n)
```

7. Add another widget parameter `nmax` to the above code to control the number of words displayed on the x-axis, e.g. `nmax=(6,14)`, and play around with both sliders.

✔ Solution

Code for a widget with sliding bars for both number of words and the inverse power:

```
from ipywidgets import interact
import matplotlib.pyplot as plt
%matplotlib inline

@interact(nmax=(6,14), p=-1.0)
def zipf_plot(nmax, p):
    plt.clf()
    #plt.figure()
    n = range(1,nmax+1)
    for f in ["data/isles.dat", "data/last.dat", "data/abyss.dat",
              "data/sierra.dat"]:
        z = zipf.zipf_analysis(f, nmax)
        z_norm = [i/z[0] for i in z]
        plt.plot(n,z_norm)
    inv_n = [i**p for i in n]
    plt.plot(n, inv_n)
```

List of exercises

Summary

JupyterLab and notebook interface:

- Tools for writing, testing and debugging code

A first computational notebook:

- An example computational notebook
- Notebooks in other languages

Notebooks and version control:

- plain-git-diff

Sharing notebooks:

- Sharing dynamic notebooks on Binder
- Importance of tracking dependencies
- Sharing an interactive notebook on Binder

Examples of Jupyter features:

- Widgets for interactive data fitting
- Cell profiling
- Installing a magic command for line profiling
- Data analysis with pandas dataframes
- Defining your own custom magic command
- Parallel Python with ipyparallel
- Mixing Python and R
- Word-count analysis with widgets

Full list

This is a list of all exercises and solutions in this lesson, mainly as a reference for helpers and instructors. This list is automatically generated from all of the other pages in the lesson. Any single teaching event will probably cover only a subset of these, depending on their interests.

Instructor guide

Why we teach this lesson

When CodeRefinery started teaching this lesson it was meant to introduce participants to a cool new tool around which there was a lot of buzz (particularly in data science), and which could be useful to researchers to quickly prototype code and analyze data in an interactive way. Since then, more and more participants are already using Jupyter for various purposes when they come to a workshop.

One purpose of teaching this lesson is still to introduce Jupyter to participants who haven't used it before. The episodes "Motivation", "The JupyterLab and notebook interface" and "A first computational notebook" are meant to inspire participants to use notebooks for certain appropriate tasks, highlighting in particular the "computational narrative" aspect which is brilliantly enabled by notebooks.

After the first three episodes the focus shifts to topics related to version control ("Notebooks and version control"), open science and reproducibility ("Sharing notebooks") which connects to topics covered in other CodeRefinery lessons.

There are also two optional episodes, "Shell commands, magics and widgets" and "Additional exercises" which go through various features and use-cases of Jupyter.

A key take-home message from this lesson should be that Jupyter notebooks can be a very useful tool for reproducible research, if used wisely.

Intended learning outcomes

By the end of this lesson, learners should:

- be able to explain what a computational narrative is
- be able to identify areas of their own work where Jupyter notebooks could be an appropriate tool
- be able to use the JupyterLab interface efficiently
- understand that version control is equally important for notebooks as for other code
- know how to version control notebooks efficiently using JupyterLab plugins
- know that notebooks can be used to document scientific analysis, and published e.g. as supplementary information with journal articles to aid reproducibility
- know how to share notebooks via Binder
- understand possible pitfalls of using notebooks

Timing and lesson placement

Detailed schedule

- 12:00 - 12:05 Jupyter notebooks
- 12:05 - 12:15 JupyterLab and notebook interface
- 12:15 - 12:40 A first computational notebook
  - Exercise (20 min)
- 12:40 - 12:50 Notebooks and version control demo
- 12:50 - 12:55 Sharing notebooks
- 12:55 - 13:05 Break
- 13:05 - 13:25 Binder exercise (20 min)
- 13:25 - 13:30 Summary

Exercises

- "A first computational notebook" can be done either as a 20 minute exercise or as a type-along demo.
- "Instructor demonstrates a plain git diff" should be done as demonstration.
- "Making your notebooks reproducible by anyone via Binder" should be done as a 20 minute exercise but can also be done as a demo.
- There are three optional exercises in "Sharing notebooks", one on trying to reproduce results from a published notebook, another on sharing an interactive notebook on Binder, and one for R users who can try to deploy R Studio/ R Markdown to Binder.
- The "Examples" episode contains many interesting examples which can be used for demonstration or as exercises. The dependencies for ipywidget examples are typically tricky to install/enable in a group exercise. Instead they can be demonstrated on Binder (there is an optional exercise for this).

How to teach this lesson

Motivation

How the instructor introduces and motivates Jupyter notebooks is flexible and can depend on the instructor's background. The first episode emphasizes the "computational narrative" aspect of notebooks, and highlights a few common use cases. The gravitational-wave discovery is used as a motivational example, and it's helpful if the instructor clicks the Binder link to see how the notebooks become available for interactive exploration in the cloud. The instructor should also open the link "Gallery of interesting Jupyter notebooks" to show the wide variety of notebooks that people have shared online.

The JupyterLab interface

The second episode deals with the JupyterLab interface and how notebooks work. At this stage the instructor should open Jupyter-Lab, demonstrate the interface by clicking around and then launch a new notebook. Inside this notebook the instructor can add headings and text and a simple code cell to illustrate how cells work (copy-pasting from the lesson works well). A few important keyboard shortcuts can be demonstrated.

There is a discussion point on integrated development environments. This can be used as a discussion exercise, where participants are invited to talk about their prefered way to write code. The instructur can mention that JupyterLab is sort of like an IDE for notebooks.

Take-home messages:

- The JupyterLab interface is flexible and one can customize the workspace by dragging notebooks, terminals and text editors around.
- There are code cells and markdown cells that work differently, and there are command and edit modes. It's easy to switch between these.

A first computational notebook

To show that Jupyter Notebooks are rather simple and intuitive, the third episode demonstrates the building up of a computational narrative to compute pi and adding comments, equations and figures. The instructor should create a new notebook, name it and then type out or copy-paste from the lesson into notebook cells. Learners should be given time to follow along interactively.

After the notebook is completed, participants and instructor should commit it to the repository.

Take-home messages:

- Notebooks provide a simple and interactive tool for various kinds of analysis.
- Keyboard shortcuts enable efficient usage. The instructor should clearly explain the most common ones for executing cells, creating new cells, changing between markdown and code cells, etc.
- The execution order of cells matters, the instructor can demonstrate this by going up and down in the notebook.

Notebooks and version control

After discussing the Git integrations, the instructor should encourage participants to initialize a Git repo in their notebook directory, and commit the first "testing" notebook. They can do this via the JupyterLab interface if they have the plugins installed, or via terminal inside JupyterLab, or via regular terminal.

Take home message: The Git integration in JupyterLab is powerful and enables tracking notebooks in just the same way as one would with source code files.

Sharing notebooks

This episode has a list of services and tools to share and collaborate on Jupyter notebooks. The instructor doesn't have to go through each item in the list, but rather emphasize that many services and tools built around Jupyter exist.

The exercise on making notebooks reproducible via Binder is an important one as it connects with other lessons and emphasizes the reproducibility benefits of notebooks. If time allows, the instructor can let participants set up a remote repository on GitHub for their local notebook repos, and push to the remote. After that they can follow the exercise steps to obtain a Binder badge to add to their repository README files. If time is short, the instructor can instead just demonstrate all steps and encourage participants to try it out themselves later.

(Optional) Shell commands, magics and widgets

Take-home messages:

- There is more to notebooks than just code.
- Magics and shell commands can be useful for various kinds of workflows, and enable users to stay within the notebook instead of jumping to another terminal.
- Widgets add even more interactivity to notebooks.

(Optional) Additional exercises

Interesting and fun exercises to learn about various Jupyter features and use cases.

Credit and license

This material is provided by CodeRefinery under the licenses stated below.

Website template

The website template is maintained by CodeRefinery and rendered with sphinx-lesson: structured lessons with Sphinx.

Instructional material

- **Attribution** - You must give appropriate credit (mentioning that your work is derived from work that is Copyright (c) CodeRefinery and, where practical, linking to https://coderefinery.org, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions - You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. With the understanding that:

- You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.
- No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.