Research Software Engineering

DAY 2

Outline

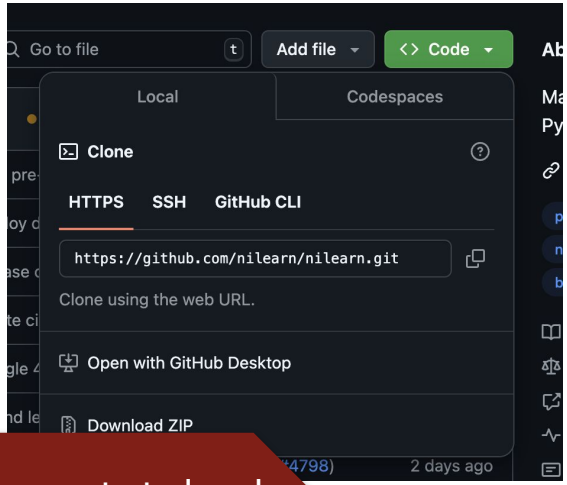
Day 2				
Topic	**Time**			**Duration**
Welcome + Introduction	9:00	-	9:15	0:15
Git continued: cloning repositories, command line git Activity 1 (15min) Activity 2 (15min)	9:15	-	10:00	0:45
Break	10:00	-	10:15	0:15
Introduction to reproducible code, good coding practices and stabilizing your computing environment	10:10	-	10:45	0:30
Activity 3 (40 min): Reproducible code, reproducible code, good coding practices and stabilizing your computing environment	10:45	-	11:25	0:40
Break	11:25	-	11:32	0:10
Thinking about the user: documentation, packaging, error messages, and more	11:35	-	11:50	0:25
Wrap-up	11:50	-	12:00	0:10

Cloning - remote to local

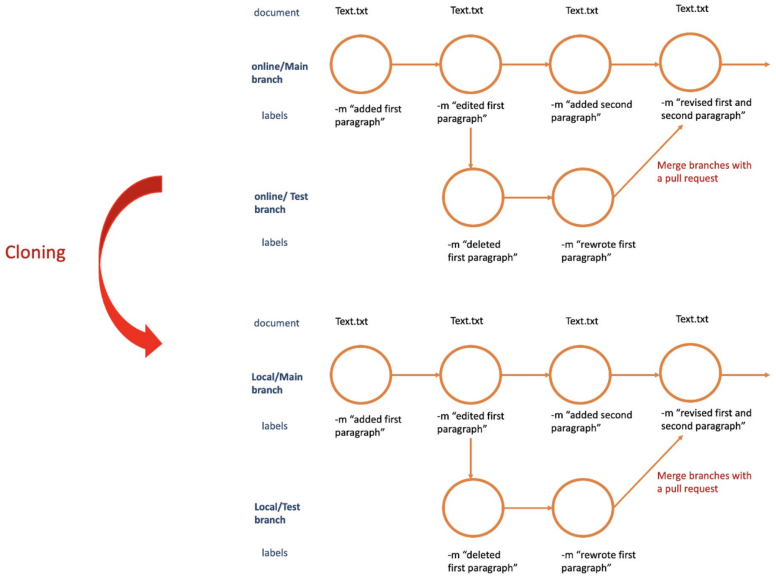
Git and Github: tracking your files, scripts and code

Collaboration (github online)

- **Clone:** local copy of repository, allowing offline work

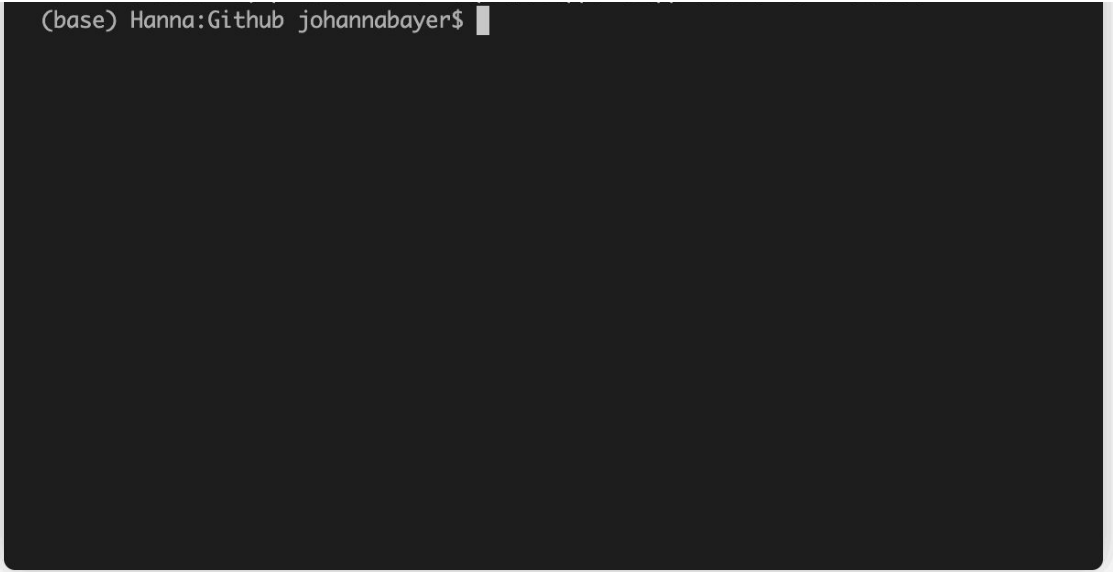


Cloning - remote to local



Cloning

```
git clone <url>

git status

git log

git add <filename>

git add .

git commit -m “commit message”

git push origin main
```



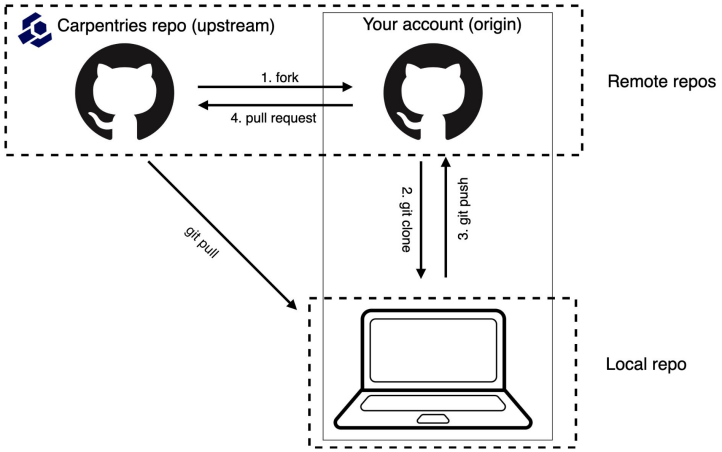
Cloning - remote to local

Git and github - terms

At some point, many versions of a repository will exist

- **Origin**: your fork of a repository
- **Upstream**: the repository the origin was forked from

- **Remote**: online repository that your local repository communicates with (can be remote or upstream)
- **Local**: local copy of repository



https://gcapes.github.io/swc-pr-tutorial/03-remotes/index.html

Cloning - remote to local

Synchronizing repositories

Changes made in one repository need to be populated to other copies

- Git pull: get changes from remote
- Git push: push changes to remote
- Git merge:
- Sync: set current branch to state of incoming branch (and disregard local changes)
- It can quite often come to merge conflicts during these processes

More commands can be found in the cheat sheet.

Git commands on windows

- Install Git bash - ubuntu terminal emulator that allows to use git using linux commands
- Select Windows
  - Works like a ubuntu terminal (for simple commands)

Other GUIs: https://git-scm.com/downloads/guis

Cloning - remote to local

Github Desktop

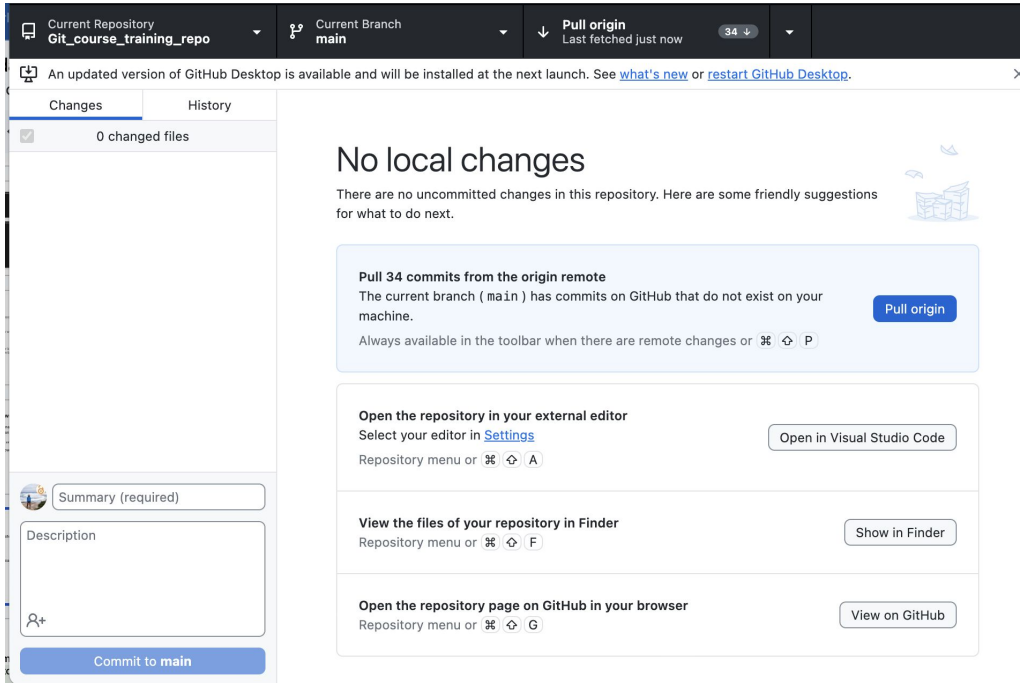
GUI that lets you interact with remote Github

Open with github desktop option for cloning

No ssh key required

Download here:

https://desktop.github.com/download/



Cloning - remote to local

Activity 1 (15’):

Interaction between Github online and your local machine

1. Clone your fork of the repository to your local machine.
2. Make some changes to your file.
3. Push back to origin.

Adding a remote to a local repository

Local git: git terminal commands

Initialize a repository as git repository

```
[johbay@ip-213-94 git_test % git init
```

Configure

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Staging and committing

```
[johbay@ip-213-94 git_test % git add .
[johbay@ip-213-94 git_test % git commit -m "added python script"
```

Adding a remote

Local git: git terminal commands

Check for a remote:

```
[johbay@ip-213-94 PCNtoolkit % git remote -v
origin  https://github.com/likeajumprope/PCNtoolkit.git (fetch)
origin  https://github.com/likeajumprope/PCNtoolkit.git (push)
upstream        https://github.com/amarquand/PCNtoolkit.git (fetch)
upstream        https://github.com/amarquand/PCNtoolkit.git (push)
```

Add a remote

```
[johbay@ip-213-94 git_test % git remote add origin https://github.com/likeajumprope/test_repo.git
[johbay@ip-213-94 git_test % git remote -v
origin  https://github.com/likeajumprope/test_repo.git (fetch)
origin  https://github.com/likeajumprope/test_repo.git (push)
```

Push/pull

```
[johbay@ip-213-94 git_test % git push
```

Adding a remote

Setting up an ssh key - authentication

Open a terminal (or git bash) and run:

```
ls -al ~/.ssh  # will list all your existing ssh keys
```

Generate a new ssh key via (just press enter twice in the process)

```
ssh-keygen -t ed25519 -C "your_email@example.com"  #or
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

When prompted, press enter (twice).

Start the ssh agent and add the key:

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519 #(Use id_rsa if you created an RSA key.)
```

Copy the **PUBLIC** key:

```
cat ~/.ssh/id_ed25519.pub
```

Setting up an ssh key - authentication

Add the key to GitHub

1. Go to GitHub.com → Profile Icon → **Settings**
2. Click **SSH and GPG Keys**
3. Click **New SSH Key**
4. Give it a title (e.g., “Laptop” or “Workstation”)
5. Paste the copied public key and click **Add SSH Key**

Test the connection:

```
ssh -T git@github.com
```

Integration: Terminal

- Git Bash for Windows:
  - https://gitforwindows.org/
- Git for UNIX/MACOS
- On a HCP server
  - Might require to generate an SSH key to log into the github account
  - Many students/researchers struggle with this

```bash
ssh-keygen -t rsa -b 4096 -C "your.email@example.com"
```

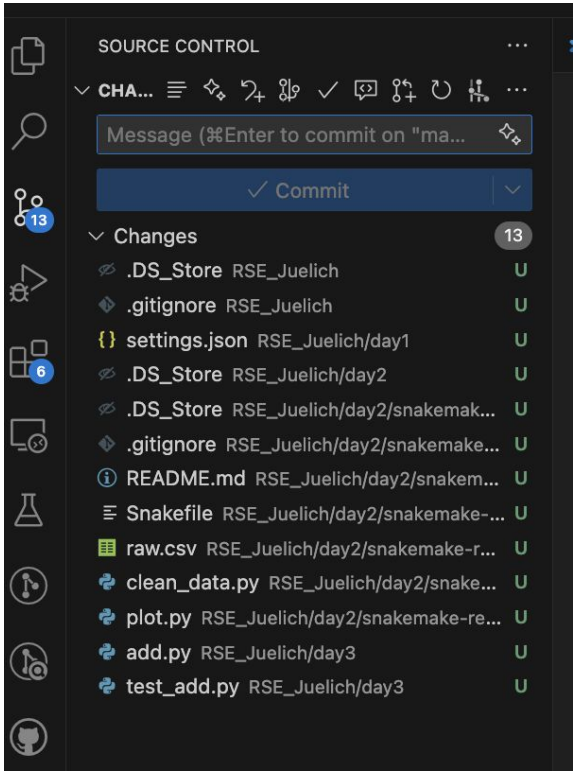
```bash
cat ~/.ssh/id_rsa.pub
```

```bash
ssh -T git@github.com
```

VS code and Github

- VS code allows full functionality to interact with Git/Github
- Extensions:
  - Source control
  - Git Lens (paid)

- Logging in with your Github account allows to pull repositories directly from Github.com



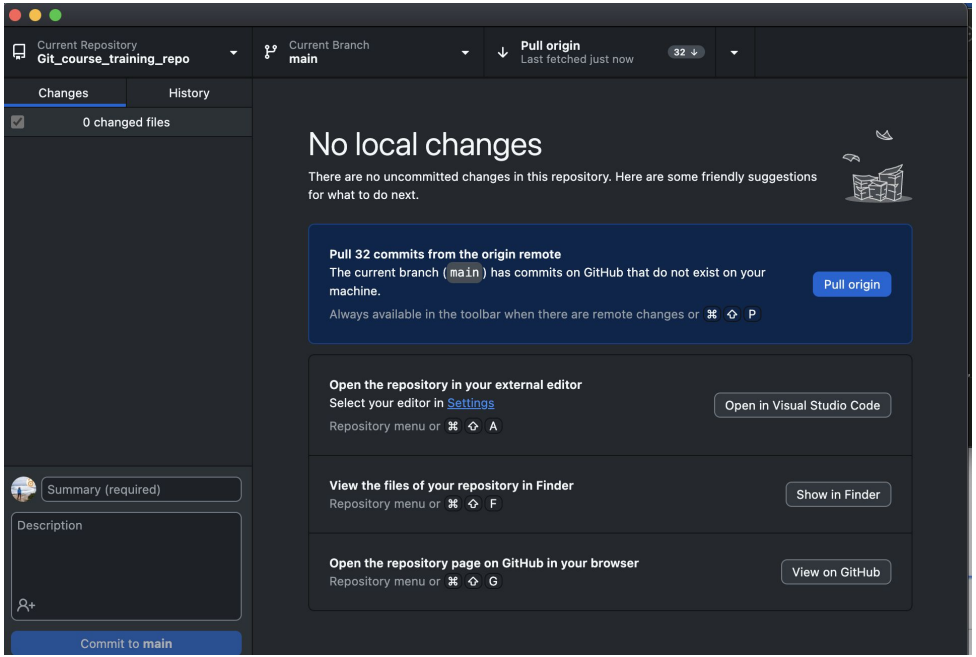
Integration: Github Desktop & Visual Studio code

Github Desktop

- In house GUI from Git:

  https://desktop.github.com/download/

- No ssh keys and no git init required
- Limited commands
- Good for newbies



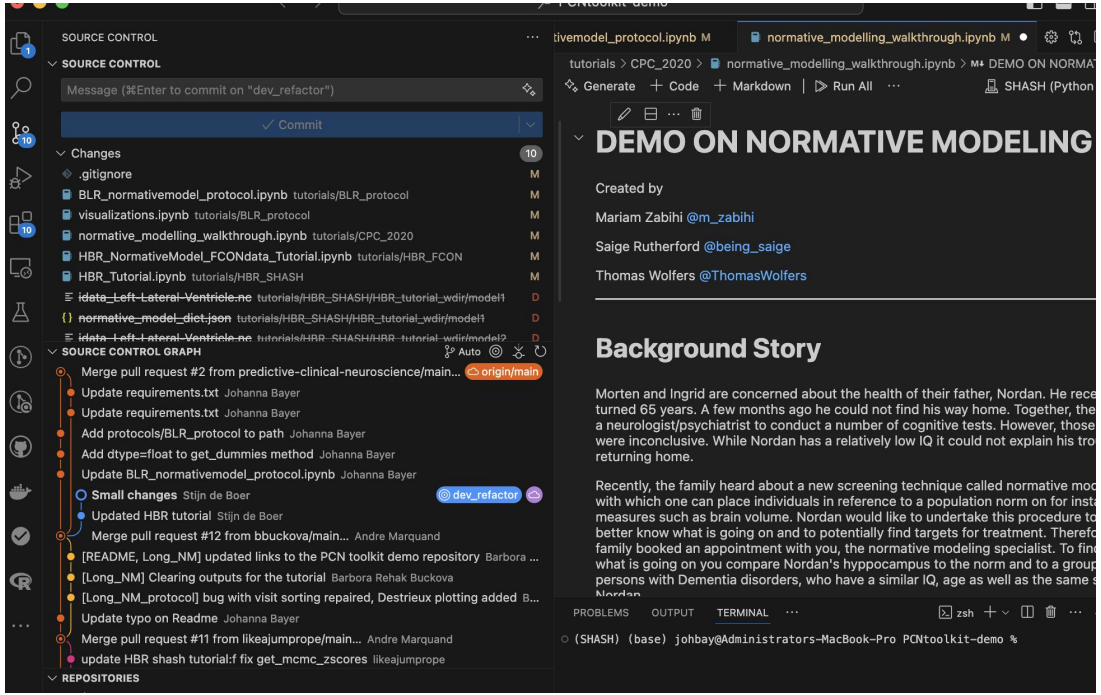
VS code

Local to remote

Integration: Github Desktop & Visual Studio code

VS code

- Various extensions, among them Git (Source Control) and Git Lense (paid)
- Log in with your github account
- Provides interface equivalent to command line



VS code

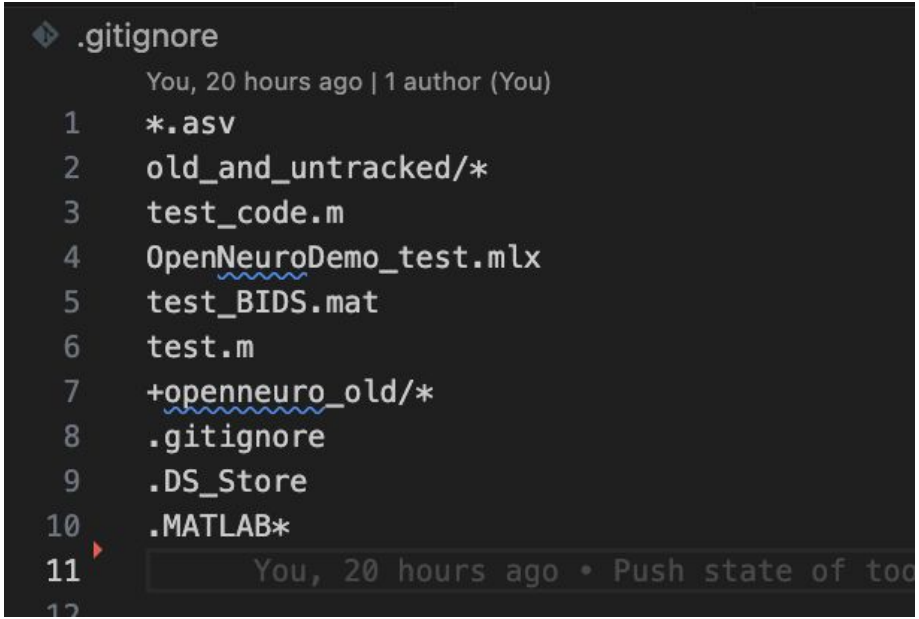
Local to remote

Git best practices

1. Make small changes, commit frequently and write good commit messages.
2. One issue, one branch, one PR. Create an issue about an outstanding feature. Link a branch to work on that feature to the issue and then submit a PR with the fix, also linked to the issue.
3. Be nice and polite when collaborating.
4. Avoid committing directly into the main branch. Better: Have feature branches and a dev branch to test your commits. This is especially true when making PRs to foreign repositories. PRs to the main branch are rarely accepted
5. Look for “good first issues” for beginners.
6. Assign several reviewers for PRs.
7. Never merge your own PRs.

.gitignore file

- File that contains files and folder that should not be tracked
- Useful when you have
  - Sensitive files
  - Large files
- Needs to be set up before first commit (tracking can be reversed but is more advanced)
- Make use of wildcards!



Advanced git - a teaser

Cherry picking:

- Select and merge individual commits

Various merge types

- Rebase
- Fast forward

Squashing:

- Collapse several commands to one commit
- Good for open source work

Rewriting history

Activity 2 (15’):

Adding a remote

1. Start with a local repositor
2. Create a connection with a remote
3. Push some changes

15 min break

Reproducible code

Reproducibility - terms

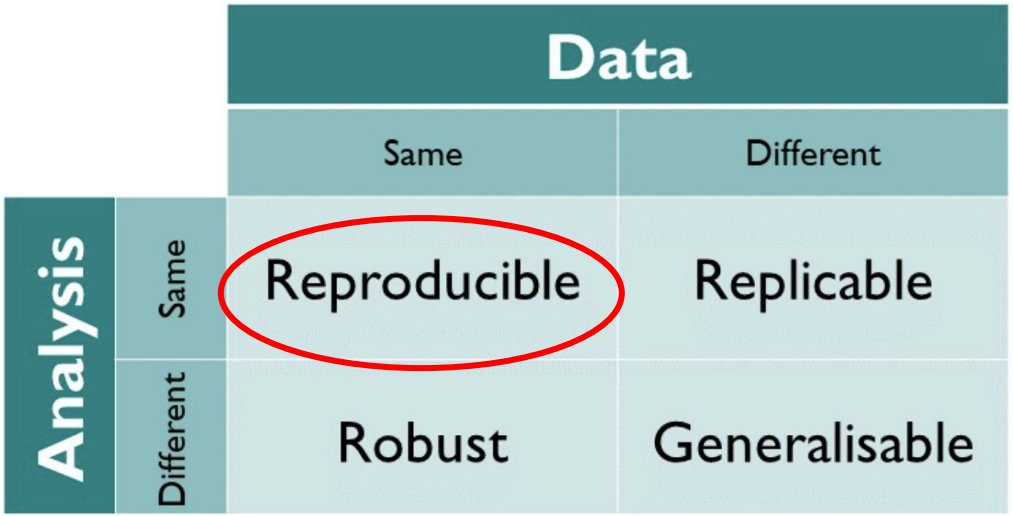


Figure 1: How the Turing Way defines reproducible research

- Reproducible:
  - Get the same results with the same data
  - Your future self
  - Someone else/ your colleague etc.

Computational reproducibility

Example of non computational reproducibility

In Neuroimaging, the reproducibility of results have been shown to be dependent on (Everything Matters):

- Computational environments matter (Glatard et al., 2015); eg. Operating system
- Tool selection matters (Tustison et al., 2014; Dickie et al., 2017);
- Tool version matters (Dickie et al., 2017);
- Statistical model matters (Tan et al., 2016);

https://www.frontiersin.org/journals/neuroinformatics/articles/10.3389/fninf.2019.00001/full#B12l;

Computational reproducibility and false positives

Open access, freely available online

Essay

Why Most Published Research Findings Are False

John P. A. Ioannidis

- When the power (1- beta) of a statistical test falls, the probability of a significant result is more likely to be due to false positives than to an actual result.
- This dynamic is more likely in fields with
  - The number of test
  - Small sample sizes
  - Small effect sizes
  - Flexibility in the description of methods and design

Discussion 1:

From your own experience/work:
What enhances computational reproducibility?
What hinders computational reproducibility?

Good coding practices

Good coding practices

… code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, “Readability counts”.

https://peps.python.org/pep-0008/

Good coding practices across platforms

Research code is lacking:

- Modularity by defining **functions**

- Documentation
  - Docstrings in Python (Functions!)
  - Roxygen (Functions!)

- Tests
  - Pytest (Functions!)
  - test_that(Functions!)

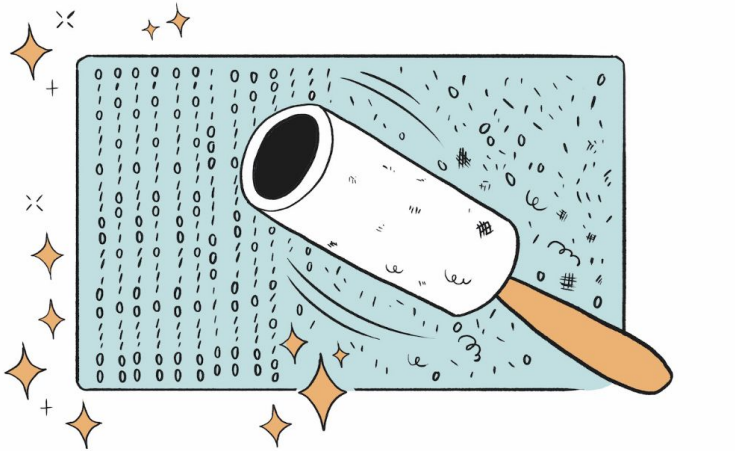


Fig. 75 The Turing Way project illustration by Scriberia. Used under a CC-BY 4.0 licence. DOI: 10.5281/zenodo.13882307.

Code style & automatic formatting

Convention on how to format code.

Examples: how to name variables? How to indent codes?

Coding styles:

- Python: PEP8
- Matlab: https://nl.mathworks.com/matlabcentral/fileexchange/46056-matlab-style-guidelines-2-0
- Fortran: https://fortran-lang.org/learn/best_practices/
- Julia: https://docs.julialang.org/en/v1/manual/style-guide/

Automatic formatting: tools that automatically format your code

Static code analysis (linting)

- method that examines code and detects software vulnerabilities before your code is executed or the project is built and deployed.

- This analysis is capable of identifying quality issues, including security weaknesses and errors. In addition to finding bugs, many of these tools can also help maintain a consistent coding style.

Formatters and linters

Language	Formatters	Linter (Static code analysis)	Notes
Python	Black, autopep8, yapf	Flake8, pylint, ruff, mypy	black is most popular; ruff is fast and multi-tool
C++	Clang-format, astyle	Clang-tidy, cppchek	Standard tools; widely supported in IDEs
Fortran	fprettify, f90nml (limited)	fortlint, FoBiS, fprettify	Tooling is limited but improving
Julia	JuliaFormatter.jl	Lint.jl, StaticLint.jl	Well-supported in VS Code
MATLAB		MLint (Code Analyzer in GUI)	Mostly GUI support; no standard CLI formatter/linter

https://book.the-turing-way.org/reproducible-research/code-quality/code-quality-style

Good coding practices: Use of linting tools

E.g pylint

```
# Run this command in your terminal to use a linter (e.g., pylint)
# pylint your_file.py

# Example of code that a linter might flag
x = 5
y = 10
z = x+y  # Linter might suggest adding spaces around the '+' operator

# Corrected version
x = 5
y = 10
z = x + y
```

Good coding practices: Write functions

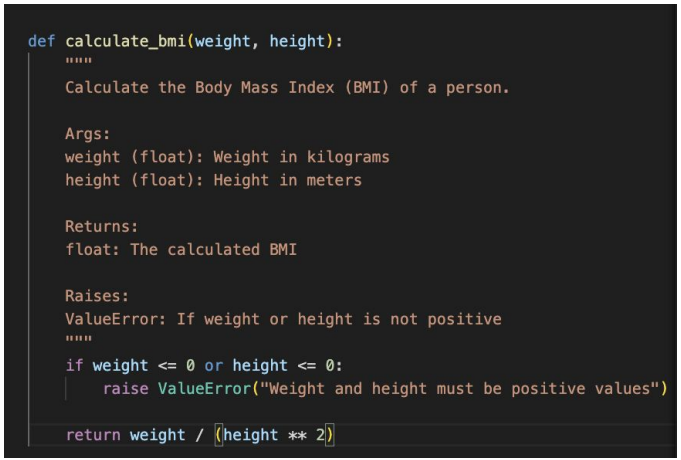
- Modularize your code by writing functions

```
def my_function(arg1, arg2, arg3):
    result1= do stuff with arg1, arg2, arg3
    result2= do stuff with arg1, arg2, arg3
    return result1, result2
```

Call the function via:

```
result1, result2 = my_function(arg1=arg1, arg2=arg2, arg3=arg3)
```

- One function, one purpose



```python
def calculate_bmi(weight, height):
    """
    Calculate the Body Mass Index (BMI) of a person.

    Args:
    weight (float): Weight in kilograms
    height (float): Height in meters

    Returns:
    float: The calculated BMI

    Raises:
    ValueError: If weight or height is not positive
    """
    if weight <= 0 or height <= 0:
        raise ValueError("Weight and height must be positive values")

    return weight / (height ** 2)
```

Positional and keyword arguments in Python

Positional arguments:

- Values are assigned to parameters **in order**.
- The position matters.

```python
def greet(name, age):
    print(f"Hi, I'm {name} and I'm {age} years old.")

greet("Alice", 30)  # OK
greet(30, "Alice")  # Wrong order, wrong meaning
```

Keyword arguments:

- Parameters are specified by **name**, not position.
- You can pass arguments in any order

```python
greet(age=30, name="Alice")
```

Mixing positional and keyword arguments:

- Positional arguments come first, then keyword arguments

```python
greet("Alice", age=30)  # OK
greet(name="Alice", 30)  # ❌
```

You can set default values as part of the function definition.

```python
def greet(name, age=18):
    print(f"Hi, I'm {name} and I'm {age} years old.")

greet("Bob")           # age defaults to 18
greet("Carol", 25)     # age is 25
```

Good coding practices: Meaningful variable names

- Descriptive names for scripts and functions instead of code comments

```
# Bad
def calc(a, b):
    return a * b

# Good
def calculate_area(length, width):
    return length * width
```

Good coding practices: Code Comments

Rule 1: Comments should not duplicate the code.

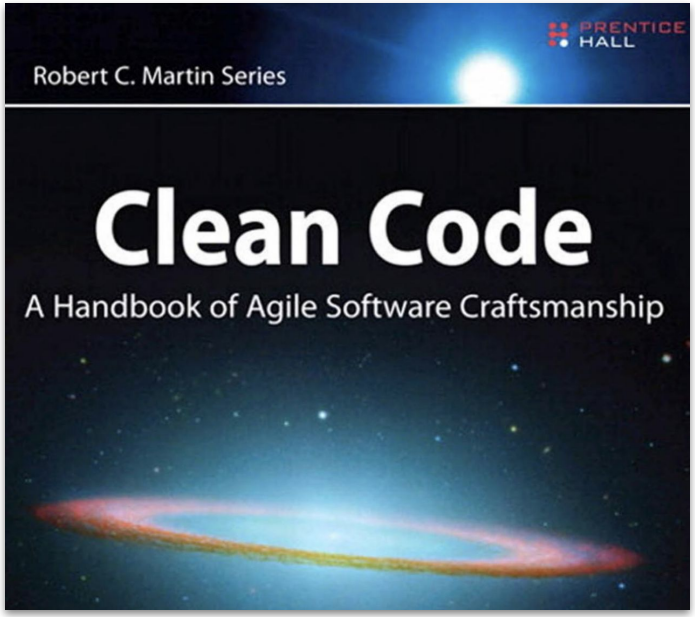
```
i = i + 1;          // Add one to i
```

Rule 2: Good comments do not excuse unclear code.

```
private static Node getBestChildNode(Node node) {
    Node n; // best child node candidate
    for (Node node: node.getChildren()) {
        // update n if the current state is better
        if (n == null || utility(node) > utility(n)) {
            n = node;
        }
    }
    return n;
}
```

Rule 3: If you can't write a clear comment, there may be a problem with the code.

```
 * You are not expected to understand this.
 */
if(rp->p_flag&SSWAP) {
        rp->p_flag =& ~SSWAP;
        aretu(u.u_ssav);
}
```



https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments/

Good coding practices: Code Comments

Rule 4: Provide links to the original source of copied code, and include links to external references

```
// Magical formula taken from a stackoverflow post, reputedly related to
// human vision perception.
return (int) (0.3 * red + 0.59 * green + 0.11 * blue);
```

Rule 5: Add comments when fixing bugs.

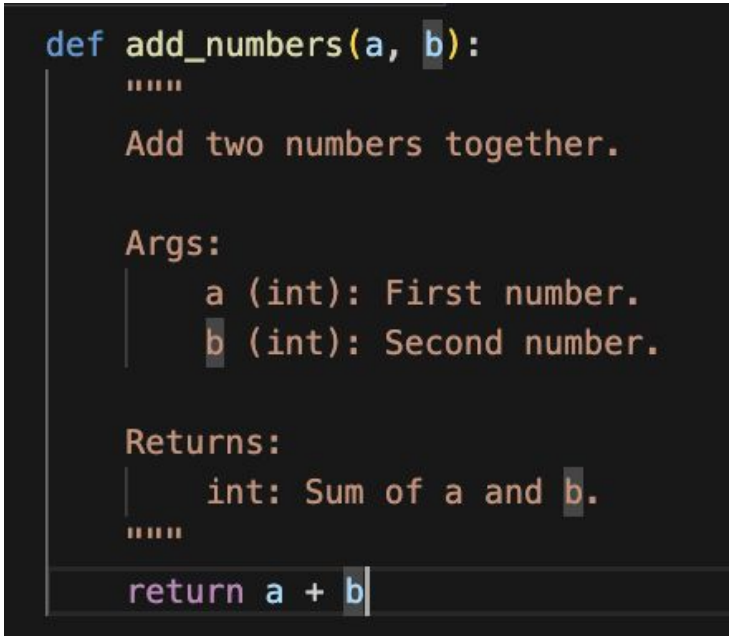
Rule 6: Use comments to mark incomplete implementations.

```
/ TODO(hal): We are making the decimal separator be a period,
/ regardless of the locale of the phone. We need to think about
/ how to allow comma as decimal separator, which will require
/ updating number parsing and other places that transform numbers
/ to strings, such as FormatAsDecimal
```

90% of all code comments:

Good coding practices - Documentation

- Write Docstrings
  - **Python**: Sphinx, pdoc
  - **JavaScript**: JSDoc
  - **C++**: Doxygen



Docstrings:

- Summary of the purpose of the function
- What arguments does the function take?
- Type specification?
- What does the function return?

Create docstrings in Python and R

```python
def calculate_bmi(weight, height):
    """
    Calculate the Body Mass Index (BMI) of a person.

    Args:
    weight (float): Weight in kilograms
    height (float): Height in meters

    Returns:
    float: The calculated BMI

    Raises:
    ValueError: If weight or height is not positive
    """
    if weight <= 0 or height <= 0:
        raise ValueError("Weight and height must be positive values")

    return weight / (height ** 2)
```

```r
#' Add together two numbers
#'
#' @param x A number.
#' @param y A number.
#' @returns A numeric vector.
#' @examples
#' add(1, 1)
#' add(10, 1)
add <- function(x, y) {
  x + y
}
```

Add together two numbers

Description

Add together two numbers

Usage

```
add(x, y)
```

Arguments

`x`	A number
`y`	A number

Value

The sum of `x` and `y`

Examples

```
add(1, 1)
add(10, 1)
```

Good coding practices

Good coding practices: Readability

```python
# Less readable
def is_prime(n):
    return n > 1 and all(n % i for i in range(2, int(n**0.5) + 1))

# More readable
def is_prime(number):
    if number <= 1:
        return False
    for i in range(2, int(number**0.5) + 1):
        if number % i == 0:
            return False
    return True
```

Good coding practices: Error handling

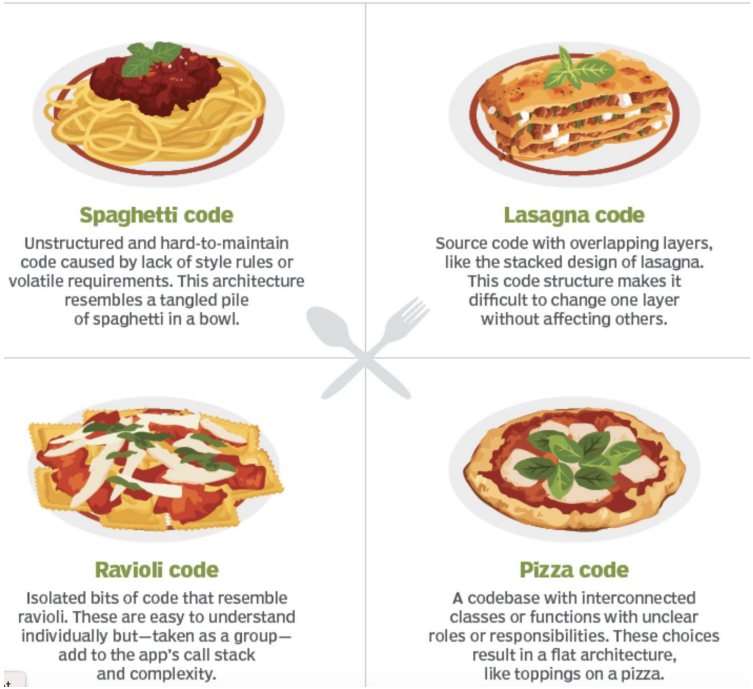
- Fail early and loudly
- Good error messages
  - Clearly pinpoints the problem.
  - Points a user to next steps: items to check, or other steps they can take to further understand or fix the problem.
  - Uses jargon appropriately, and keeps its target audience in
  - honest about what it knows and does not know.

Good coding practices: Single responsibility principle

```python
# Bad: Multiple responsibilities
class User:
    def __init__(self, name):
        self.name = name

    def save_to_database(self):
        # Database logic here
        pass

    def send_welcome_email(self):
        # Email logic here
        pass
```

```python
# Good: Separated responsibilities
class User:
    def __init__(self, name):
        self.name = name

class UserDatabase:
    def save_user(self, user):
        # Database logic here
        pass

class EmailService:
    def send_welcome_email(self, user):
        # Email logic here
        pass
```

Good coding practices: Software architecture



https://www.techtarget.com/searchsoftwarequality/tip/Fix-spaghetti-code-and-other-pasta-theory-antipatterns

Good coding practices - Testing

Research code is chronically undertested

- obstacle might lie in code structure

```python
import unittest

def add_numbers(a, b):
    return a + b

class TestAddNumbers(unittest.TestCase):
    def test_add_positive_numbers(self):
        self.assertEqual(add_numbers(2, 3), 5)

    def test_add_negative_numbers(self):
        self.assertEqual(add_numbers(-1, -1), -2)

    def test_add_zero(self):
        self.assertEqual(add_numbers(5, 0), 5)

if __name__ == '__main__':
    unittest.main()
```

Stabilizing the computing environment

Capturing & stabilizing your computational environment

What is reproduced? \ Interaction style	Graphical	Command line
Software & versions	binder	CONDA
Entire system		docker

https://book.the-turing-way.org/reproducible-research/renv/renv-options

Binder (https://mybinder.org/)

Binder is a service which generates fully-functioning versions of projects from a git repository and serves them on the cloud. These “binderized” projects can be accessed and interacted with by others via a web browser. In order to do this, Binder requires that the software (and, optionally, versions) required to run the project are specified. Users can make use of Package Management Systems or Dockerfiles to do this if they so desire.

How it works

- Enter you repo information
- Binder builds a docker image of it
- A JupyterHub will host your repo
- Comes with badge and link to share

Turn a Git repo into a collection of interactive notebooks

Have a repository full of Jupyter notebooks? With Binder, open those notebooks in an executable environment, making your code immediately reproducible by anyone, anywhere.

New to Binder? Get started with a Zero-to-Binder tutorial in Julia, Python, or R.

Build and launch a repository

GitHub repository name or URL

GitHub ▾ | GitHub repository name or URL

Git ref (branch, tag, or commit)

HEAD

Path to a notebook file (optional)

Path to a notebook file (optional) | File ▾

launch

Copy the URL below and share your Binder with others:

```
Fill in the fields to see a URL for sharing your Binder.
```

Expand to see the text below, paste it into your README to show a binder badge: launch binder

Computational reproducibility

Package management systems (conda, venv etc)

Package Management Systems are tools used to install and keep track of the software (and critically versions of software) used on a system and can export files specifying these required software packages/versions. The files can be shared with others who can use them to replicate the environment, either manually or via their Package Management Systems.

In a research context:

Allows to export and share environment settings via:

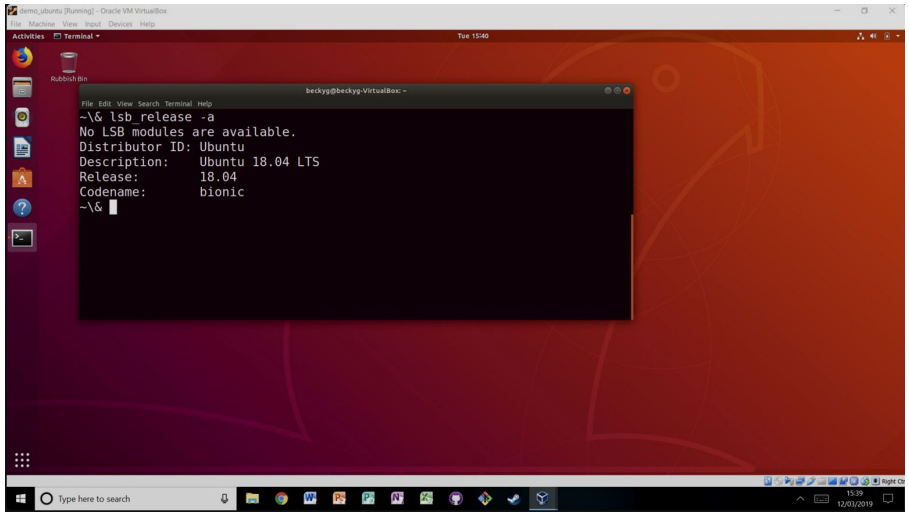
$conda env export > environment.yml

And re-create projects using:

$conda create --name Project_Two --clone Project_One

Virtual machines (for reproducible research)

VirtualBox allows to create virtual machines that can be shared exporting a single file. A second researcher can import the shared file and run an experiment in the same environment.



s://book.the-turing-way.org/reproducible-research/renv/renv-virtualmachine

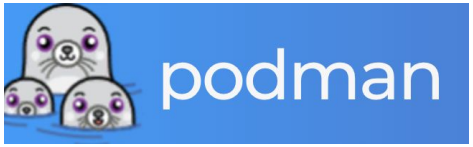
Virtual machines (for reproducible research)

Vagrant “enables users to create and configure lightweight, reproducible, and portable development environments”. In this context, an environment is a virtual machine (its CPUs, RAM, networking and so on) and the machines state (operating system, packages).

Vagrant can set up virtual machines **using text scripts,** instead of pointing and clicking through a graphical user interface. This makes it particularly useful for automating the process of setting up virtual machines and making that process reproducible.

Containers

- contain the individual components they need in order to operate the project they contain -> performance boost



Common container formats are:

- Docker (https://www.docker.com/)
- Podman (https://podman.io/)
- Singularity (Apptainer)

More on containers: https://book.the-turing-way.org/reproducible-research/renv/renv-containers

Singularity (Apptainer)

- Docker historically requires root access
- Singularity (which does not) as alternative on HPC systems
- The Open Source version **Apptainer** is a fork of Singularity and is run by the Linux foundation, after a change in licensing by Singularity.

Makefiles

- build automation tool
- Language agnostic - > bash commands
- Smart execution of pipelines, recognizes inputs and outputs
- Consists of a set of rules, such like this:

Makefile

```
targets: prerequisites
  command
  command
  command
```


https://makefiletutorial.com/

Makefiles

```
# Makefile

.PHONY: all clean

# Define filenames
RAW_DATA = data/student_habits_performance.csv
CLEANED_DATA = output/clean_data.csv
PLOT_IMAGE = output/plot.png

# Rule to create cleaned data
$(CLEANED_DATA): src/clean_data.py $(RAW_DATA)
    python3 src/clean_data.py $(RAW_DATA) $(CLEANED_DATA)

# Rule to create plot
$(PLOT_IMAGE): src/plot_data.py $(CLEANED_DATA)
    python3 src/plot_data.py $(CLEANED_DATA) $(PLOT_IMAGE)

# "all" target: runs both
all: $(PLOT_IMAGE)
```

PHONY: by default, makefiles check if the target file exists and only execute if not

- Targets under .PHONY are excluded from that rule

Good coding practices - automation

Snakemake (python)

- Allows to schedule scientific workflows, data analysis pipelines, especially in bioinformatics and computational science.
- Install using conda

```
rule clean_data:
    input: "raw_data.csv"
    output: "clean_data.csv"
    shell: "python clean_data.py {input} {output}"

rule plot:
    input: "clean_data.csv"
    output: "plot.png"
    shell: "python plot_data.py {input} {output}"
```

Activity 3 (40’)

Reproducible code

Work on Activity 3

Break (15’)

Thinking about the user

Documentation

Documentation != code comments

Many programming languages allow to write an pull Docstrings automatically to create documentation for you.

Example:

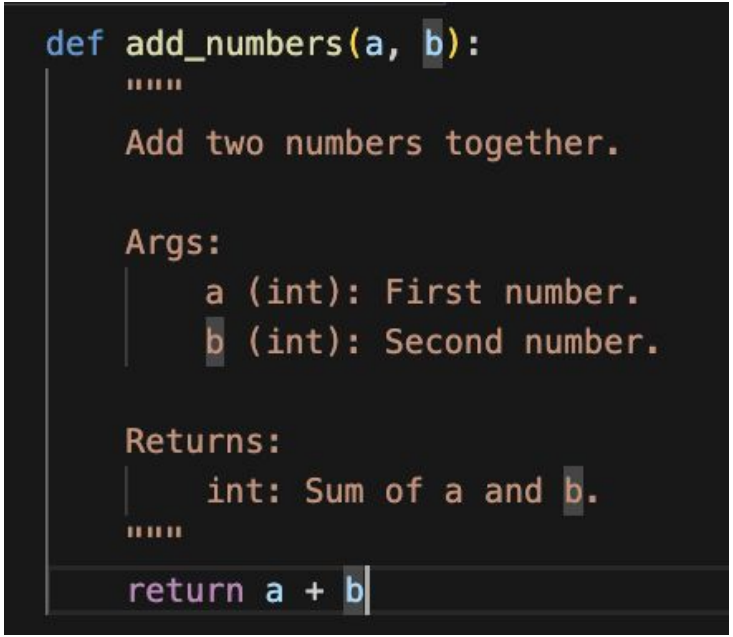
- Sphinx for Python
- Doxygen for C++/Fortran
- MATLAB’s built-in publishing

Writing docstrings - Python

- Help users (and yourself) understand what a function, class, or module does.
- Appear automatically in help systems (help(), IDEs, documentation tools).
- Essential for reproducibility and collaboration.

How to Write a Good Docstring

- Use triple quotes (""" """)
- Start with a short summary (one line)
- Args: define every input argument
- Returns: define what the function returns



Write docstrings: Matlab

- Docstrings start with % at the beginning of the function
- MATLAB automatically uses these comments when you call help functionname.

Example

```
function c = addme(a,b)
% ADDME  Add two values together.
%   C = ADDME(A) adds A to itself.
%
%   C = ADDME(A,B) adds A and B together.
%
%   See also SUM, PLUS.

switch nargin
    case 2
        c = a + b;
    case 1
        c = a + a;
    otherwise
        c = 0;
end
```

Documentation

https://nl.mathworks.com/help/matlab/matlab_prog/add-help-for-your-program.html

Documentation generators

Sphinx

- Sphinx is a documentation generator for Python (and other languages too).

- It reads your .py code files (if you want) and extracts docstrings to build API documentation (autodoc).
- It reads your manual .rst or .md files to create tutorials, guides, introductions, etc.
- It builds outputs like:
    - HTML websites
    - PDFs
    - ePub books

Potential: Host html files with ReadtheDocs

https://www.sphinx-doc.org/en/master/ https://about.readthedocs.com/

Packaging

Python:

- Can be versioned
- Can be uploaded to pip
- Can be (re) used by yourself and others
- A folder that contains an empty __init__.py file is considered a package.
- Functions can be imported using:

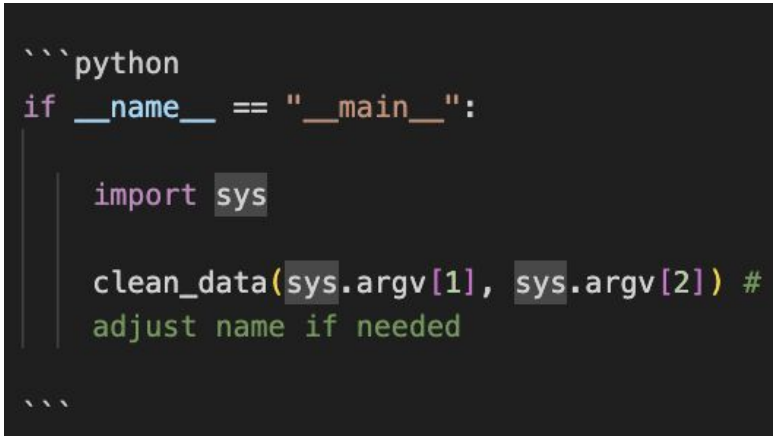
```
my_project/
├── src/
│   └── my_package/
│       ├── __init__.py
│       ├── clean_data.py
│       └── plot_data.py
├── data/
├── output/
├── README.md
├── Makefile
├── requirements.txt
└── setup.py (optional for installation)
```

```
from .clean_data import clean_dataset
from .plot_data import plot_study_hours
```

```
from my_package.clean_data import clean_dataset
```

Running python scripts from the command line

- Running functions from python scripts from the command line requires another “dunder” method, __main__
-
- This method (similar to other main methods) provides an entry point for execution and allows to run the file from the command line
-
- Additional arguments can be passed using the `sys` standard library.



https://realpython.com/python-magic-methods/

User interaction with your code

Files that help users to interact with your code:

- Contributing.md: How can users contribute?
- License.md: How can the code be re-used?
- Install information: How can the software be installed?
- User tutorials: Give examples on how to use the code.

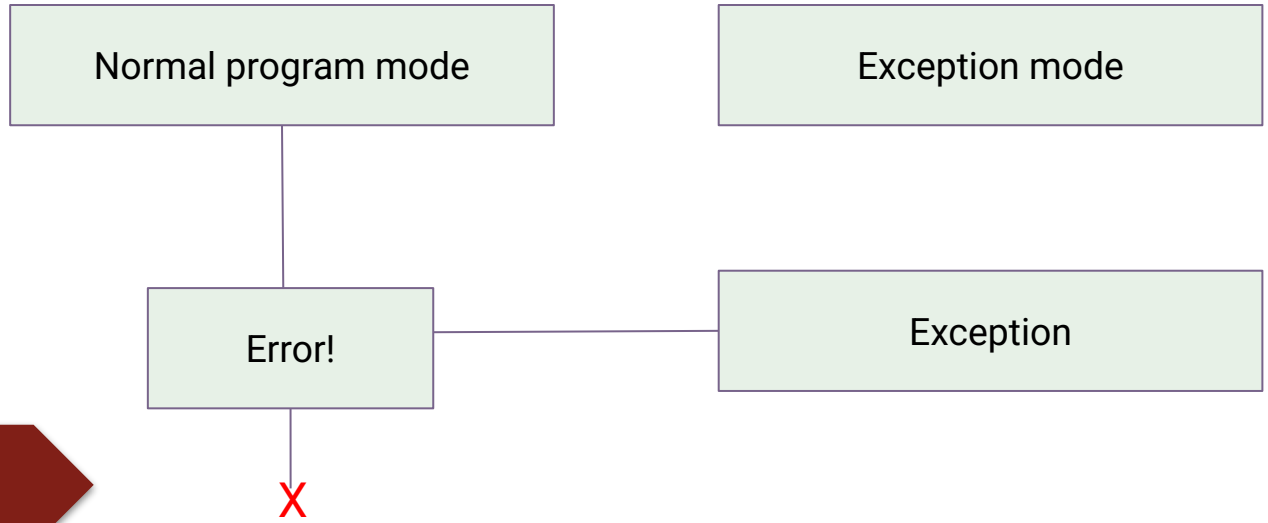
Example repositories:

Nilearn

PhysIO

Error handling

- The purpose of errors is to **halt/disrupt the program** if something goes wrong/some condition is not met.
- The programm switches from normal mode to error mode. All operations are being paused to handle the exception.



Error handling

Different types of errors:

- Syntax errors - > parser throws an error due to incorrect syntax
- Exceptions - > syntactically correct code results in an error

```
>>> print(0 / 0))
  File "<stdin>", line 1
    print(0 / 0))
                ^
SyntaxError: unmatched ')'
```

```
>>> print(0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```


https://realpython.com/python-exceptions/

Error handling

- In Python, exceptions follow the **raise Exception + Message** format:

```
number = 10
if number > 5:
    raise Exception(f"The number should not exceed 5. ({number=})")
print(number)
```

- To be more specific, python (and many other languages) have a variety of build in exception types

  You can write your own exceptions if you don't find a fitting one (requires OOP).

Error handling

- In MATLAB, exceptions are created via instances of Mexception objects that are created during the runtime of the program and consists of:
  - An errorID
  - An error Message

- The throw() statement is used to raise the corresponding exception

```
function indexIntoArray(A,idx)

% 1) Detect the error.
try
    A(idx)
catch

    % 2) Construct an MException object to represent the error.
    errID = 'MYFUN:BadIndex';
    msg = 'Unable to index into array.';
    baseException = MException(errID,msg);

    % 3) Store any information contributing to the error.
    if nargin < 2
        causeException = MException('MATLAB:notEnoughInputs','Not enough input arguments.');
        baseException = addCause(baseException,causeException);

        % 4) Suggest a correction, if possible.
        if(nargin > 1)
            exceptionCorrection = matlab.lang.correction.AppendArgumentsCorrection('1');
            baseException = baseException.addCorrection(exceptionCorrection);
        end

        throw(baseException);
    end
```

```
Error using indexIntoArray
Unable to index into array.

Caused by:
    Error using assert
    Indexing array is not numeric.
    Indexing array is too large.
```


https://nl.mathworks.com/help/matlab/matlab_prog/throw-an-exception.html

Good error messages

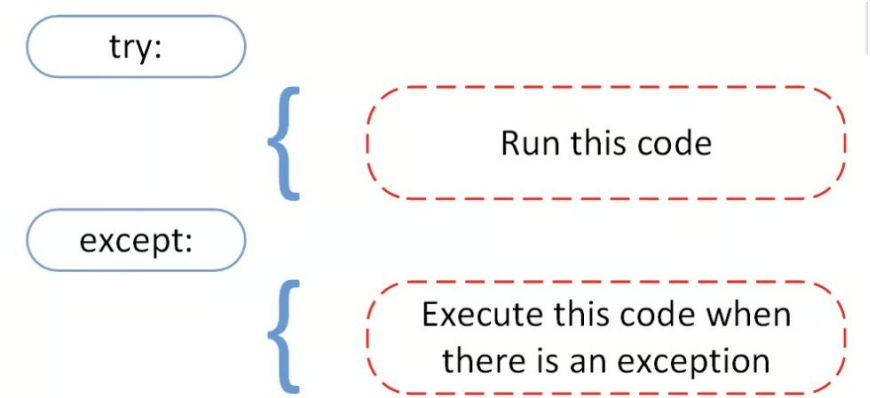
What makes a good error message?

- Clear: Say **what went wrong**
- Specific: Say **where** it went wrong
- Helpful: Suggest **what to do next**, if possible
- Polite: Don’t blame the user

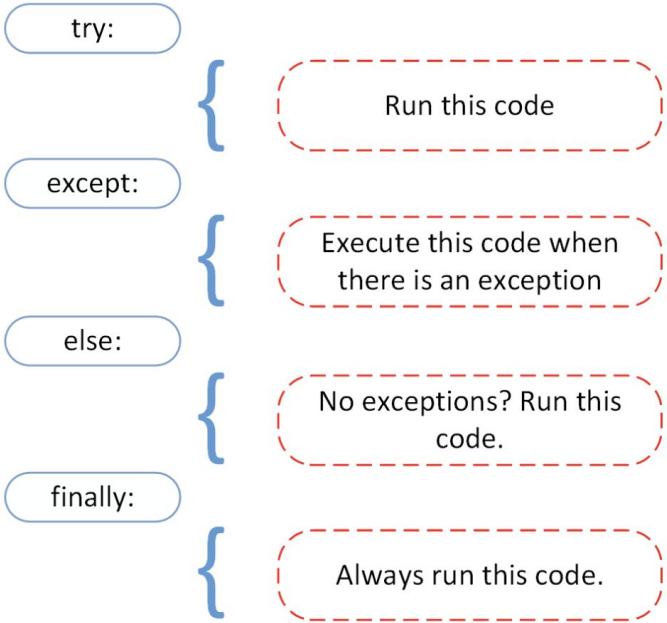
https://community.spiceworks.com/t/whats-the-funniest-error-message-you-have-seen/735421/39

Error handling

Exceptions can be wrapped into try - except blogs:



Error handling




```
# ...

try:
    linux_interaction()
except RuntimeError as error:
    print(error)
else:
    try:
        with open("file.log") as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print("Cleaning up, irrespective of any exceptions.")
```

Debugging

Debugging = the process of identifying, isolating, and fixing errors (bugs) in your code.

In Jupyter/Colab:

- %debug after an exception opens a post-mortem debugger.
- %pdb on automatically triggers the debugger on errors.

In IDEs:

- VS Code, Matlab have full visual debugging support (breakpoints, step into, watch variables)

Debugging

Breakpoints:

- Mark locations where code should be stopped. Program execution is halted, the current state of all variables can be observed (watch variables)

Step into:

- Allows to sequentially progress with the program (for example, step into the next function)

Discussion 2:

What pain points have you experienced when trying to (re)-use software?

WRAP UP