Back to Part 1

Table of Contents

3. Software design

Different things can be meant by the term "software design":

- **algorithm design** - what methods are we going to use in our software to meet the software's requirements?
- **software architecture** - what components will it have and how will they cooperate/interact?
- **system architecture** - what other things will this software have to interact with and how will it do this?
    - getting software and particularly system architecture right requires to address technical and other organisational challenges/requirements *in conjunction* - an interesting problem space!
- **UI/UX (user interface / user experience)** - how will users interact with the software?

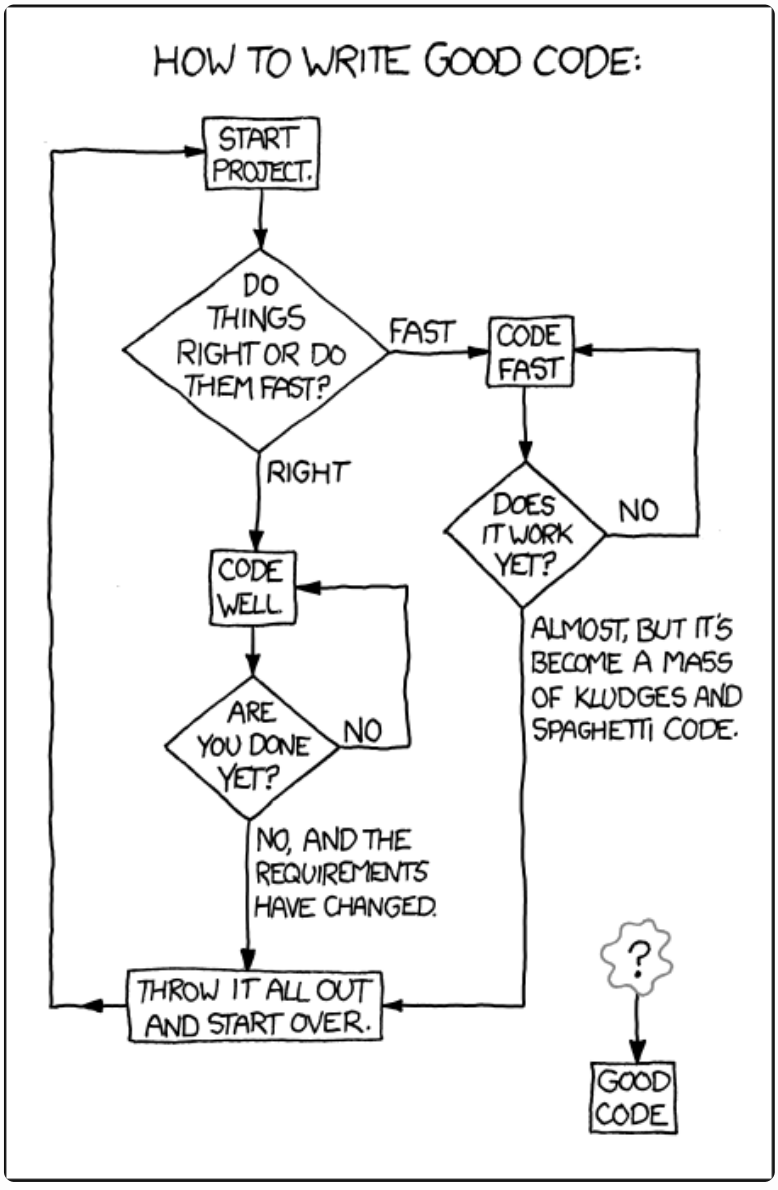
Design patterns are typical solutions to commonly occurring problems in software design (from any of the domains/levels mentioned above). From Refactoring Guru:

- "You can't just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.
- Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. [...] An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you."

Programming paradigms such as **object-oriented** or **functional programming** (we'll get to those in a minute!) are not so straightforward to allocate w. r. t. the different facets of software design mentioned above: a programming paradigm represents an own way of thinking about and structuring code, with pros and cons when used to solve particular types of problems.

Technical debt: If we don't follow best practices around code, including addressing design questions, we may build up too much technical debt - the cost of refactoring code due to having chosen a quick-and-dirty solution instead of having used a better approach that would have initially taken longer.

- It is normal to accumulate technical debt in a software or code-based project to some degree, however, it can (and often does) go overboard: bad, but quick/easy solutions at the start may make the software too messy and too difficult to understand and maintain, thereby hampering its further development.
- We want to write code such that we are able to respond well to changing requirements in the future - because requirements will change for sure (as in life generally, the only constant is *change*), so we want to design our software to be easily *modifiable* and *extensible*.
- There's of course a trade-off/tension between the time available and the quality of our work: How much effort should we spend on designing our code properly and using good development practices? Look for the solution in this XKCD comic:



3.1 Programming paradigms

There are two major families that we can group the common programming paradigms into: **Imperative** and **Declarative**.

- **Imperative programming** prescribes how a program operates *step by step* via a set of explicit instructions.
    - While it executes those steps, it may also produce so-called **side effects** - modifications of some state variable values outside the local environment, or, in other words, the production of observable effects other than the program's primary effect of returning a value to the invoker of the operation.
    - **Procedural programming** falls under imperative programming and is the programming paradigm you're probably most familiar with: it uses lists of instructions that are executed one after the other starting from the top. In this way, code is grouped into *procedures*, or, as we normally call them, *functions* performing a single task, with exactly one entry and one exit point.
    - **Object-oriented programming** is often classified as an extension of the Imperative family of programming paradigms (with the extra feature being the objects one is dealing with), but opinions differ.
        - In Object Oriented Programming, we represent the data and the things we want to do with it as **objects** which have specific **properties** and **behaviours**.
        - As an example, if we're writing a simulation for our physics research, we're probably going to need to represent atoms. Any atom can be characterized by the mass and electric charge, so we can build an object structure that includes mass and electric charge as properties. We can also specify what to do with those properties, e.g., we might want to add two masses of atoms. Moreover, multiple atoms can make up a molecule which may be modelled as a separate object. In that case, we can also specify what the relationship between atoms and molecules is using object-oriented programming.
- **Declarative programming** prescribes what data processing should happen, i.e., *what* the outcome is supposed to be rather than *how* it is achieved (as in imperative programming). In other words, a declarative program expresses the logic of a computation in terms of what should be accomplished rather than in terms of its control flow as an explicit sequence steps.
    - **Functional Programming** falls under declarative programming:
        - It is illuminative of the distinction between *code* and *data*, as in Functional Programming, a function can accept and transform other functions - code *is* data.
        - Side effect are avoided wherever possible.
        - Functional Programming is very advantageous w. r. t. **Big Data** where we can't move the data around easily, and, instead, aim to send our code to where the data is.
        - It's also advantageous w. r. t. running operations in parallel, as each operation is guaranteed to *no* interact with other operations.

    - However, within the research context and apart from Big Data, functional programming will rarely be clearly advantageous - but it's still useful/interesting for you to know about it.

We will look into two major paradigms from the imperative and declarative families that may be useful to you - **functional programming** and **object-oriented programming**.

- Most of modern languages can be used with multiple paradigms, and single programs often uses multiple ones.
- Python is a multi-paradigm and multi-purpose programming language. Procedural, object-oriented and functional programming all work well. However, as all its core data types (strings, integers, floats, booleans, lists, sets, arrays, tuples, dictionaries) as well as functions, modules and classes are objects, it does naturally lend itself to an object-oriented approach.

3.2 Object-oriented programming

In object-oriented programming, objects encapsulate data in the form of attributes and code in the form of methods that manipulate the objects' attributes and define how objects can behave (in interaction with each other).

A class is a template for a structure and a set of permissible behaviors that we want our data to comply to, thus, each time we create some data using a class, we can be certain that it has the same structure.

If you know about Python lists and dictionaries, you may recognize that they behave similarly to how we may define a class ourselves:

- they each hold some data (attributes),
- they provide some methods that describe how the data is supposed to behave (e.g., Lists can be appended to, indexed, sliced, and in dictionaries, key-value pairs can be added etc.)

Encapsulating data
Let's have a look at a simple class:

```
class Patient:
    def __init__(self, name):
        self.name = name
        self.observations = []

Alice = Patient('Alice')
print(Alice.name)
```

Output:

```
Alice
```

- We start defining a class with `__init__` - the initialiser method which sets up the initial values and structure of the data inside a new instance of the class. We call the `__init__` method every time we create a new instance of the class, as in Patient('Alice'). The argument `self` refers to the instance on which we are calling the method and gets filled in automatically by Python whenever we instantiate a new class instance.
- We encapsulate the patient's name and a list of inflammation observations as data/attributes, either by providing values for those as arguments when creating a new class instance, or by setting those values in the initialiser method.
    - In the example, we set a patient's name ('Alice') to the value provided when creating a new class instance (here 'Alice'), and create a list of inflammation observations for the patient (initially empty).
- We can access the encapsulated data by calling the attribute alongside the class instance using the dot (as in `Alice.name`).

Encapsulating behavior

Let's add a method to the above class which operates on the data that the class contains: adding a new observation to a Patient instance.

```
class Patient:
    """A patient in an inflammation study."""
    def __init__(self, name):
        self.name = name
        self.observations = []

    def add_observation(self, value, day=None):
        if day is None:
            try:
                day = self.observations[-1]['day'] + 1

            except IndexError:
                day = 0

        new_observation = {
            'day': day,
            'value': value,
        }

        self.observations.append(new_observation)
        return new_observation

Alice = Patient('Alice')
print(Alice)

observation = Alice.add_observation(3)
print(observation)
print(Alice.observations)
```

Output:

```
<__main__.Patient object at 0x7f67f424c190>
{'day': 0, 'value': 3}
[{'day': 0, 'value': 3}]
```

- Methods on classes are the same as normal functions, except that they live inside a class and have an extra first parameter `self` (using this name is not strictly necessary, but is a very strong convention). Similar to the initialiser method, when we call a method on an object, the value of self is automatically set to this object - hence the name.
- We can use the encapsulated method by calling it alongside the class instance using the dot (as in `Alice.add_observation(3)`).

Dunder Methods

The `__init__` method begins and ends with a double-underscore - it is a dunder method. These dunder methods (also called magic methods) are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action. Built-in classes such in Python as the `int` class define many magic methods.

- When we called `print(Alice)`, it returned `<__main__.Patient object at 0x7fd7e61b73d0>` which is the string represenation of the Alice object. Functions like `print()` or `str()` use `__str__()`.
- However, we can override the `__str__` method within our class to display the object's name instead of the object's string representation.

```
class Patient:
    """A patient in an inflammation study."""
    def __init__(self, name):
        self.name = name
        self.observations = []

    def add_observation(self, value, day=None):
        if day is None:
            try:
                day = self.observations[-1]['day'] + 1

            except IndexError:
                day = 0

        new_observation = {
            'day': day,
            'value': value,
        }

        self.observations.append(new_observation)
        return new_observation

    def __str__(self):
        return self.name

Alice = Patient('Alice')
print(Alice)
```

Output:

```
Alice
```

Relationships between classes

There are two fundamental types of object characteristics which also denote the relationships among classes:

- ownership - x *has* a y - this is composition,
- identity - x *is* a y - this is inheritance.

Composition

In object oriented programming, we can make things components of other things, e.g., we may want to say that a doctor *has* patients or that a patient *has* observations. In the way

we had written our class so far, a patient already has observations - which is a case of composition.

Let's separate the two and make an own Observation class, and make use of it in the Patient Class.

```
class Observation:
    def __init__(self, day, value):
        self.day = day
        self.value = value

    def __str__(self):
        return str(self.value)

class Patient:
    """A patient in an inflammation study."""
    def __init__(self, name):
        self.name = name
        self.observations = []

    def add_observation(self, value, day=None):
        if day is None:
            try:
                day = self.observations[-1].day + 1

            except IndexError:
                day = 0

        new_observation = Observation(day, value)

        self.observations.append(new_observation)
        return new_observation

    def __str__(self):
        return self.name


Alice = Patient('Alice')
obs = Alice.add_observation(3, 3)

print(obs)
```

Output:

```
3
```

Inheritance

Inheritance is about data and behaviour that two or more classes share: if class X inherits from (is a) class Y, we say that Y is the *superclass* or *parent class* of X, or X is a *subclass* of Y - X gets all attributes and methods of Y.

If we want to extend the previous example to also manage people who aren't patients we can add another class Person. But Person will share some data and behaviour with Patient - in this case both have a name and show that name when you print them. Since we expect all patients to be people (hopefully!), it makes sense to implement the behaviour in Person and then reuse it in Patient.

To write our class in Python, we used the class keyword, the name of the class, and then a block of the functions that belong to it. If the class inherits from another class, we include the parent class name in brackets.

```
class Observation:
    def __init__(self, day, value):
        self.day = day
        self.value = value

    def __str__(self):
        return str(self.value)

class Person:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

class Patient(Person):
    """A patient in an inflammation study."""
    def __init__(self, name):
        super().__init__(name)
        self.observations = []

    def add_observation(self, value, day=None):
        if day is None:
            try:
                day = self.observations[-1].day + 1

            except IndexError:
                day = 0

        new_observation = Observation(day, value)

        self.observations.append(new_observation)
        return new_observation
```

- To make the Patient class inherit the Person class, we need to indicate the parent class in the class name (`class Patient(Person)`), as well as in the initialiser (`super().__init__(name)`).
    - If we don't define a new `__init__` method for our subclass, Python will look for one on the parent class and use it automatically). This is true of all methods - if we

call a method which doesn't exist directly on our class, Python will search for it among the parent classes.

- `self.name = name` in the Patient class becomes obsolete.

> **! QUESTION 1 ! What outputs do you expect here?**

```
Alice = Patient('Alice')
print(Alice)

obs = Alice.add_observation(3)
print(obs)

Bob = Person('Bob')
print(Bob)

obs = Bob.add_observation(4)
print(obs)
```

> **! TASK 4 ! Write a Doctor class to hold the data representing a single doctor:**

- It should have a name attribute, as well as a list of patients that this doctor is responsible for.
- If you have the time, write corresponding tests in `test_patient.py`.

Final note: When deciding how to implement a model of your particular system, you often have a choice of either composition or inheritance, where there is no obviously correct choice - multiple implementations may be equally good. (See more on that in the The Composition Over Inheritance Principle.

Key points and benefits of OOP:

- Use OOP when working with data that has a complex structure and dynamic behavior.
- Modular code that can be easily and safely be extended upon, especially when combined with tests.
- Clearly exposed classes and endpoints should be easier to use for other researcher and developer that might be using the codebase later. The higher the ease of use, the more likely our work is to be used by other and thus cited.

3.3 Functional programming

In functional programming, programs apply and compose/chain functions. It is based on the mathematical definition of a function `f()` which does a transformation/mapping from input `x` to output `f(x)`).

Contrary to imperative paradigms, it does not entail a sequence of steps during which the state of the code is updated to reach a final desired state. It describes the transformations to be done without producing such side effects.

The following two code examples implement the calculation of a factorial in procedural and functional styles, respectively. The factorial of a number `n` (denoted by `n!`) is calculated as the product of integer numbers from 1 to n.

Procedural style factorial function

```
def factorial(n):
    """Calculate the factorial of a given number.

    :param int n: The factorial to calculate
    :return: The resultant factorial
    """
    if n < 0:
        raise ValueError('Only use non-negative integers.')

    factorial = 1
    for i in range(1, n + 1): # iterate from 1 to n
        # save intermediate value to use in the next iteration
        factorial = factorial * i

    return factorial
```

- In the function, we have a list of instructions to change the state of the program (e.g., the variable `factorial` in the for loop) and advance towards the result.

Functional style factorial function

```
def factorial(n):
    """Calculate the factorial of a given number.

    :param int n: The factorial to calculate
    :return: The resultant factorial
    """
    if n < 0:
        raise ValueError('Only use non-negative integers.')

    if n == 0 or n == 1:
        return 1 # exit from recursion, prevents infinite loops
    else:
        return  n * factorial(n-1) # recursive call to the same function
```

- In the function, we don't update any program states (as with the variable `factorial` in the above example), or modify data that exists outside the current function, including the input data (e.g., printing text, writing to a file, modifying the value of an input argument, or changing the value of a global variable).

    - Functional computations only rely on the values that are provided as inputs to a function which is also referred to as the **immutability of data**.
    - Such functions do not create any **side effects**, i.e. do not perform any action that affects anything other than the value they return.
    - Functions without side effects that return the same data each time the same input arguments are provided are called **pure functions**.
- Rather than using iteration to repeat a series of steps as in procedural programming, functional programming typically uses recursion, i.e., it calls/repeats itself until a particular condition is reached.
- **Trade-offs:** the Fibonacci function is a good illustrations of trade-offs between different paradigms. You will find that the cost in time of the functional implementation rises exponentially w.r.t the value of `n`, while the procedural impl. runs faster. It is vital to consider your use case before chosing which kind of paradgim to use for your software.

! QUESTION 2 ! Which of these functions are pure?

```
def add_one(x):
    return x + 1

def say_hello(name):
    print('Hello', name)

def append_item_1(a_list, item):
    a_list += [item]
    return a_list

def append_item_2(a_list, item):
    result = a_list + [item]
    return result
```

3.3.1 MapReduce Data Processing Approach

- This approach relies heavily on composability and parallelisability of functional programming.
- A quick ethymology of **MapReduce**:
    - **Map**: aplly an operation (function) to each value of a dataset
    - **Reduce**: collect/aggregate individual data points / results to produce a singla result.
- This technique is widely use not only for processing big data, but also distributed computing (HPC: High-Performance Computing -> super computers, clusters) or more generally, large scale data processing.

3.3.1.1 Mapping

General syntax: `map(f, C)`: apply function `f` to each element of the collection `C` and return the results as a new collection of the *same size*.

As an example, we are interested in getting the length of each name in the list `["Mary", "Isla", "Sam"]`:

▶ Show answer

Another example: let's write a one-liner that squares every number in the collection `C=[0, 1, 2, 3, 4]` using an anonymous `lambda` expression:

▶ Show answer

A quick note on Lambda expressions in Python

- Lambda expression allow us to create *anonymous function* that can be used to write more compact code
- General syntax: `lambda x, y, z, ... : <expression>`
  - Takes in any number of parameters `x, y, z ...`,
  - then returns the value of the `<expression>`
- It is the equivalent of the standard python function definition:

```
def f(x, y, z, ...):
    return <expression>
```

! TASK 6 ! Check Inflammation Patient Data Against A Threshold Using Map:

Write a new function named `daily_above_threshold()` in our inflammation `models.py` that determines whether or not each daily inflammation value for a given patient exceeds a given threshold.

Given a patient row number in our data, the patient dataset itself, and a given threshold, write the function to use map() to generate and return a list of booleans, with each value representing whether or not the daily inflammation value for that patient exceeded the given threshold.

▶ Show solution

3.3.1.2 Reduce

Conversely, `reduce(f, C, initialiser)` functions taken in a function `f()`, a collection `C` of data items (and an optional `initialiser`), then returns a *single cumulative value* that aggregates all values in the collection.

- First, it applies `f()` to either a) the first two values in `C` or b)the `initialiser` and the first value in `C`.
- Then continues to apply `f()` to the result of the previous operation and the next element in `C`, until the whole list is exhausted.

```
f(f(f(f(C[0], C[1]), C[2]), C[3]), C[4])
# This leverages composability of functional programming
```

As an example, let's use `from functools import reduce` to compute the product of the sequence of numbers `l = [1, 2, 3, 4]`

▶ Show answer

! TASK 7 ! Calculate the sum of a sequence of numbers using reduce:

We aim to reproduce the behavior of Python's native `sum()` using `reduce`:

▶ Show answer

3.3.1.3 Putting it all together: MapReduce

Let us now write a function `sum_of_squares` that calculates the *sum of the squares of the values in a list* using the MapReduce approach:

▶ Show answer

Test it with

```
print(sum_of_squares([0]))
print(sum_of_squares([1]))
print(sum_of_squares([1, 2, 3]))
print(sum_of_squares([-1]))
print(sum_of_squares([-1, -2, -3]))
```

and confirm the follwoing input:

```
0
1
14
1
14
```

> **! TASK 8 ! Extend Inflammation Threshold Function Using Reduce**:

Extend the `daily_above_threshold()` function you wrote previously to return a count of the number of days a patient's inflammation is over the threshold.
Use `reduce()` over the boolean array that was previously returned to generate the count, then return that value from the function.

You may choose to define a separate function to pass to `reduce()`, or use an inline lambda expression to do it (which is a bit trickier!).

Some hints:

- You can define an `initialiser` value with `reduce()` to help you start the counter
- If defining a lambda expression, note that it can conditionally return different values using the syntax `<value> if <condition> else <another_value>` in the expression.

▶ Show answer

3.3.2 Decorators

As an example of composability, let's look at **Python decorators**: as seen in the episode on parametrising our unit tests, a decorator can take a function, modify/decorate it, then return the resulting function.
This is possible because in Python, functions can be passed around as normal data.
Here, we discuss decorators in more detail and learn how to write our own.
Let's look at the following code for ways on how to "decorate" functions.

```
# define function where additional functionality is to be added
def ordinary():
    print("I am an ordinary function")

# define decorator, or outer function for first function
def decorate(func):
    # define the inner function
    def inner():
        # add some additional behavior to original function
        print("I am a decorator")
        # call original function
        func()
    # return the inner function
    return inner

# decorate the ordinary function
decorated_func = decorate(ordinary)

# call the decorated function
decorated_func()
```

Output:

```
I am a decorator
I am an ordinary function
```

- `ordinary()` is to be decorated,
- `decorate(func)` is the function that decorates another function,
- calling `decorate(ordinary)` builds another function that adds functionality to `ordinary()`.

Another way to use decorators is to add @decorate before the function to be decorated:

```
# define decorator, or outer function for first function
def decorate(func):
    # define the inner function
    def inner():
        # add some additional behavior to original function
        print("I am a decorator")
        # call original function
        func()
    # return the inner function
    return inner

# define function where additional functionality is to be added
@decorate
def ordinary():
    print("I am an ordinary function")

# call the decorated function
ordinary()
```

Output:

```
I am a decorator
I am an ordinary function
```

> **! TASK 9 ! Write a decorator that measures the time time taken to execute a particular function** using the time.process_time_ns() function.

- You need to import `time`.
- To get a time stamp, you can simply write `start = time.process_time_ns()`, and get another time stamp once the calculation in question is done using `end = time.process_time_ns()`.
- Use this function to measure its execution time:

```
def measure_me(n):
    total = 0
    for i in range(n):
        total += i * i

    return total
```

▶ Example solution

4. Writing software with - and for - others: workflows on GitHub, APIs, and code packages

4.1 GitHub pull requests

Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch. **Code review** plays an essential role in this process.

Code review

Code review is one of the most important practices of collaborative software development that improves code quality and increases knowledge about the codebase across the team. Before contributions are merged into the main branch, code will need to be reviewed, e.g., by the maintainer(s) of the repository.

Although the role of code review can't be overstated, we will not go into the details here, as it's better suited for self-study compared to other building blocks in research software

engineering that we touch upon in this tutorial. See, e.g., a guide on code review from Kimmo Brunfeldt here.

Types of development models

The way you and your team provide contributions to the shared codebase depends on the type of development model you use in your project. Two commonly used models are the following:

- **shared repository model**: folks are granted push access to a single shared code repository, but feature branches for new developments are still created. This model is good for core contributors who may wish to have faster workflows in the testing and merging cycle.

- **fork and pull model**: folks fork an existing repository (to create their copy of the project linked to the source) and push changes to their personal fork. A contributor can therefore work independently on their own fork as they do not need permissions on the source repository to push modifications to their own fork. The project maintainer can then pull the contributors' changes into the source repository on request and after a code review process.
    - One advantage of this model is that it makes it easy for new contributors to join a project, without upfront coordination with source project maintainers. It may be well suitable for, e.g., external collaborators as opposed to, e.g., core team members.

> **! TASK 10 ! Making a PR to the original project, mock code review and discussion around the process.**

- Essentially commit all the changes that have been done so far into `develop`.
- Create a PR from Github's web interface
- Let's explore the PR as they come, mock a code review, and solve potential conflicts that arise.

▶ **Simple PR Task for new user**: Let's make a PR to add

> The Continuous Integration using Github Actions can be used at the PR level to make sure incoming code is adequate and passess the relevant tests.

4.2 How can others use the programs you write: application programming interfaces (API)

We will now have a look at `inflammation-analysis.py` which, in our example, is the entry point of our simple application - users will need to call it within a CLI, alongside a set of arguments:

```
python3 inflammation-analysis.py data/inflammation-03.csv
```

How to use the application and which arguments to specify can be accessed via

```
python3 inflammation-analysis.py --help
```

`inflammation-analysis.py` can be run in different ways - as an imported library, or as the top-level script in which case the global dunder variable `__name__` will be set to `"__main__"`.

Advanced API configuration tools

- Hydra Easily configure complex applications
- [Hydra] OmegaConf: YAML based hierarchical configuration system, with support for merging configurations from multiple sources (files, CLI argument, environment variables).
- custom parsers, etc...

Global variable `__name__`

In `inflammation-analysis.py`, we see the following code:

```
# import modules

def main():
    # perform some actions

if __name__ == "__main__":
    # perform some actions before main()
    main()
```

`__name__` is a special dunder variable which is set, along with a number of other special dunder variables, by the python interpreter before the execution of any code in the source file. What value is given by the interpreter to `__name__` is determined by the way in which it is loaded.

If you run the following command (i.e., run the file as a script), `__name__` will be equal to `__main__`, and everything following after the if-statement will be executed:

```
$ python3 inflammation-analysis.py
```

If you import your file as a module via `import inflammation-analysis`, `__name__` will be set to the module name, i.e., `__name__ = "inflammation-analysis"`.

In other words, the global variable `__name__` allows you to execute code when the file runs as a script, but not when it's imported as a module.

Python sets the global name of a module equal to `__main__` if the Python interpreter runs your code in the top-level code environment.

"Top-level code" is the first user-specified Python module that starts running. It's "top-level" because it imports all other modules that the program needs.

Command-line options

To be able to run `inflammation-analysis.py` in the CLI, we need to enable Python to read command line arguments. The standard Python library for reading command line arguments passed to a script is `argparse`. Let's look into `inflammation-analysis.py` again.

```
# we first initialise the argument parser class,
# passing an (optional) description of the program:
parser = argparse.ArgumentParser(
    description='A basic patient inflammation data
 system')

# we can now add the arguments that we want argparse
# to look out for; on our case, we only want to process
# the names of the file(s):
parser.add_argument(
    'infiles',
    nargs='+',
    help='Input CSV(s) containing inflammation series for each patient')

# we parse the arguments passed to the script:
args = parser.parse_args()
```

- We have defined what the argument will be called (`infiles`), the number of arguments to be expected (nargs='+', where '+' indicates that there should be 1 or more arguments passed); and a help string for the user (`help='Input CSV(s) containing inflammation series for each patient'`).
- You can add as many arguments as you wish, and these can be either mandatory (as the one above) or optional.
- `parser.parse_args()` returns an object (called `arg`) containing all the arguments requested. These can be accessed using the names that we have defined for each argument, e.g., `args.infiles` would return the filenames that were used as inputs.
- When we run, e.g., `python3 inflammation-analysis.py data/inflammation-03.csv`, nothing will happen at that point, as `views.py` used `matplotlib`, but a our CLI will

output only text. But we could add another modality in `views.py` to be able to generate output that is shown in the CLI.

4.3 Producing a code package

We will now look at how we can package software for release and distribution, using `Poetry` to manage our Python dependencies and produce a code package we can use with a Python package indexing service such as PyPi.

Preparing software for release

Here, we only marginally touch upon important factors to consider before publishing software, most of which have to do with documentation. Documentation is a foundational pillar in coding/writing software. While its significance can't be overstated, we omit this part in this tutorial, as it's better for self-study compared to other building blocks in research software engineering.

Documentation

Before releasing software for reuse, make sure you have

- documented your code sufficiently (see, e.g., this blog post: What are best practices for research software documentation?),
- included essentials such as a README and a LICENSE file. A README may include the following:
    - **installation/deployment**: step-by-step instructions for setting up the software so it can be used,
    - **basic usage**: step-by-step instructions that cover using the software to accomplish basic tasks,
    - **contributing**: for those wishing to contribute to the software's development, this is an opportunity to detail what kinds of contribution are sought and how to get involved,
    - **contact information/getting help**: which may include things like key author email addresses, and links to mailing lists and other resources,
    - **credits/acknowledgements**: where appropriate, be sure to credit those who have helped in the software's development or inspired it,
    - **citation**: particularly for academic software, it's a very good idea to specify a reference to an appropriate academic publication so other academics can cite use of the software in their own publications and media. You can do this within a separate CITATION text file within the repository's root directory and link to it from the Markdown.

Marking a software release

There are different ways in which we can make a software release from our code in Git/on

GitHub, one of which is **tagging**: we attach a human-readable label to a specific commit, e.g., "v1.0.0", and push the change to our remote repo:

> **! FOLLOW ALONG IN YOUR CODESPACE !**

```
$ git tag -a v1.0.0 -m "Version 1.0.0"
$ git push origin v1.0.0
```

Packaging up software

We will use Python's `Poetry` library which we'll install in our virtual environment (make sure you're in the root directory when avtivating the virtual environment, and let's check afterwards that we installed `Poetry` within it):

> **! FOLLOW ALONG IN YOUR CODESPACE !**

```
$ source venv/bin/activate # if not already
$ pip3 install poetry
$ which poetry
```

Poetry uses a `pyproject.toml` file to describe the build system and requirements of the distributable package.

- In the context of software development, *build* is the process of “translating” source code files into executable binary code files that can be run directly.
- A *build system* is a collection of software tools that is used to facilitate the build process.

To create a `pyproject.toml` file for our code, we can use `poetry init` which will guide us through the most important settings (for each prompt, we either enter our data or accept the default).

Below, you see the questions with the recommended responses, so do follow these (and use your own contact details).

- We’ve called our package “inflammation” instead of “inflammation-analysis” to match the name of our module package, so that `Poetry` can automatically find the code.

```
$ poetry init
```

Output:

```
This command will guide you through creating your pyproject.toml config.

Package name [example]:  inflammation
Version [0.1.0]: 1.0.0
Description []:  Analyse patient inflammation data
Author [None, n to skip]: Nadine Spychala <nadine.spychala@gmail.com>
License []:  MIT
Compatible Python versions [^3.8]: ^3.8

Would you like to define your main dependencies interactively? (yes/no) [yes] no
Would you like to define your development dependencies interactively? (yes/no) [
Generated file

[tool.poetry]
name = "inflammation"
version = "1.0.0"
description = "Analyse patient inflammation data"
authors = ["Nadine Spychala <nadine.spychala@gmail.com>"]
license = "MIT"

[tool.poetry.dependencies]
python = "^3.8"

[tool.poetry.dev-dependencies]

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"

Do you confirm generation? (yes/no) [yes] yes
```

When we add a dependency using `Poetry`, Poetry will add it to the list of dependencies in the `pyproject.toml` file, and automatically install the package into our virtual environment.

- There are two different types of dependency: *runtime dependencies* and *development dependencies*. The former are those dependencies that need to be installed for our code to run, like `NumPy`. The latter are dependencies which are needed/essential in order to develop code, but not required to run it, e.g., `pylint` or `pytest`.

```
$ poetry add matplotlib numpy
$ poetry add --dev pylint
$ poetry install
```

Let's build a distributable version of our software:

```
$ poetry build
```

This should produce two files for us in the `dist` directory of which the most important one is the `.whl` or *wheel* file. This is the file that `pip` uses to distribute and install

Python packages, so this is the file we'd need to share with other people who want to install our software.

If we gave this wheel file to someone else, they could install it using `pip`:

```
$ pip3 install dist/inflammation*.whl
```

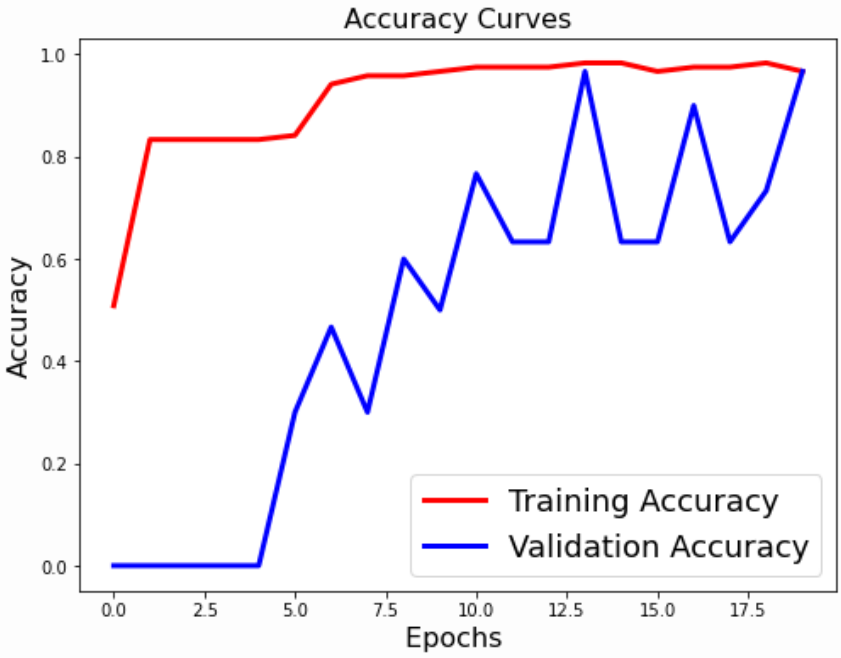
If we need to publish an update, we just update the version number in the `pyproject.toml` file, then use `Poetry` to build and publish the new version. Any re-publishing of the package, no matter how small the changes, needs to come with a new version number.

5. Collaborating research and sharing experiment results

So far, we have seen general principles for collaborative research software design.
In deep learning and its tangential fields, we usually train models, log various metrics along the way, which are then used to analyze the performance and support the claims made in the sibling paper.
This section aims to give a brief overview of different ways to handling experiment results and analyzing them, culminating into a set of principles and tools for collaborative experiment management and analysis.

5.1 Experiment analysis: custom training curve plotting

One way to analyze the results it to have manually designed plotting functions.



While it allows us a very precise control on the type of graphs and analysis tool we can deploy, it comes at a relatively higher cost in engineering, and is relatively rigid.

Some caveats

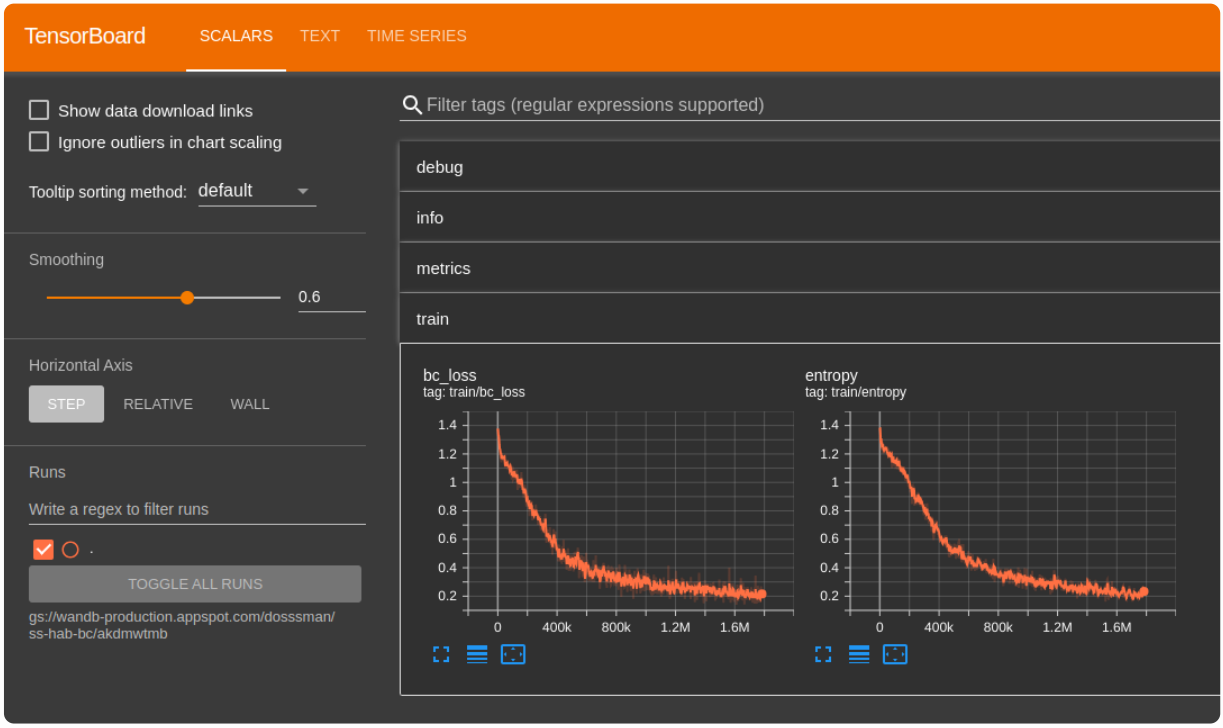
- What if a collaborator wants to analyze from a different perspective, but is not familiar enough with the structure of the code to do a custom plots ? Or not familiar with visualization tools in general ?
- The reviewers of our papers might be interested in some specific experiment results that are not featured in the paper nor supplementary plots ?
- What a new collaborator or someone who tries to build on top of our published code needs access to detailed results for their own plots ?

When publishing code and a paper, we should ideal also publish the datasets and raw experiments runs that were used in the paper **for better reproducibility**.

5.2 Tensorboard: a standard for training metrics plotting and export

Tensorboard is a visualization toolkit tailed for deep learning experiments.
Its main use is to track various metrics during the training of deep learning models.
It also provides a set of graphical user interfaces for quick assessment of the logged metrics and comparison between runs.

You are probably already familiar with this interface.



While Tensorboard might be easy to setup and use, depending on the type of experiment one runs, it can quickly become lacking for more advanced analysis.

- This training metrics can be shared either in Tensorboard's native format, or exported into even more general formats (CSV, JSON) for further analysis.
- This can be easily shared with collaborators for their own, independent analysis.

    - Either we share the files around and eveyrone manually opens it
    - We set up a local webserver that can be accessed by the team for analysis (limited security though)
- A researcher that builds on top of our published code can use them to report baselines in their own work.
- A dedicated review could make the effort of inspecting those for additional analysis and confirm the soundness of the result in a published paper.

However, the analysis tools of Tensorboard can be limited depending on one's use case and the type of analysis that needs to be perform.
Anything more advanced would thus require a fall back on custom analysis pipeline.

5.3 DL Experiment Tracking and Management as a tool for collaborative research and result sharing

Nowadays, there exist tools that handle ML experiment tracking like Tensorboard, while offering a lot of flexibility for plotting and analysis.
The prominent ones at the time of this writing are as follows:

- Neptune.ai
- Weight and Biases (WANDB)
- Comet ML

A more exhaustive list on neptune.ai site with comparison.

Let's walk though a few of the benefits of using those tools for collaborative analysis and publishing results, with an emphasis on WANDB (disclaimer: **not** a sponsored advertisment).

Benefits

- Cloud based tracking allows access from anywhere (although it is worth considering security implications.)
- Intuitive interface to quickly create plots and inspect training process Demo 1.
    - For a single experiment, plot "bc_loss", and "success_rate" together, for example.
- Aggregation over seeds and comparison of experiment variants. [Demo 2]
    - Assume we have two experiment which differ in the value of an arbitrary hyperparameter,
    - and each experiment was run on two seeds.
    - We can quickly obtain a comparsion plot of the performance, averaged across seeds.
- More qualitative analysis with media support [Demo 3]

    - In case we are working on computer vision, we might be interested in qualitative results (image generation or reconstruction)
    - WANDB also supports various type of media. Let's do an example with a reconstruction task.
    - Other features allow comparison of hyperparameters between runs
    - Or determining which hyperparameter value is more important.
- Team members can run different experiments independently, which can then be compared centraly. [Demo 4]
- WANDB's **Report** feature can be leveraged to created intutive and more interactive analysis reports for our experiments.
    - Those can then be shared as links along with the Github repository or the research paper for readers and reviewers to have a deeper insight on the published results.
    - Alternative, it could be used for website showcase when publishing a paper, such as this example by a collaborator

Overall, this type of tool can improve the productivity at an individual level, which also scales to teams of researchers, with a relatively low engineering cost.

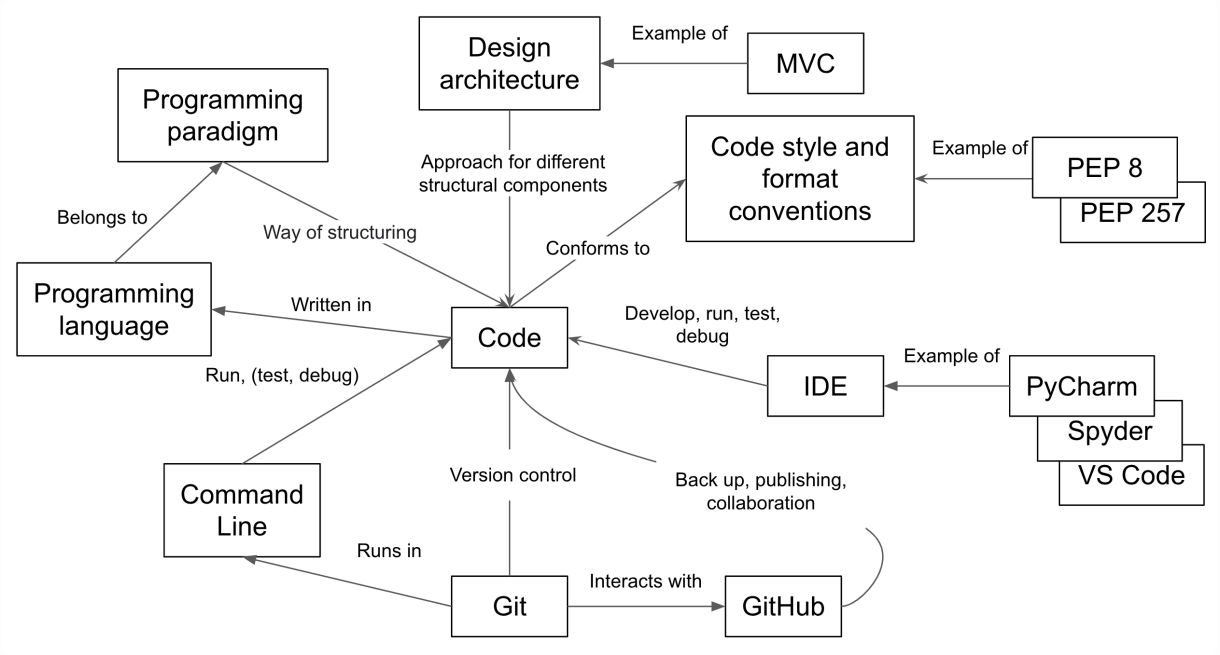
6. Wrap-up

Finally, let's recapitulate the content of this tutorial into a few takeaways points:

- Basic principle and tools for collaborative software engineering, be it research or industry
    - Version control: `git`, separation of active development into branches
    - Development environments: virtual envs. with Python and dependency management for reproducibility
    - IDE such as VSCode provide quality of life improvements, but are not mandatory.
- Testing: systematically making sure the code stays consistent with the established SW design principles
    - Write test that covers functionality as they are added
    - Automotate testing at key points of the workflow to maintain a consistent code base across collaborators.
- Design paradigms:
    - Heavily dependent on the research field and use case.
    - Knowledge of different programming paradigms and related concepts to better structure the code.

    - Object-Oriented Programming: adequate for data structuring and manipulation; provides modularity in the code, eases re-usability and extension.
    - Functional programming: mathematically inspired principles to write data processing functions for, yields unambiguous code, easy to test, compose and parallelize.
- PR based collaboration
    - Pull Request (PR) based collaborative workflow.
    - Code review proces and merging changes with PRs.
    - Incorporating testing into the PR workflow.
- Publishing code and results
    - Documentation, guidance for contributors, citation
    - Having easy to use API to increase traction around the published code and research.
    - Publish packages in a form that makes it as easy as `pip install` then `python -m my-program`
    - Distribute environment requirements, docker images etc... to ease reproduction efforts and re-use (helps with citations)
- Tools for collaborative research in machine learning
    - Solutions like Weight and Biases to manage experiment from small to large scale.
    - Sharing experiment runs (logged metrics, etc...) and reports with collaborators.
    - Publishing intutive and interactive reports along with the codebase and research papers.

A graphical summary of the concept covered in this tutorial, from the original course material.



7. Further resources

- Research Software Engineering with Python -
  Building software that makes research possible, see also a description of the book here.

8. Original course

The present tutorial is a modified version of Nadine Spychala's Collaborative Research Software Tutorial Course.

Modifications Overview

- Breaks the original course by Nadine into two parts, while fusing it with the answer part using "Spoiler" tags.
- Added / modified Section 5 for collaborative machine learning research tools and suggestions.
- Minor modifications and simplification across across the document.

9. License

Instructional material

- to share-copy and redistribute the material in any medium or format,
- to adapt-remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Work is derived from work that is Copyright © Nadine Spychala, 2023

Software

Except where otherwise noted, the example programs and other software provided by Nadine Spychala are made available under the OSI-approved MIT license.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal with the Software without restriction (i.e., without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software), subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

10. Acknowledgements

- Huge thanks to Nadine Spychala for the original version of this condensed tutorial on Collaborative Research Software Engineering, as well as the opportunity to co-instruct at the ALIFE 2023 Conference Ghost in the Machine.
- Thanks to The Carptentries Incubator for publishing their **Intermediate Research Software Development** course and allow for flexible re-use.
- Thanks to Hackmd.io for the easy to use markdown based editor and publication tools this tutorial rely upon.