Scientific programming patterns

Lisa Schwetlick and Pietro Berkes

What is wrong with y’all?!

- You studied the language
- You learned the libraries
- You coded for months

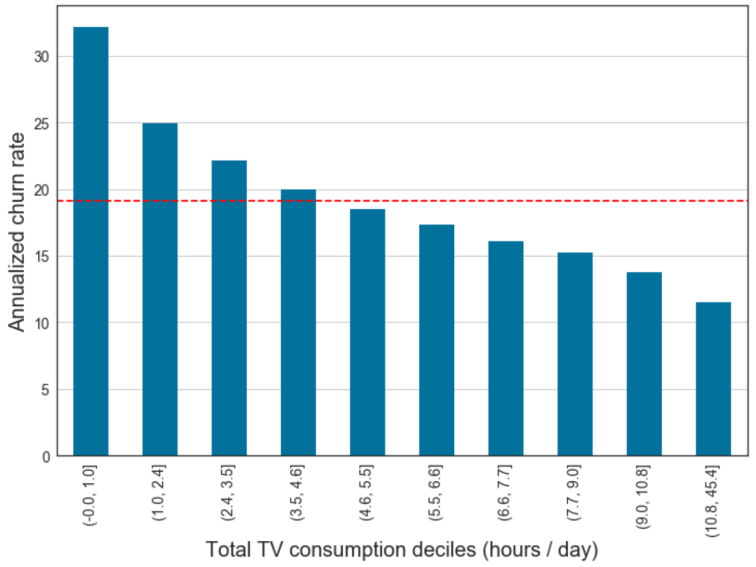
- And yet the code still feels like a 7-headed apocaliptic monster. Changes are painful, new features break old functionality, reproducing previous results becomes a git-checkout juggling exercise

The good news: you can smell it

7.1 Total TV

In [118]:
```
ttv_h_deciles = pd.qcut(ttv_h, 10)
ttv_deciles_churn = tv_merged.groupby(ttv_h_deciles).churned_all.mean() * 12 / 10 * 100
```

In [119]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = ttv_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Total TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['({:.1f}, {:.1f}]'.format(x.left, x.right) for x in ttv_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```

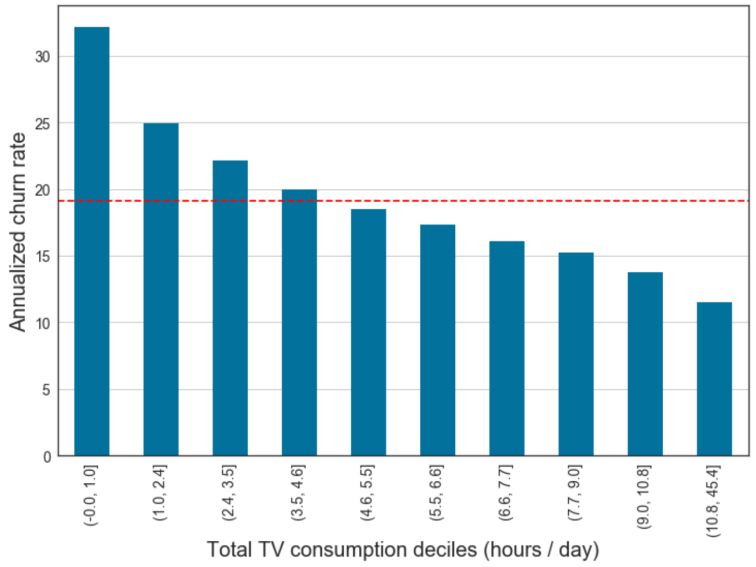


The good news: you can smell it

7.1 Total TV

In [118]:
```
ttv_h_deciles = pd.qcut(ttv_h, 10)
ttv_deciles_churn = tv_merged.groupby(ttv_h_deciles).churned_all.mean() * 12 / 10 * 100
```

In [119]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = ttv_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Total TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['({:.1f}, {:.1f})'.format(x.left, x.right) for x in ttv_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```



7.2 Playback

In [120]:
```
replay_deciles = pd.qcut(tv_merged.PLAYBACK / ttv_sec_to_hpd, 10)
replay_deciles_churn = tv_merged.groupby(replay_deciles).churned_all.mean() * 12 / 10 * 100
```

In [121]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = replay_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Replay TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['({:.1f}, {:.1f})'.format(x.left, x.right) for x in replay_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```

In [122]:
```
replay_h = pd.cut(tv_merged.PLAYBACK / ttv_sec_to_hpd, np.arange(0, 8), include_lowest=True)
replay_churn = tv_merged.groupby(replay_h).churned_all.mean() * 12 / 10 * 100
```

In [123]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = replay_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Replay TV consumption (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

7.3 Trends

In [137]:
```
tv_delta = pd.cut(tv_merged.TTV_201703_delta, np.arange(-5.5, 5.6, 1.0))
tv_delta_churn = tv_merged.groupby(tv_delta).churned_all.mean() * 12 / 10 * 100
```

In [138]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = tv_delta_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('TV consumption trend')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

In [1651]:
```
tv_delta = pd.cut(tv_merged.TTV_201703_delta, np.arange(-5.5, 5.6, 1.0))
tv_delta_churn_some = tv_merged[at_least_some_ttv_mask].groupby(tv_delta[at_least_some_ttv_mas
```

In [1652]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = tv_delta_churn_some.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('TV consumption trend')
    plt.ylabel('Annualized TV+Vodafone churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

What is wrong with smelly code?

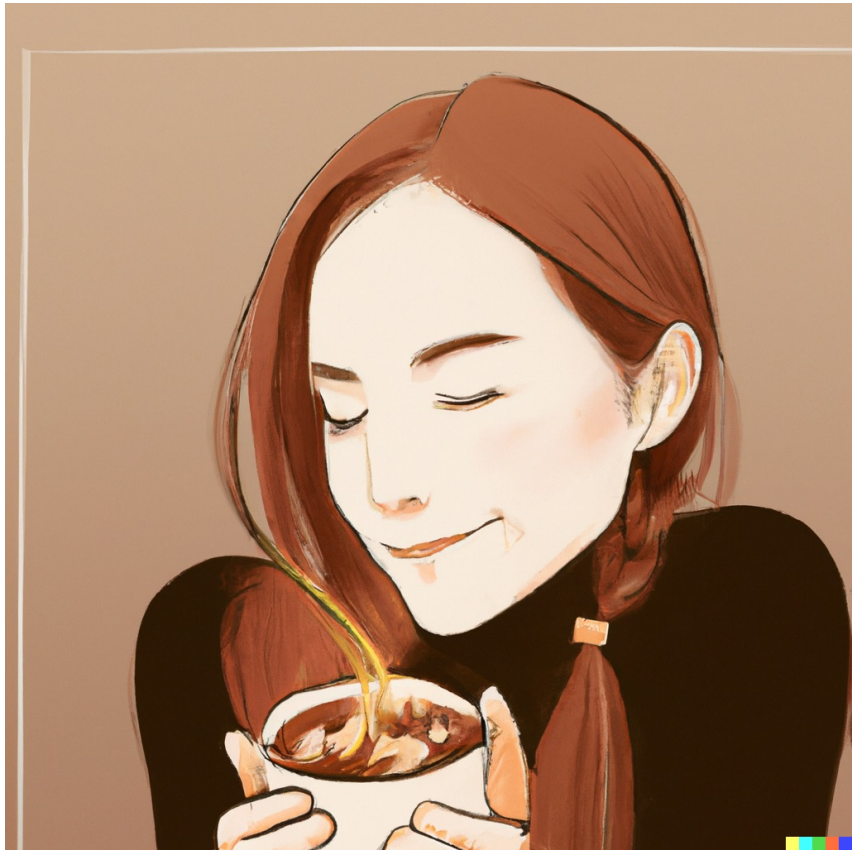
Smelly code might work right now, but in time it is going to have one or more of these issues:

- **Redundant**: an update in one place would need to be duplicated multiple times
- **Hard to read and test**: the code that performs the interesting computation is mixed with the code that does the repetitive boilerplate
- **Not flexible**: adding new functionality and modifying exiting features require extensive rewrites or hacks

Here is how to fix it, class dismissed

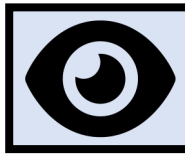
What are you missing? A few patterns that make your code odor as nice as a spring meadow

1. Group together things that belong together
2. Break out things that vary independently
3. Keep code open for extension



Chapter 1: Introduction to classes
Put together things that belong together

Classes live coding

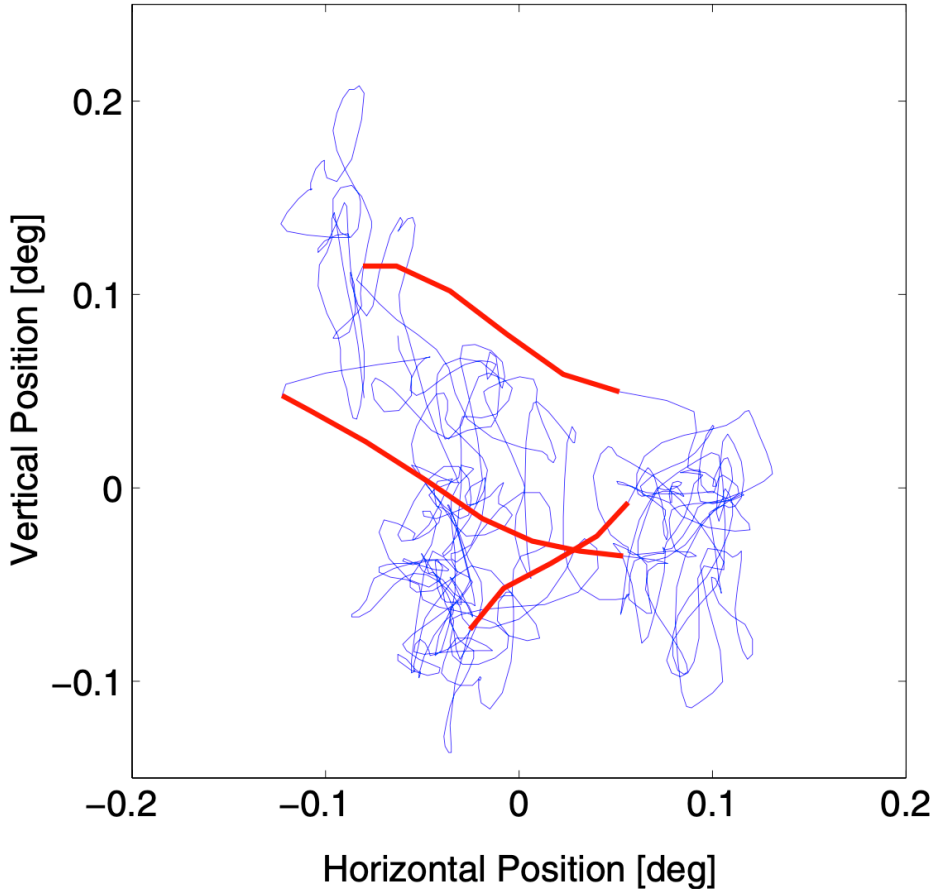


Live Coding

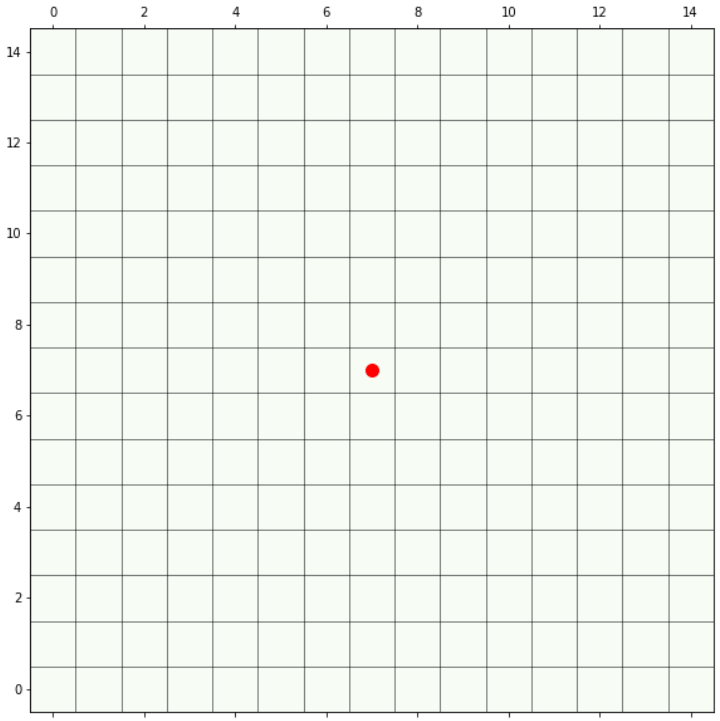
`notebooks/01a Classes.ipynb`

Excursion: let's walk!

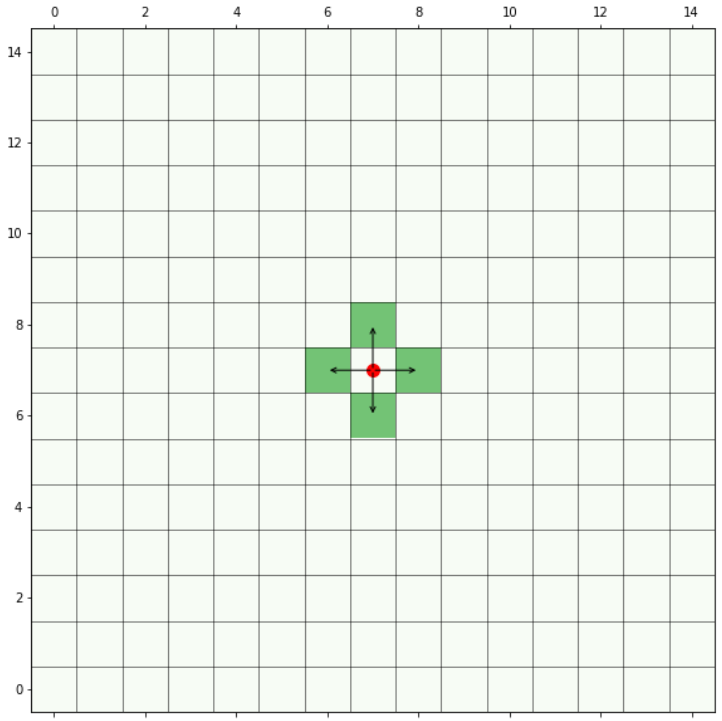
- Even when people think they are holding their eyes still, they still move around:
  - Drift, Microsaccades, jitter
- The movement has statistical properties such as self avoidance, directional persistence…
- The upcoming exercise is a simplified version of a model Lisa is working on



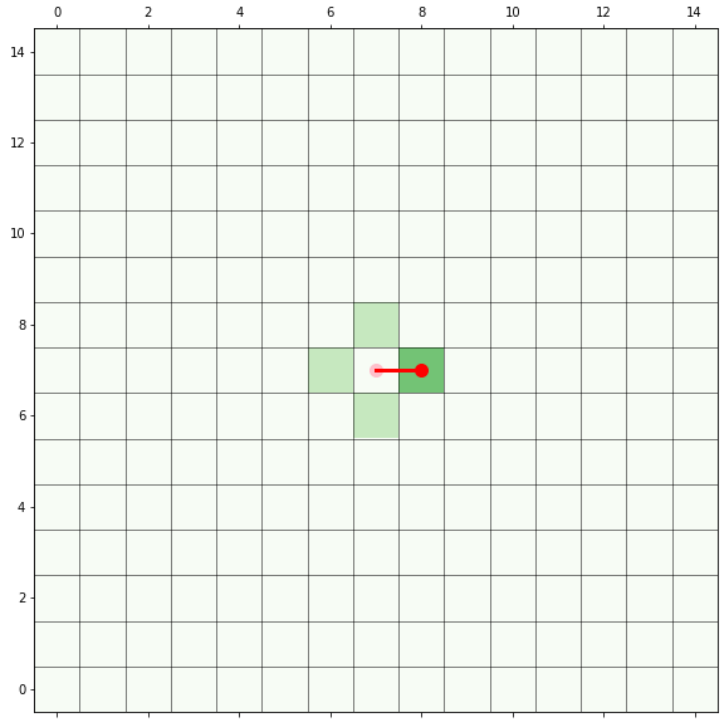
Excursion: let’s walk!



Walker starts somewhere

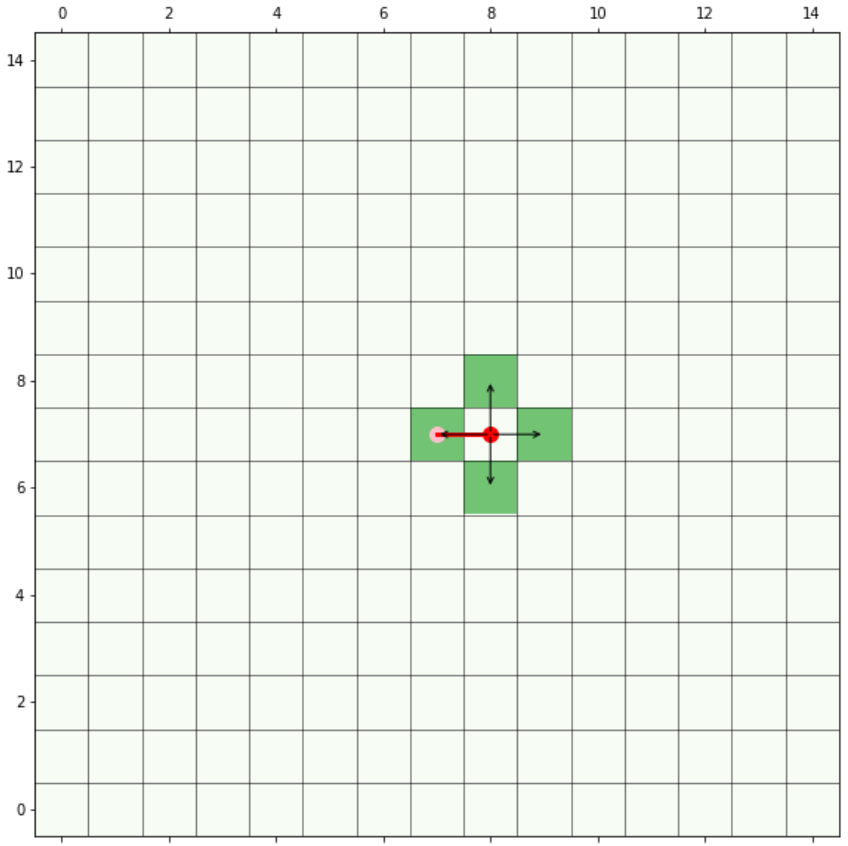


It could walk one step in any direction

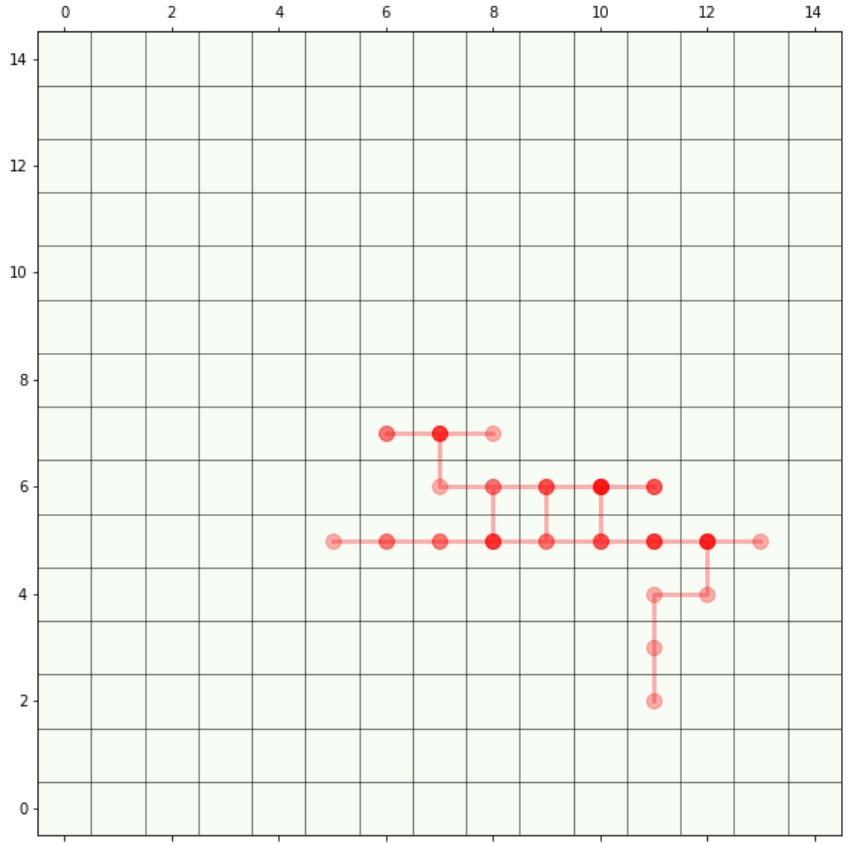


It randomly selects one step

Excursion: let’s walk!



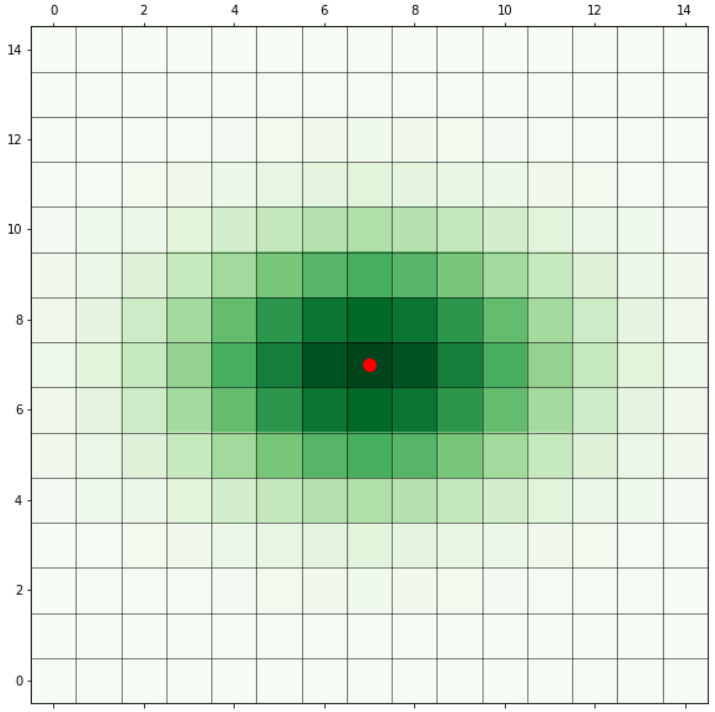
In the next step it has the same stepping options



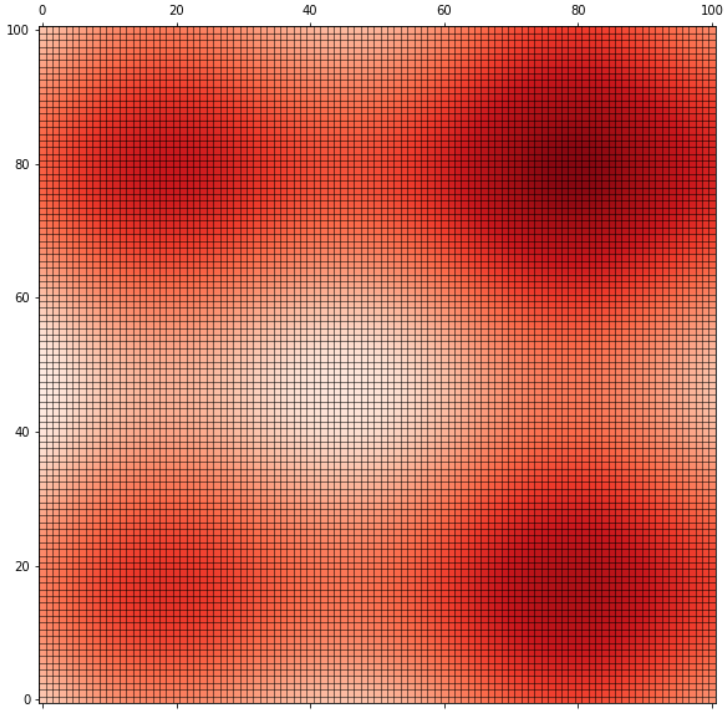
By iterating this procedure, we get a trajectory

Excursion: let's walk!

- To make the behavior a little bit more interesting the walker in the exercise…
  - choses its next step from a probability
  - Can also walk over a non uniform background, that influences how likely it is to go there

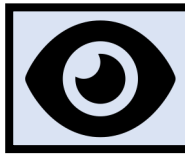


Stepping Probability



Background Activation

The walker Functions



Live Coding

`walker/01 Introduction/`

Hands-on

Turn the walker code into a class

Exercise

`walker/Step 1 classes/`

- All the exercises in this class are about **reformatting code!** You are not expected to go into detail about how the code works- copy&pasting is fine!
- There are many ways to solve the exercises. **It is more important to think about the implementation choices with your partner than to finish**. We will give you working code to start from at each step, so don’t worry about finishing!
- Submit a PR for Issue #1 on GitHub.

Summary: Classes organize your code

```
def first_function(x, y, z):
    # Something

def second_function(x, y, z):
    # Something else

def third_function(x, y, z):
    # Something more
```

… becomes …

```
class Xyz:
    def __init__(self, x, y, z):
        ...
    def first_function(self):
        # Something
    def second_function(self):
        # Something else
    def third_function(self):
        # Something more
```

- Classes help us group data and functionalities that belong together
- When used judiciously, the code becomes more usable as we get rid of a lot of manual book-keeping; details are hidden away
- Understanding what belongs to the class and what does not is important to keep the code flexible!

Chapter 2: There are many ways to build an instance
Break out things that vary independently

There are many ways to build an instance

- In tests, typically one wants to specify all the parameters *exactly*
- Sometimes there are multiple ways of setting the parameters, e.g. setting the initial weights in a neural network
- The instance could be created from some saved parameters (serialization)

The smells of the Walker constructor

```
def __init__(self, sigma_i, sigma_j, size, map_type='flat'):
    self.sigma_i = sigma_i
    self.sigma_j = sigma_j
    self.size = size

    if map_type == 'flat':
        context_map = np.ones((size, size))
    elif map_type == 'hills':
        grid_ii, grid_jj = np.mgrid[0:size, 0:size]
        i_waves = np.sin(grid_ii / 130) + np.sin(grid_ii / 10)
        i_waves /= i_waves.max()
        j_waves = np.sin(grid_jj / 100) + np.sin(grid_jj / 50) + \
            np.sin(grid_jj / 10)
        j_waves /= j_waves.max()
        context_map = j_waves + i_waves
    elif map_type == 'labyrinth':
        context_map = np.ones((size, size))
        context_map[50:100, 50:60] = 0
        context_map[20:89, 80:90] = 0
        context_map[90:120, 0:10] = 0
        context_map[120:size, 30:40] = 0
        context_map[180:190, 50:60] = 0

        context_map[50:60, 50:200] = 0
        context_map[179:189, 80:130] = 0
        context_map[110:120, 0:190] = 0
        context_map[120:size, 30:40] = 0
        context_map[180:190, 50:60] = 0
    context_map /= context_map.sum()
    self.context_map = context_map

    # Pre-compute a 2D grid of coordinates for efficiency
    self._grid_ii, self._grid_jj = np.mgrid[0:size, 0:size]
```

The smells of the Walker constructor

```
def __init__(self, sigma_i, sigma_j, size, map_type='flat'):
    self.sigma_i = sigma_i
    self.sigma_j = sigma_j
    self.size = size

    if map_type == 'flat':
        context_map = np.ones((size, size))
    elif map_type == 'hills':
        grid_ii, grid_jj = np.mgrid[0:size, 0:size]
        i_waves = np.sin(grid_ii / 130) + np.sin(grid_ii / 10)
        i_waves /= i_waves.max()
        j_waves = np.sin(grid_jj / 100) + np.sin(grid_jj / 50) + \
            np.sin(grid_jj / 10)
        j_waves /= j_waves.max()
        context_map = j_waves + i_waves
    elif map_type == 'labyrinth':
        context_map = np.ones((size, size))
        context_map[50:100, 50:60] = 0
        context_map[20:89, 80:90] = 0
        context_map[90:120, 0:10] = 0
        context_map[120:size, 30:40] = 0
        context_map[180:190, 50:60] = 0

        context_map[50:60, 50:200] = 0
        context_map[179:189, 80:130] = 0
        context_map[110:120, 0:190] = 0
        context_map[120:size, 30:40] = 0
        context_map[180:190, 50:60] = 0
    context_map /= context_map.sum()
    self.context_map = context_map

    # Pre-compute a 2D grid of coordinates for efficiency
    self._grid_ii, self._grid_jj = np.mgrid[0:size, 0:size]
```

1. The constructor will become longer with more map types
2. We cannot contribute a new map type without modifying the code
3. It is difficult to test
4. It is not flexible, e.g. what happens if we want to create an instance from parameters saved on file?

How would you fix it?

The smells of the Walker constructor

```
def __init__(self, sigma_i, sigma_j, size, map_type='flat'):
    self.sigma_i = sigma_i
    self.sigma_j = sigma_j
    self.size = size

    if map_type == 'flat':
        context_map = np.ones((size, size))
    elif map_type == 'hills':
        grid_ii, grid_jj = np.mgrid[0:size, 0:size]
        i_waves = np.sin(grid_ii / 130) + np.sin(grid_ii / 10)
        i_waves /= i_waves.max()
        j_waves = np.sin(grid_jj / 100) + np.sin(grid_jj / 50) + \
            np.sin(grid_jj / 10)
        j_waves /= j_waves.max()
        context_map = j_waves + i_waves
    elif map_type == 'labyrinth':
        context_map = np.ones((size, size))
        context_map[50:100, 50:60] = 0
        context_map[20:89, 80:90] = 0
        context_map[90:120, 0:10] = 0
        context_map[120:size, 30:40] = 0
        context_map[180:190, 50:60] = 0

        context_map[50:60, 50:200] = 0
        context_map[179:189, 80:130] = 0
        context_map[110:120, 0:190] = 0
        context_map[120:size, 30:40] = 0
        context_map[180:190, 50:60] = 0
    context_map /= context_map.sum()
    self.context_map = context_map

    # Pre-compute a 2D grid of coordinates for efficiency
    self._grid_ii, self._grid_jj = np.mgrid[0:size, 0:size]
```

1. The constructor will become longer with more initialization types
2. We cannot contribute a new initialization type without modifying the code
3. It is difficult to test
4. It is not flexible, e.g. what happens if we want to create an instance from parameters saved on file?

These are smells of the fact that the initialization of context_map varies independently of the class Walker

We need two ingredients to fix this

Factory methods build instances in different ways

```python
class MyClass:

    def __init__(self, a, b):
        """The basic constructor takes 'raw' values."""
        self.a = a
        self.b = b

    @classmethod
    def from_random_values(cls, random_state=np.random):
        """Create a MyClass instance with random parameters."""
        a = random_state.rand()
        b = random_state.randn()
        return cls(a, b)

    @classmethod
    def from_json(cls, json_fname):
        """Create a MyClass instance with parameters read form a json file."""
        with open(json_fname, 'r') as f:
            dict_ = json.load(f)
        a = dict_['a']
        b = dict_['b']
        return cls(a, b)

my_class = MyClass.from_random_values()
```

The main constructor should not do anything fancy

The factory methods create the parameters in some fancy way, then call the basic constructor

A @classmethod is a method that takes a reference to the **class**, not the **instance**, as its input

Their main use is as factory methods, to create instances

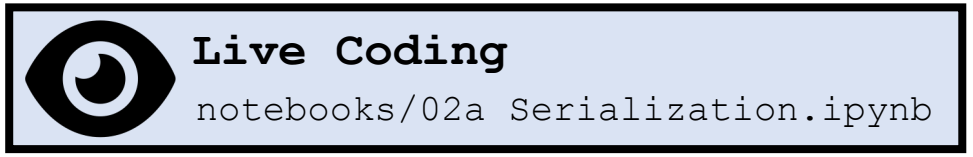
Live Coding factory methods

Persistence and serialization: Saving grouped data

- Persistence: saving the state of an instance to disk
- Serialization: transforming an object into another representation; in this context, something that can be saved to disk

- pickle
  - Easy to use, it shoud just work
  - It breaks easily when the code changes (a simple renaming of the class, for instance)
- JSON
  - the file equivalent of a dictionary
  - you'll need to handle the transformation to and from json yourself
  - custom types like Numpy arrays are not supported by default, need to turn them into lists or strings, or else save them separately and add the path in the JSON file

Serialization live coding



Hands On:
Serialize the Walker

Exercise

`walker/Step 3 serialize/`

- Submit a PR for Issue #2 on GitHub.
- Have a look at the "**Step 2 factory method**" notebook
- Write the `walker.to_json()` method and `walker.from_json()` factory method

Recap: Class structure

```
class MyClass:
    def __init__(self, param1, param2):
        self.param1 = param1
        if param2 is None:
            param2 = 12.3
        self.param2 = param2

    def my_method(self, foo, bar):
        result = self.param2 * foo + self.param2 * bar
        return result

    def to_json(self):
        params = {
            'param1': self.param1,
            'param2': self.param2,
        }
        return json.dumps(params)

    @classmethod
    def from_json(cls, json_str):
        json_dict = json.loads(json_str)
        instance = cls(json_dict['params1'], json_dict['params2'])
        return instance


instance = MyClass(5, 18)
instance = MyClass.from_json(json_str)
```

The constructor is used to first populate an instance, called by convention “self”

Classes can define “methods”, i.e. functions that have access to the data stored in an instance

A ”class method” can be used to build an instance in some alternative way, e.g. using data from a file

Here is how you create instances from the constructor or a class method

Factory methods take us only this far...

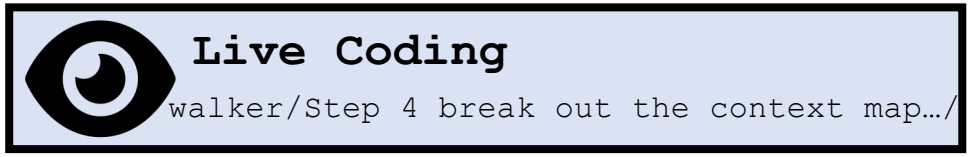
```python
def __init__(self, sigma_i, sigma_j, size, context_map):
    self.sigma_i = sigma_i
    self.sigma_j = sigma_j
    self.size = size
    self.context_map = context_map
    # Pre-compute a 2D grid of coordinates for efficiency
    self._grid_ii, self._grid_jj = np.mgrid[0:size, 0:size]

@classmethod
def from_context_map_type(cls, sigma_i, sigma_j, size, map_type):
    """ Create an instance of Walker with a context map defined by type."""
    if map_type == 'flat':
        context_map = np.ones((size, size))
    elif map_type == 'hills':
        grid_ii, grid_jj = np.mgrid[0:size, 0:size]
        i_waves = np.sin(grid_ii / 130) + np.sin(grid_ii / 10)
        i_waves /= i_waves.max()
        j_waves = np.sin(grid_jj / 100) + np.sin(grid_jj / 50) +\
            np.sin(grid_jj / 10)
        j_waves /= j_waves.max()
        context_map = j_waves + i_waves
    elif map_type == 'labyrinth':
        context_map = np.ones((size, size))
        context_map[50:100, 50:60] = 0
        context_map[20:89, 80:90] = 0
        context_map[90:120, 0:10] = 0
        context_map[120:size, 30:40] = 0
        context_map[180:190, 50:60] = 0

        context_map[50:60, 50:200] = 0
        context_map[179:189, 80:130] = 0
        context_map[110:120, 0:190] = 0
        context_map[120:size, 30:40] = 0
        context_map[180:190, 50:60] = 0

        context_map /= context_map.sum()
    return cls(sigma_i, sigma_j, size, context_map)
```

```python
walker = Walker.from_context_map_type(sigma_i=3, sigma_j=4, size=200, map_type='hills')
```

We broke apart the monolitic constructor into a basic, parameter setting part, and the specialized context-map initialization.

1. The ~~constructor~~ factory method will become longer with more initialization types
2. We cannot contribute a new initialization type without modifying the code
3. It is still somewhoat difficult to test
4. ~~It is not flexible, e.g. what happens if we want to create an instance from parameters saved on file?~~

Breaking out live coding



Break out the part that varies independently!

```python
class Walker:

    def __init__(self, sigma_i, sigma_j, size, context_map):
        self.sigma_i = sigma_i
        self.sigma_j = sigma_j
        self.size = size
        self.context_map = context_map
        # Pre-compute a 2D grid of coordinates for efficiency
        self._grid_ii, self._grid_jj = np.mgrid[0:size, 0:size]

    @classmethod
    def from_context_map_builder(cls, sigma_i, sigma_j, size, context_map_builder):
        """Initialize the context map from an external builder.

        `builder` is a callable that takes a `size` as input parameter
        and outputs a `size` x `size` numpy array of positive values.
        """
        context_map = context_map_builder(size)
        context_map /= context_map.sum()
        return cls(sigma_i, sigma_j, size, context_map)
```

```python
walker = Walker.from_context_map_builder(
    sigma_i=3,
    sigma_j=4,
    size=200,
    context_map_builder=hills_context_map_builder,
)
```

```python
def flat_context_map_builder(size):
    """ A context map where all positions are equally likely. """
    return np.ones((size, size))
```

```python
def hills_context_map_builder(size):
    """ A context map with bumps and valleys. """
    grid_ii, grid_jj = np.mgrid[0:size, 0:size]
    i_waves = np.sin(grid_ii / 130) + np.sin(grid_ii / 10)
    i_waves /= i_waves.max()
    j_waves = np.sin(grid_jj / 100) + np.sin(grid_jj / 50) + \
        np.sin(grid_jj / 10)
    j_waves /= j_waves.max()
    context_map = j_waves + i_waves
    return context_map
```

```python
def labyrinth_context_map_builder(size):
    """ A context map that looks like a labyrinth. """
    context_map = np.ones((size, size))
    context_map[50:100, 50:60] = 0
    context_map[20:89, 80:90] = 0
    context_map[90:120, 0:10] = 0
    context_map[120:size, 30:40] = 0
    context_map[180:190, 50:60] = 0

    context_map[50:60, 50:200] = 0
    context_map[179:189, 80:130] = 0
    context_map[110:120, 0:190] = 0
    context_map[120:size, 30:40] = 0
    context_map[180:190, 50:60] = 0

    return context_map
```

Break out the part that varies!

```python
@classmethod
def from_context_map_builder(cls, sigma_i, sigma_j, size, context_map_builder):
    """Initialize the context map from an external builder.

    `builder` is a callable that takes a `size` as input parameter
    and outputs a `size` x `size` numpy array of positive values.
    """
    context_map = context_map_builder(size)
    context_map /= context_map.sum()
    return cls(sigma_i, sigma_j, size, context_map)
```

```python
def flat_context_map_builder(size):
```

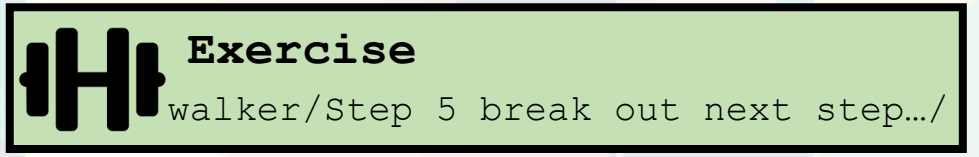
```python
def hills_context_map_builder(size):
```

```python
def labyrinth_context_map_builder(size):
```

- But wait, now we are passing a function, as if it were a variable!
- Functions, Classes, Class Instances, Variables etc. are all treated the same way, until used in some way (**duck typing)**

Hands On:

breaking out the next step probability



- Submit a PR for Issue #3 on GitHub.
- Have a look at the "**Step 5 break out next step probability**" notebook
- In it you will find some different ways of computing the next step probability. Break out the next_step_probability function to initialize the walker with different variations.

Separate what varies independently Part 2
The same principle applies everywhere, including at the level of project

The holy trinity of scientific computing

1. Provenance
2. Reproducibility
3. Organization

The holy trinity of scientific computing

1. Provenance

It needs to be clear **where** data and plots come from, **when** and **how** they were generated

2. Reproducibility

All information necessary to **get the same result** needs to be saved

3. Organization

For your own sanity, you should have a **consistent system** for all this **data** and **artefacts**

1. Provenance

- Data Provenance, or lineage, documents **where** data comes from
- Recording **when** and **by which** code the dataset has been changed
- But HOW?
  - External software? Usually not scpecific for scientific usecase.
  - Folder structure/Filenames?
  - Code generated meta-information files? (see next slide)
- In case of plots: data generated the plots
  - For work in progress plots: plt.annontate()
  - Save .ipynb as pdf (with all the paths/version information in the notebook)
  - Could use meta flags in images https://github.com/dfm/savefig

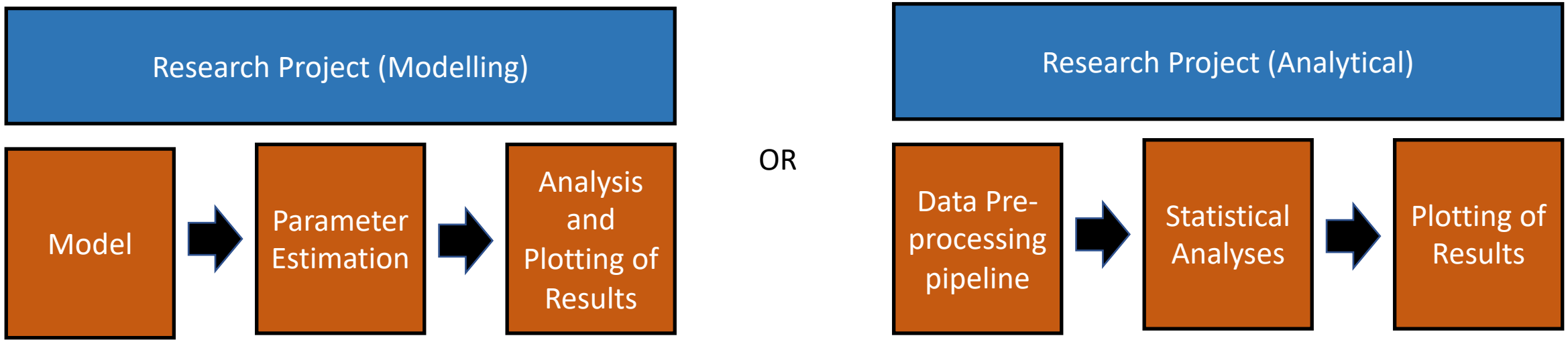
2. Reproducibility

- Save all information necessary to get the same result again
  - All input parameters to the code
  - Randomness- if used, save the seed
  - Which version of the code was used?

- Serialization of intermediate steps in the code (estimation and analysis)

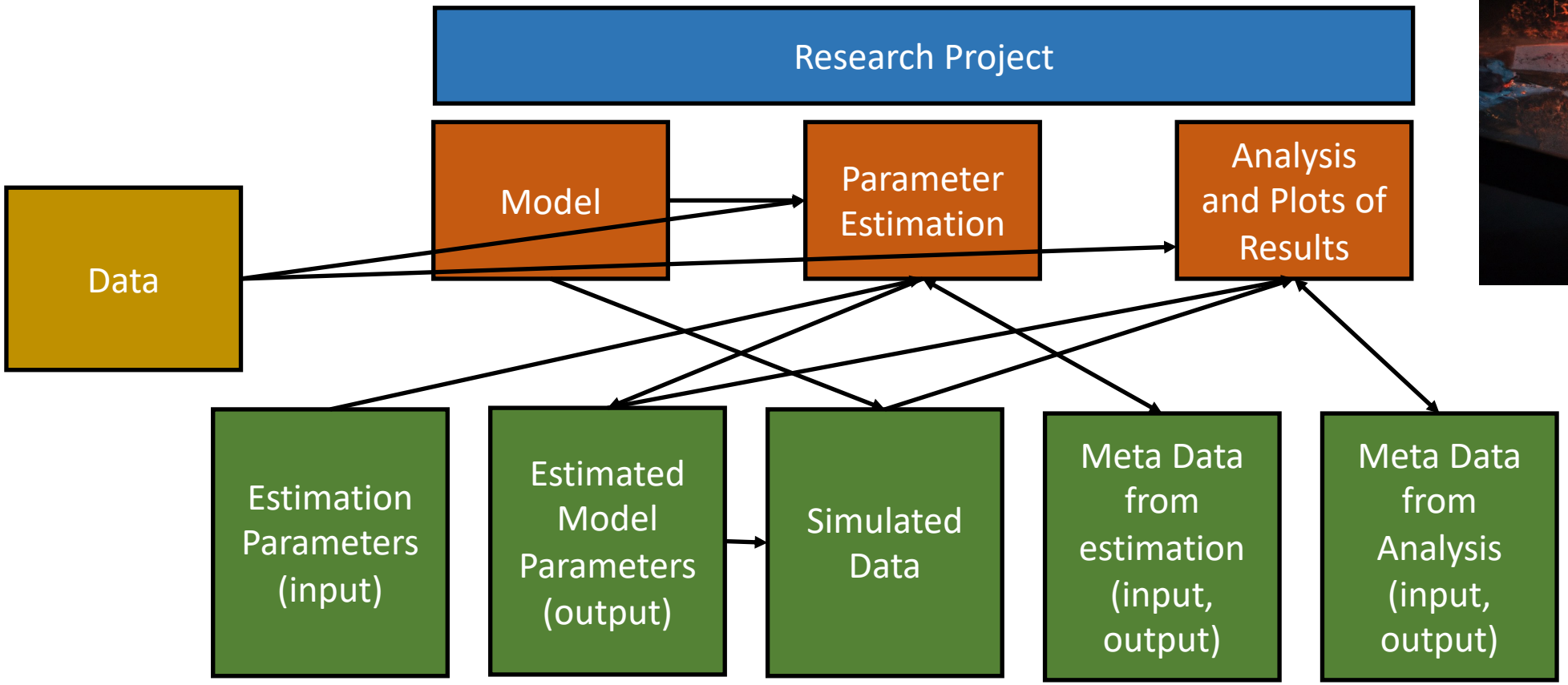
```python
import git
import time

# lookup git repository
repo = git.Repo(search_parent_directories=True)
sha = repo.head.object.hexsha

# get current time
estim_time = (time.strftime("%Y%m%d-%H%M%S"))

with open('readme.txt', 'w') as f:
    f.write(f'I estimated parameters at {estim_time}')
    f.write(f'The git repo was at commit {sha}')

^
```

3. Organization



- Your research project may look something like this (or maybe you have different steps or a subset of steps)
- In any case it is likely that you will go through the steps many times before you‘re ready to publish your work

3. Organization

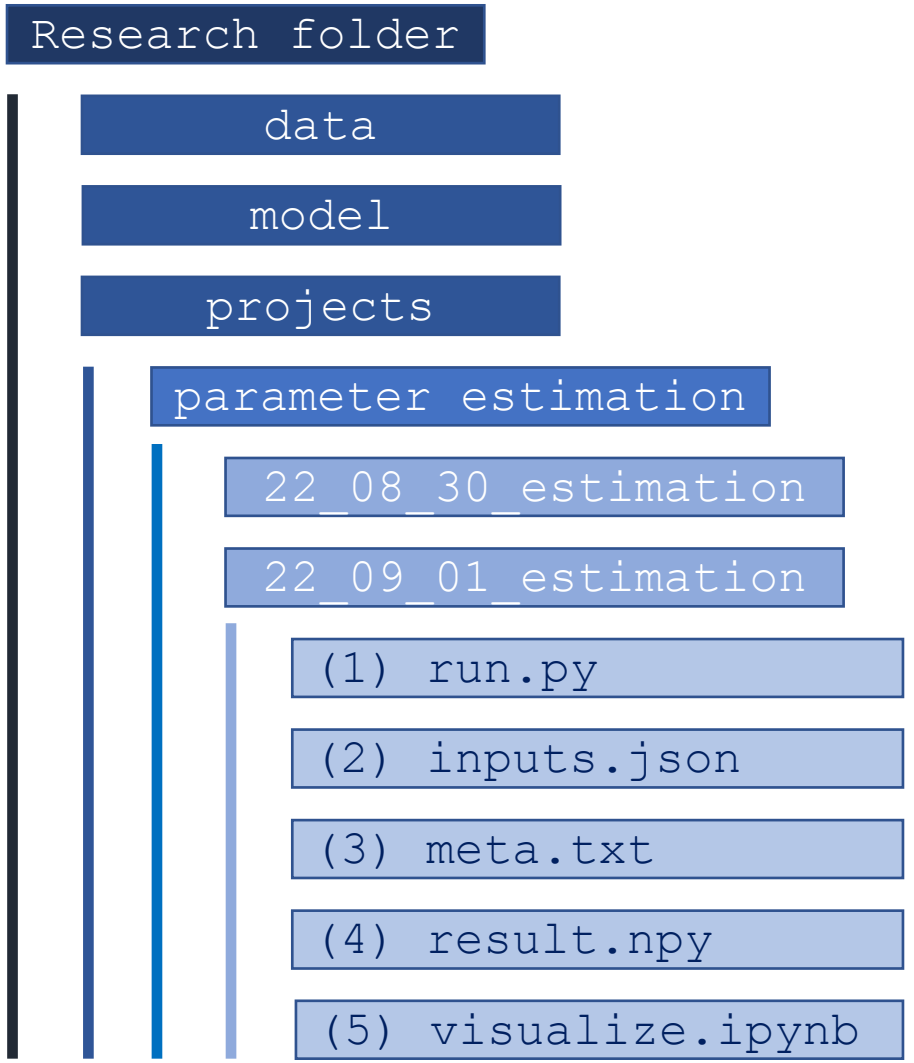


Each run has a bunch of associated data resulting folders and folders of data where no one knows which version of the code generated it or is using it!

3. Organization

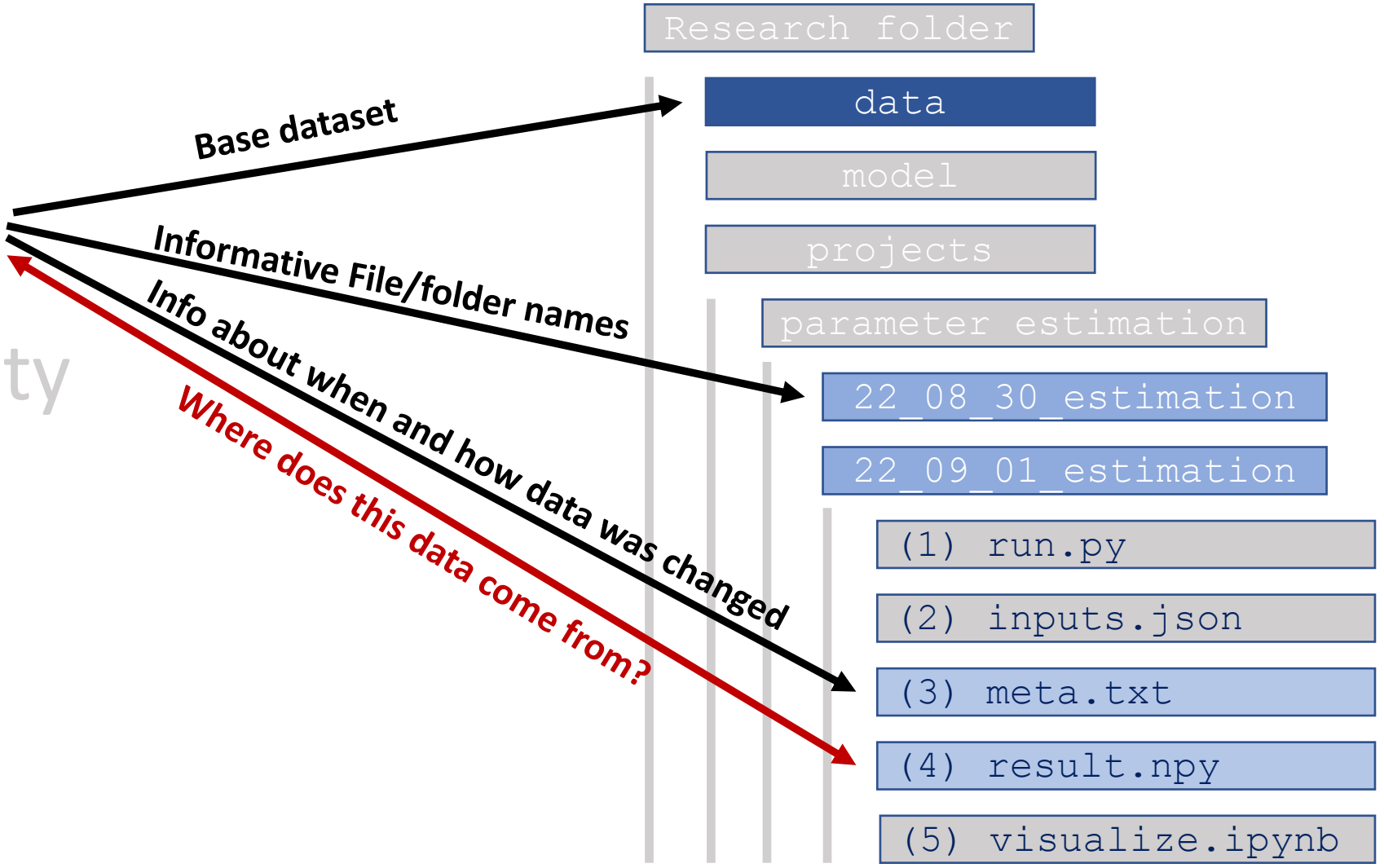
Suggestion:

- Data should always be separated from code
- The model or algorithms or things that are applied to your data should be packages (see packaging lecture)
- Think of “runs” as Experiments. Each experiment has its own folder. The folder contains:
  1. **Minimal code** that calls the model and saves the result
  2. All **inputs** necessary to produce the result saved separately
  3. **Meta information** about which version of your code was used (and maybe a note about what you were trying to achieve, if you want to be extra nice to future you)
  4. The **result**
  5. Maybe the visualization of the result



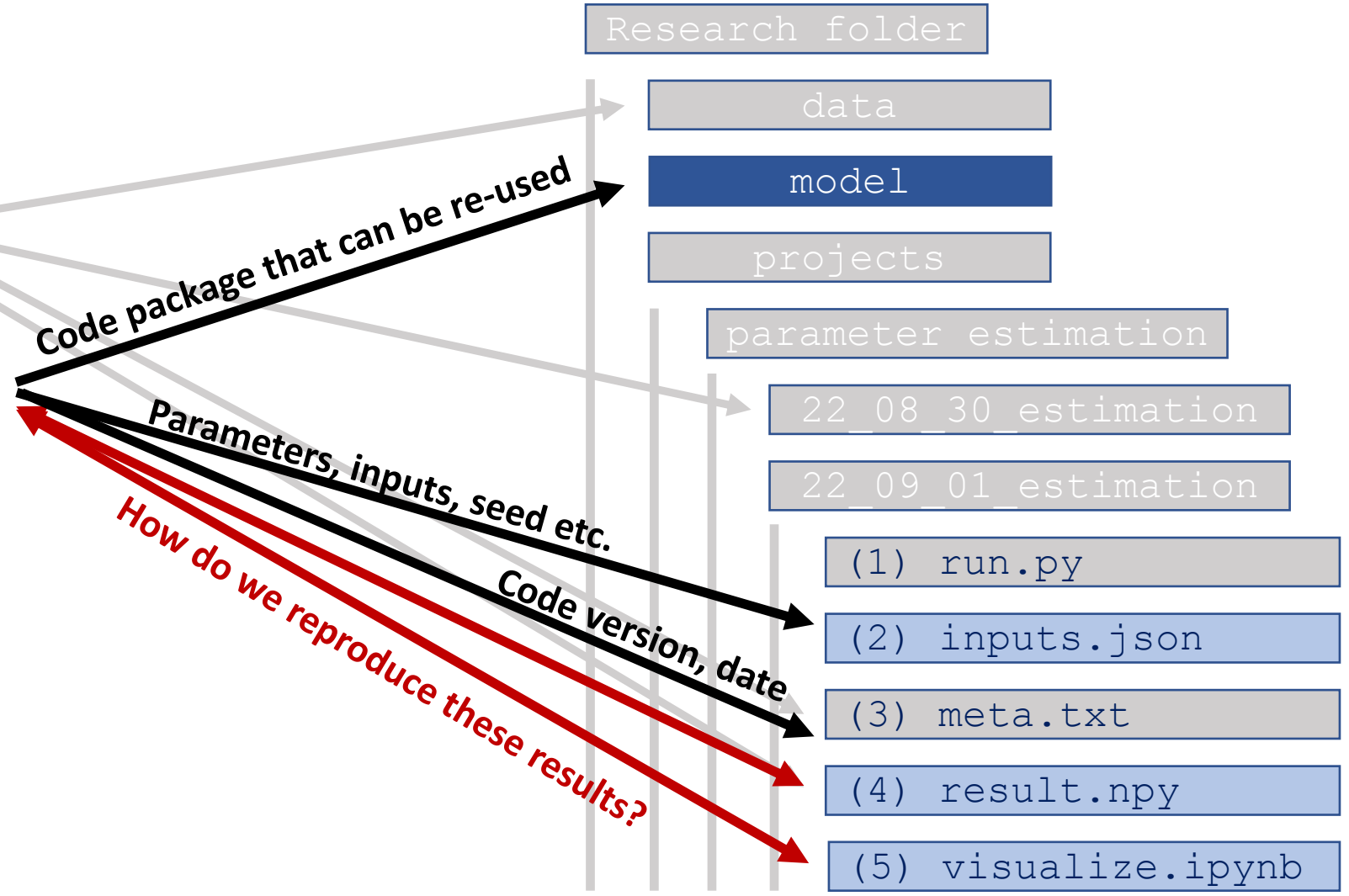
Organization

1. Provenance
2. Reproducibility
3. Organization



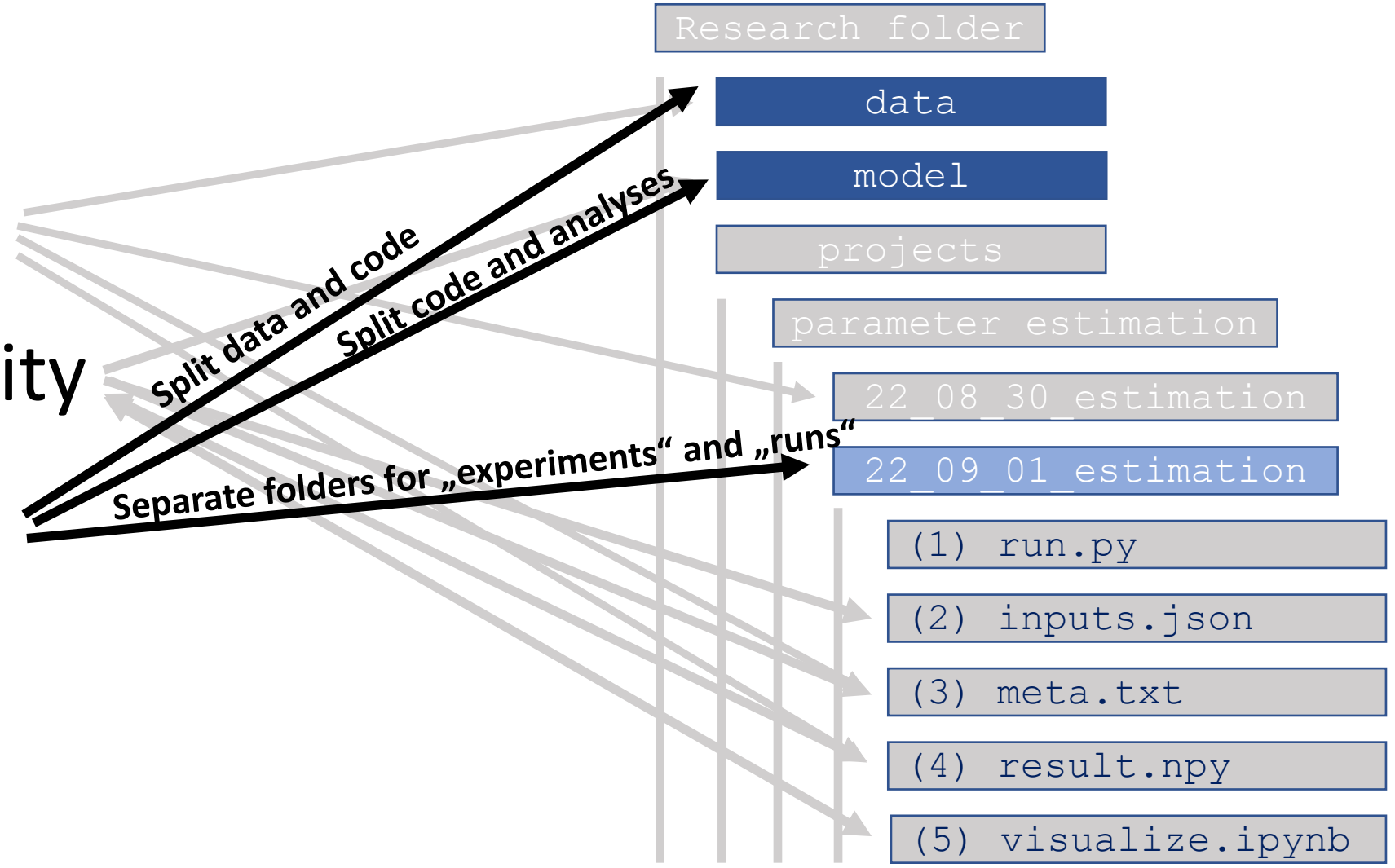
Organization

1. Provenance
2. Reproducibility
3. Organization



Organization

1. Provenance
2. Reproducibility
3. Organization



Hands On: Final Exercise

Exercise

`walker/Step 6 full workflow/`

Go through the following steps!

1. Submit PR for Issue #4 on GitHub
2. Look at the inputs.json file
3. Use your Walker class to simulate a Trajectory
4. Write some code that automatically generates a file with information about the run (time, git hash, run time, your favorite color…)
5. Save the trajectory and walker

Thank you!

Keep things open for e

Bonus material

Hooks patterns

Bonus material

- common cases:
  - in graph traversing algorithms (e.g., depth-first search) the graph traversing is generic, but the operation to be done with the data on the nodes is specific to the application. Graph libraries often implement the traverversing, and allow implementing the operation through hooks (hook when first visiting node, and when all children are visited on the way back)
  - in some UI frameworks, hooks can be added to react to certain UI events

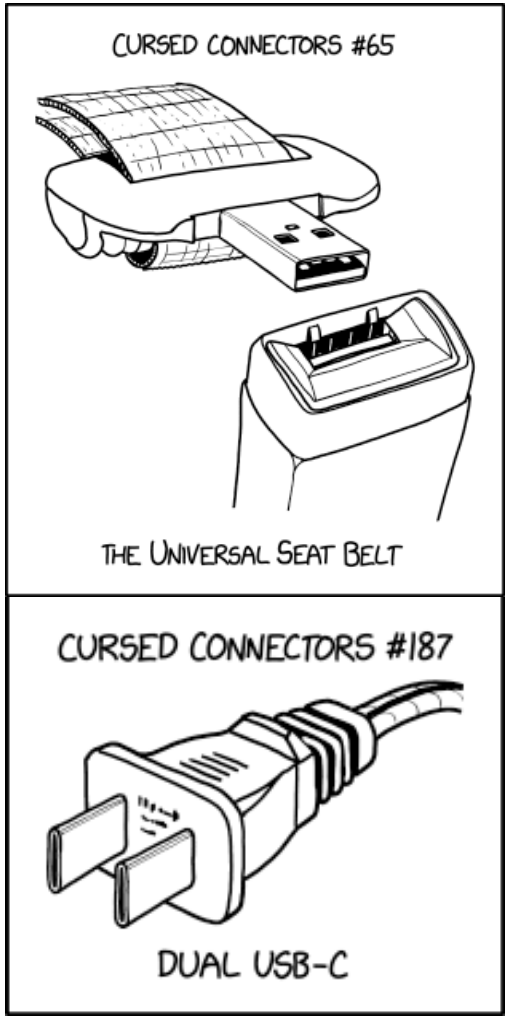
Architecture discussion?

- walker.from_data(data)
- walker.fit(data)
- walker_from_data(data), return fitted instance
- fit(walker, data), return parameters

- trajectory from walker
- walker.trajectory(n_steps) (hooks might be useful)
- trajectory(walker, n_steps) (hooks not so useful – just write another trajectory creator)

What is an API?

- How is the interface between your code and your manager scripts?
- Other things to consider when writing your code:
  - Who will be using it?
    - Maybe your code has a practical application
    - Even if it‘s most likely no one, imagine someone trying to replicate your research after you publish
  - Are there parts of the code you may want to use in your next project?
    - E.g. a fitting algorithm can be reused when you move on to the next model
    - E.g. a class for your data may be reusable for the next dataset



Where to go from here…

realpython.com