



Scientific programming patterns

Lisa Schwetlick and Pietro Berkes

What is wrong with you?!

- You studied the language
- You learned the libraries
- You coded for months
- And yet the code still feels like a 7-headed apocalyptic monster. Changes are painful, new features break old functionality, reproducing previous results becomes a git-checkout juggling exercise



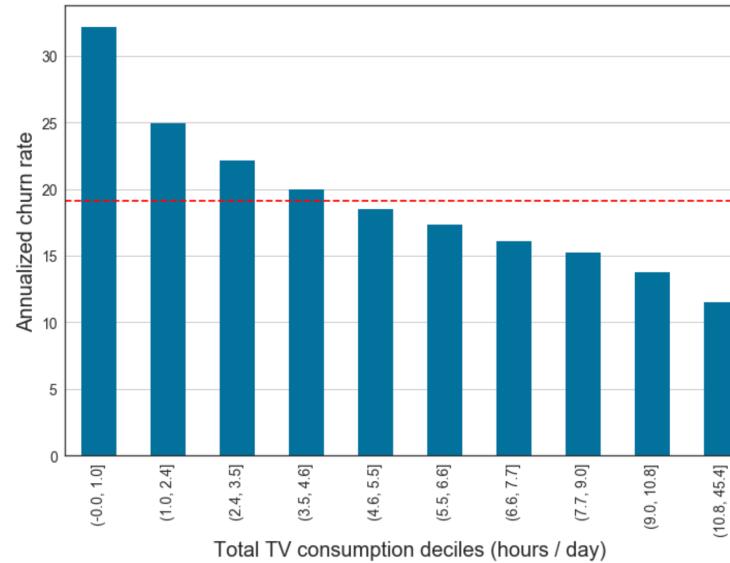
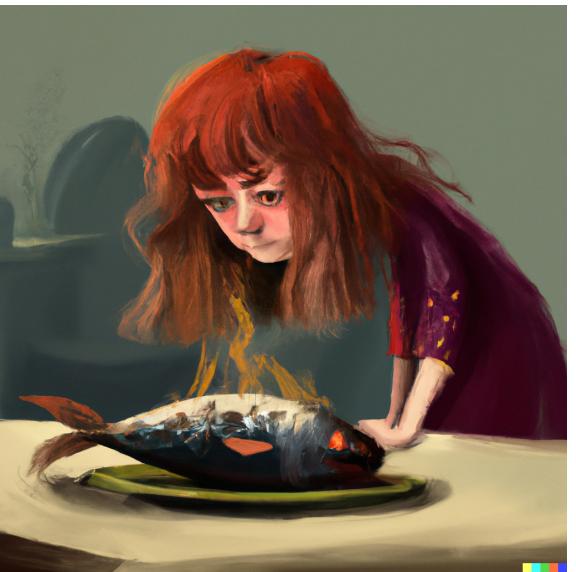
The good news: you can smell it

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)
```



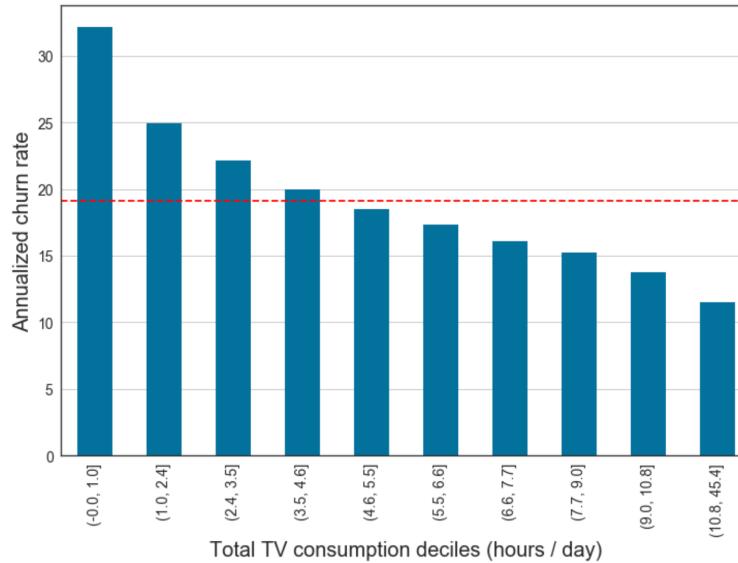
The good news: you can smell it

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_ma
```

```
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:

- **Hard to read and test:** it is difficult to see an overall structure; understanding the code in one place requires checking other code all over
- **Coupled:** an update in one place requires several other changes in other places
- **Not flexible:** adding new functionality and modifying existing features require extensive rewrites or hacks



Code should be as a construction block structure

Functions and classes group together things that are coupled together and form the basic blocks.

We can easily and quickly rearrange the blocks to extend a structure or build a new one!

Flexible code is just like a block construction:

- is easy to understand in terms of blocks
- tolerates changes
- is reusable



Chapter 1: Introduction to classes

Put together things that belong together



Classes live coding



Hands-on

Add a new method to the Particle class

- Make the function `update_position` into a method of the class `Particle`.
- Where do the position position of the particle belong? Modify the class constructor if necessary.
- 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!

Recap: Class structure

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

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

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

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

```
    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
```

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

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

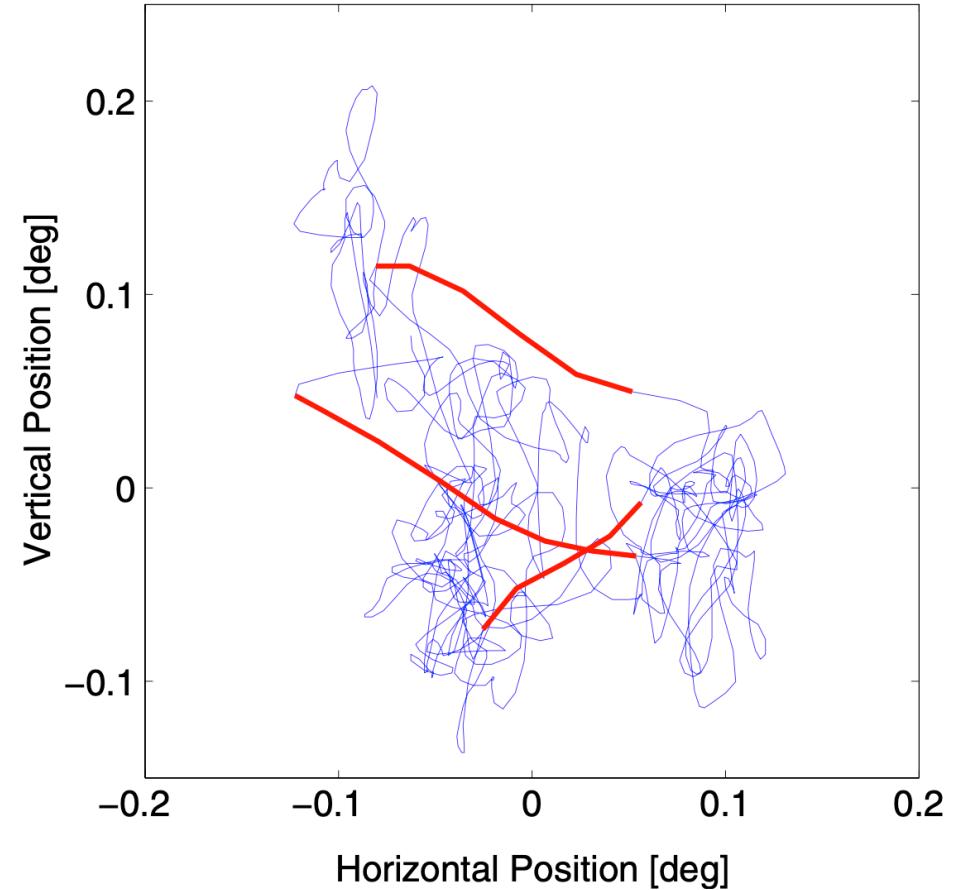
Here is how you create instances from the constructor or a class method

Chapter 2: Break out things that vary independently

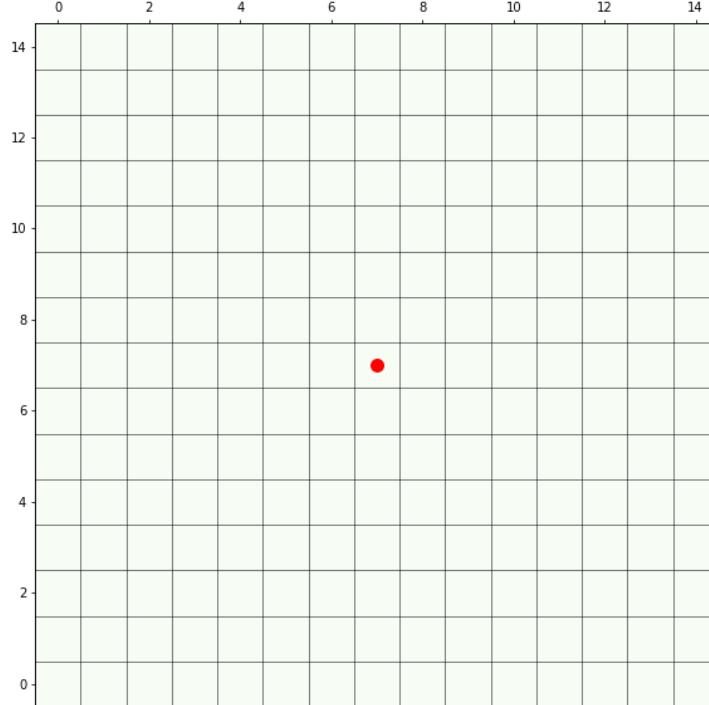
Excursion: let's walk!



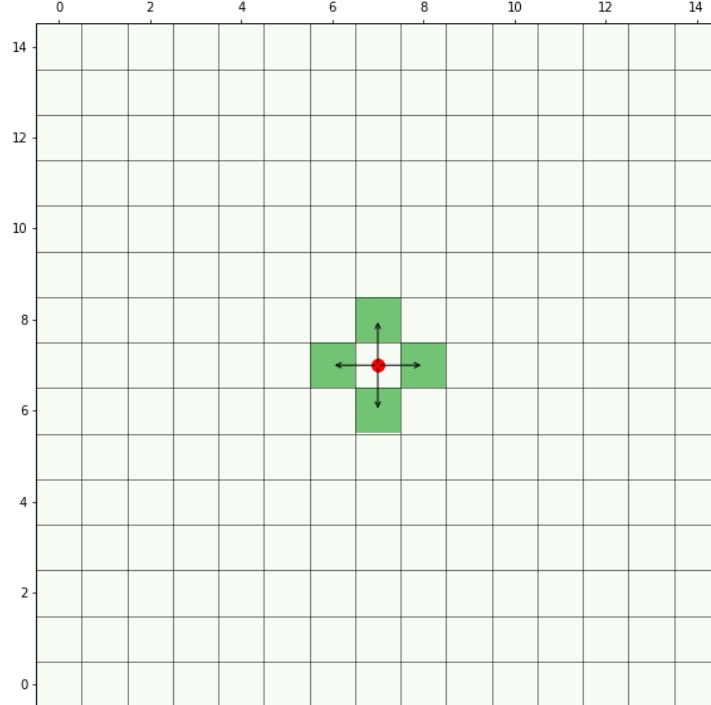
- Even when people think they are holding their eyes still, they still move around:
 - Drift, Microsaccades, jitter
- The movement has statistical properties such as self avoidance, directional persistence...
- The upcoming exercise is a simplified version of a model Lisa is working on



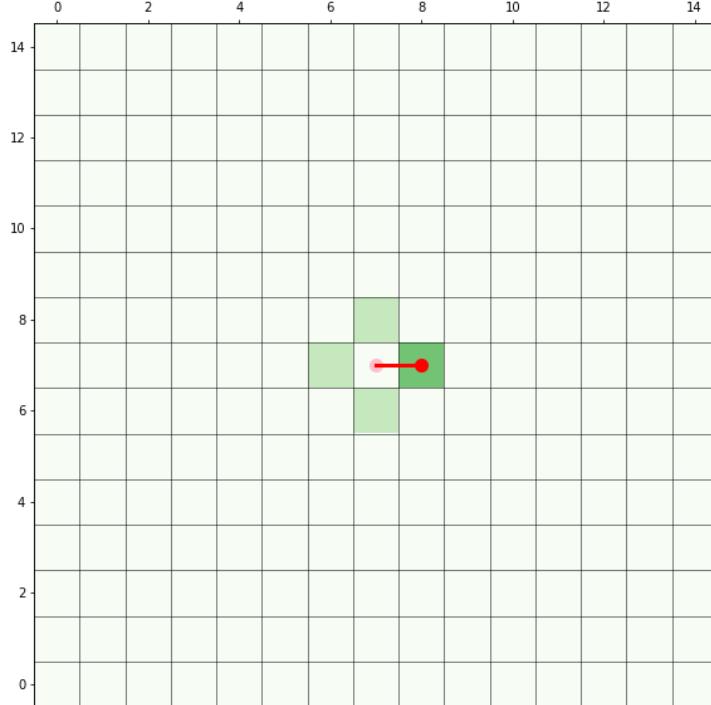
Excursion: let's walk!



Walker starts
somewhere

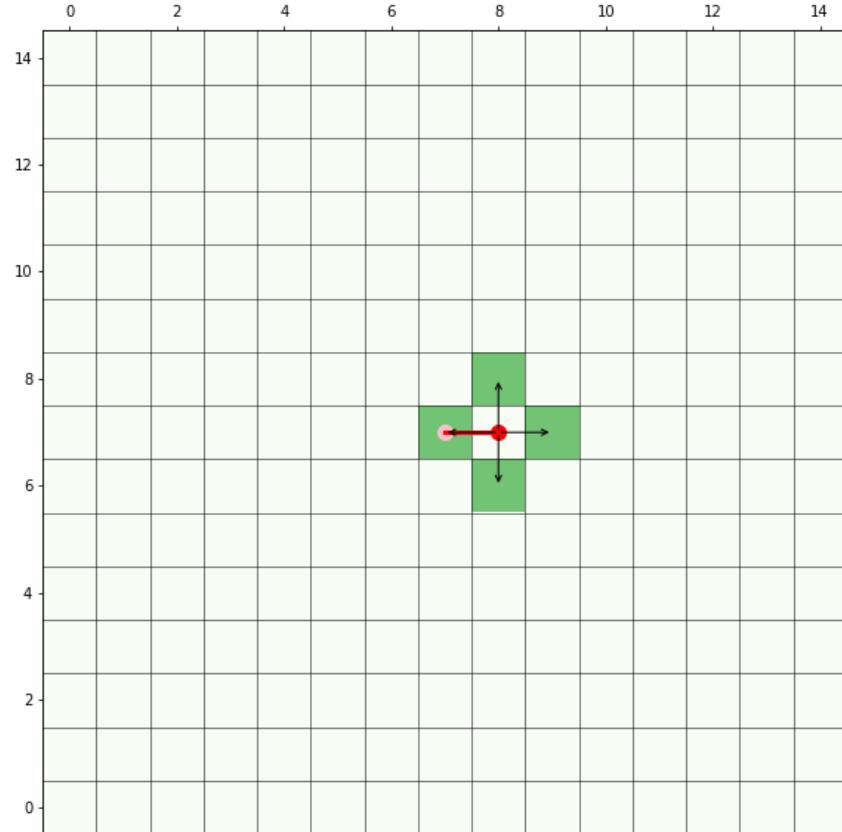


It could walk
one step in any
direction

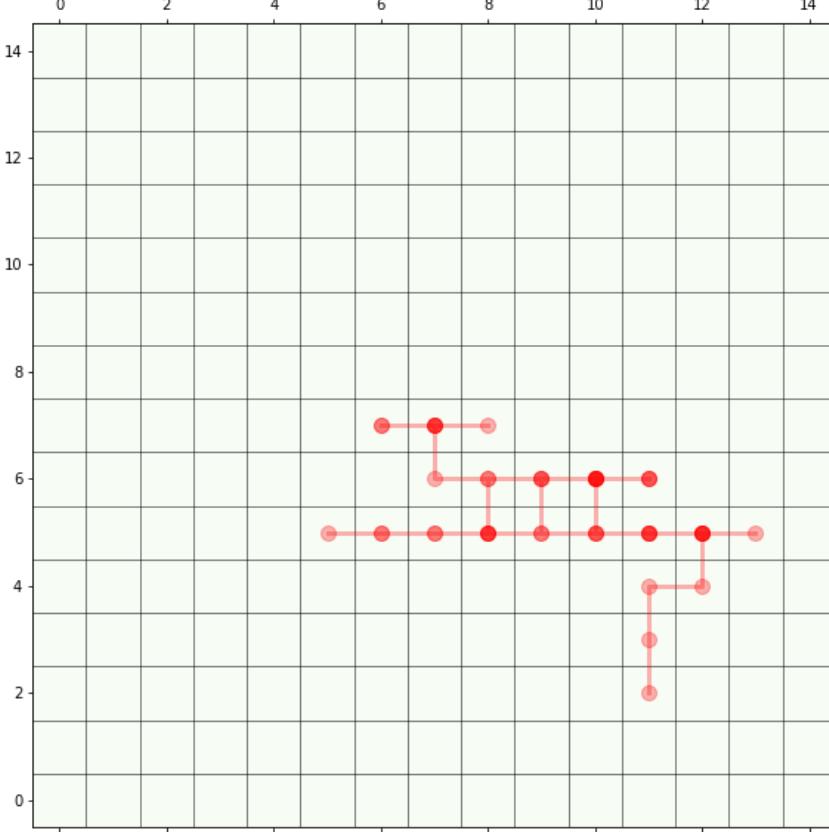


It randomly
selects one step

Excursion: let's walk!



In the next step it has the same stepping options

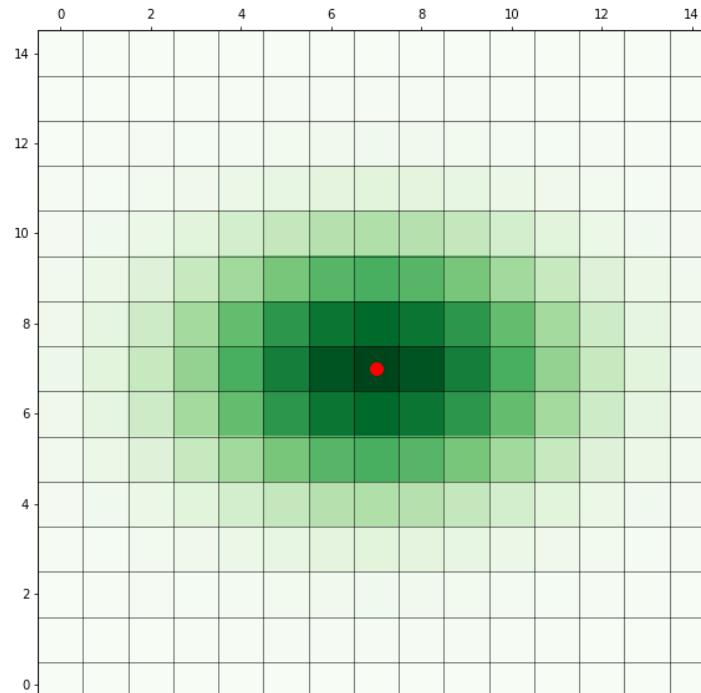


By iterating this procedure,
we get a trajectory

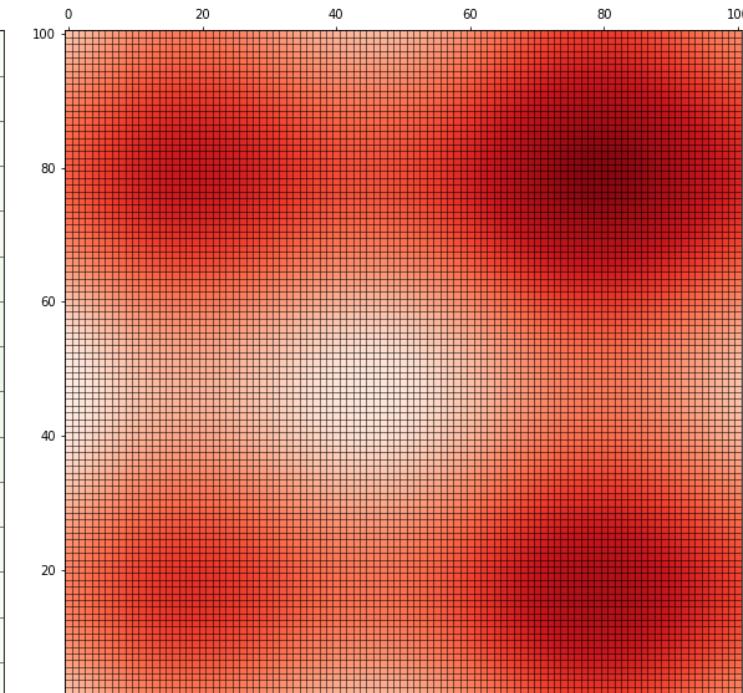
Excursion: let's walk!



- To make the behavior a little bit more interesting the walker in the exercise...
 - chooses 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/Step_0_Introduction/

Hands-on

Turn the walker code into a class



- 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 #2 on GitHub.



Thinking about Design

- Open the file that will call your function/class
- **What would the calling code look like in your wildest dreams?**
 - Which things will you do a lot -> should be easy visible interface
 - Which things should be modifiable from a user point of view?
 - As a rule of thumb, think about how it sounds when you read it aloud
 - “Giving orders”-style
 - “Telling a story”-style

From...

```
i, j = 100, 50 # initial position
sigma_i, sigma_j = 3, 4 # parameters of the next step map
size = 200 # size of the image
context_map = create_context_map(size, 'hills') # fixed context map

# Sample a next step 1000 times
trajectory = []
for _ in range(1000):
    i, j = sample_next_step(i, j, sigma_i, sigma_j, context_map)
    trajectory.append((i, j))
```

To... ?

Thinking about Design

- Open the file that will call your function/class
- **What would the calling code look like in your wildest dreams?**
 - Which things will you do a lot -> should be easy visible interface
 - Which things should be modifiable from a user point of view?
 - As a rule of thumb, think about how it sounds when you read it aloud
 - “Giving orders”-style
 - “Telling a story”-style

From...

```
i, j = 100, 50 # initial position
sigma_i, sigma_j = 3, 4 # parameters of the next step map
size = 200 # size of the image
context_map = create_context_map(size, 'hills') # fixed context map

# Sample a next step 1000 times
trajectory = []
for _ in range(1000):
    i, j = sample_next_step(i, j, sigma_i, sigma_j, context_map)
    trajectory.append((i, j))
```

To... ?

```
walker = Walker(sigma_i=3, sigma_j=4, ...)
# Sample a next step 1000 times
...
```

Hands-on

Turn the walker code into a class

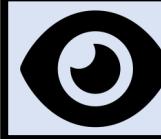
- Submit a PR for Issue #2 on GitHub.
- One way to start is to open the `Step_1_classes_exercise.ipynb`, and think about how you would like the simulation code to look like
What would that code look like in your wildest dreams?
- Now open the `walker.py` module and make the necessary changes
- Other tips in the `Step_1_classes_exercise.ipynb` notebook



Exercise

walker/Step_1_classes/

Plotting does not belong to the Walker



Live Coding

walker/Step_2_plotting/

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

class Walker:
    """ The Walker knows how to walk at random on a context map. """

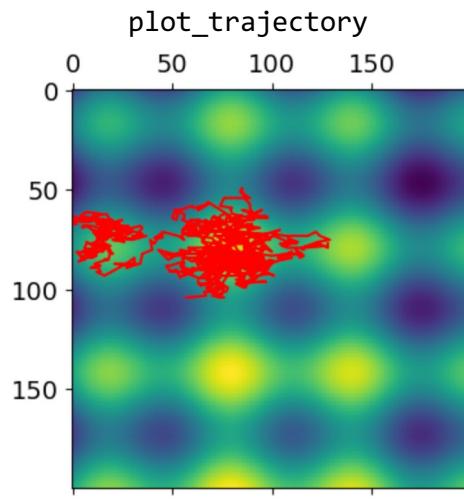
    def __init__(self, sigma_i, sigma_j, size, map_type='flat'):
        # ...

    def plot_trajectory(self, trajectory):
        """ Plot a trajectory over a context map. """
        trajectory = np.asarray(trajectory)
        plt.matshow(self.context_map)
        plt.plot(trajectory[:, 1], trajectory[:, 0], color='r')
        plt.show()

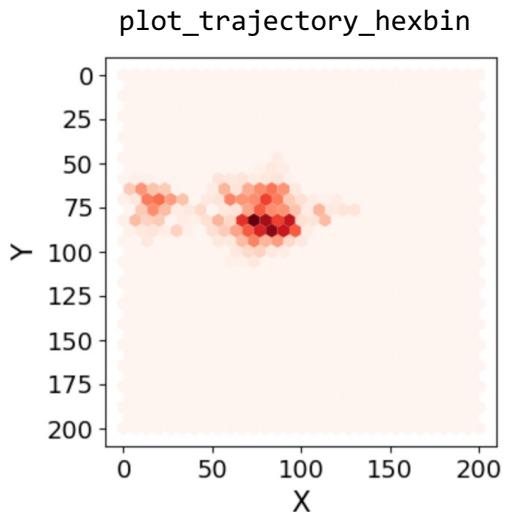
    def plot_trajectory_hexbin(self, trajectory):
        """ Plot an hexagonal density map of a trajectory. """
        trajectory = np.asarray(trajectory)
        plt.hexbin(
            trajectory[:, 1], trajectory[:, 0],
            gridsize=30, extent=(0, 200, 0, 200),
            edgecolors='none', cmap='Reds'
        )
        plt.gca().invert_yaxis()
        plt.xlabel('X')
        plt.ylabel('Y')
```

1. Changing or adding new way of plotting (e.g. by a colleague) would require modifying the Walker's code, without changing its behavior
2. Changing the Walker behavior will typically not modify the plotting code
3. Using your walker requires you to install matplotlib

These are smells of the fact that plotting varies independently of the Walker



VS.



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 a context map saved on file?

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

`if ... elif ... else` chain is often the smell of independent plug-in blocks

```
def add_block(blocks, size, type):  
    if type == 'circle':  
        # build circle of given size  
        block =   
    elif type == 'square':  
        # build square of given size  
        block =   
    elif type == 'triangle':  
        # build triangle of given size  
        block =   
    blocks.append(block)
```

```
add_block(blocks, size=11,  
          type='square')
```

`if ... elif ... else` chain is often the smell of independent plug-in blocks

```
def add_block(blocks, size, type):  
  
    if type == 'circle':  
        # build circle of given size  
        block =   
    elif type == 'square':  
        # build square of given size  
        block =   
    elif type == 'triangle':  
        # build triangle of given size  
        block =   
    blocks.append(block)
```

```
add_block(blocks, size=11,  
          type='square')
```

```
def build_circle(size):  
    """ Build a circle. """  
    return   
  
def build_square(size):  
    """ Build a square. """  
    return   
  
def build_triangle(size):  
    """ Build a triangle. """  
    return   
  
def add_block(blocks, block):  
    blocks.append(block)
```

```
block = build_square(size=11)  
add_block(blocks, block)
```

`if ... elif ... else` chain is often the smell of independent plug-in blocks

```
def add_block(blocks, size, type):  
  
    if type == 'circle':  
        # build circle of given size  
        block =   
    elif type == 'square':  
        # build square of given size  
        block =   
    elif type == 'triangle':  
        # build triangle of given size  
        block =   
    blocks.append(block)
```

```
add_block(blocks, size=11,  
          type='square')
```

```
def build_circle(size):  
    """ Build a circle. """  
    return   
  
def build_square(size):  
    """ Build a square. """  
    return   
  
def build_triangle(size):  
    """ Build a triangle. """  
    return   
  
def add_block(blocks, size, block_builder):  
    block = block_builder(size)  
    blocks.append(block)
```

```
add_block(blocks, size=11, build_square)
```

Hands-on

Move context map creation to separate module

- Move context map initialization to three functions in a separate `context_map.py` module. The functions take a `size` argument and return a `context_map` array
- Modify the constructor of `Walker` to take a `context_map` array instead of a `map_type`
- Modify the notebook to use the new code
- Submit a PR for Issue #3 on GitHub.



Exercise

Walker/Step_3_break_out ...

New requirement: we need to use different next-step proposals for different experiments

```
class Walker:  
    # ...  
  
    def sample_next_step(self, current_i, current_j, random_state=np.random):  
        """ Sample a new position for the walker. """  
        # Combine the next-step proposal with the context map to get a  
        # next-step probability map  
        next_step_map = self._next_step_proposal(current_i, current_j)  
        selection_map = self._compute_next_step_probability(next_step_map)  
  
        # Draw a new position from the next-step probability map  
        r = random_state.rand()  
        cumulative_map = np.cumsum(selection_map)  
        cumulative_map = cumulative_map.reshape(selection_map.shape)  
        i_next, j_next = np.argwhere(cumulative_map >= r)[0]  
  
        return i_next, j_next  
  
    def _next_step_proposal(self, current_i, current_j):  
        """ Create the 2D proposal map for the next step of the walker. """  
        # 2D Gaussian distribution , centered at current position,  
        # and with different standard deviations for i and j  
        grid_ii, grid_jj = self._grid_ii, self._grid_jj  
        sigma_i, sigma_j = self.sigma_i, self.sigma_j  
  
        rad = (  
            (((grid_ii - current_i) ** 2) / (sigma_i ** 2))  
            + (((grid_jj - current_j) ** 2) / (sigma_j ** 2))  
        )  
  
        p_next_step = np.exp(-(rad / 2.0)) / (2.0 * np.pi * sigma_i * sigma_j)  
        return p_next_step / p_next_step.sum()
```

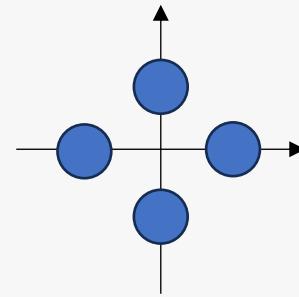
- We would like to run some experiments with a Gaussian next step proposal, some with a rectangular proposal, etc.
- **How can we do that?**

The inheritance solution

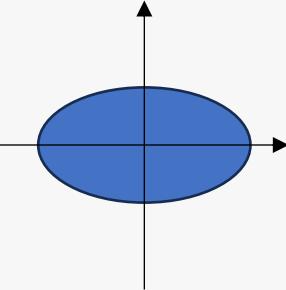
```
class Walker:  
    # ...  
  
    def _next_step_proposal(self, current_i, current_j):  
        """ Create the 2D proposal map for the next step of the walker. """  
        raise NotImplementedError(`_next_step_proposal` not implemented")
```



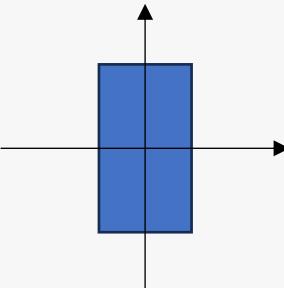
```
class JumpingWalker(Walker):  
    # ...  
  
    def _next_step_proposal(self, current_i, current_j):
```



```
class GaussianWalker(Walker):  
    # ...  
  
    def _next_step_proposal(self, current_i, current_j):
```



```
class RectangularWalker(Walker):  
    # ...  
  
    def _next_step_proposal(self, current_i, current_j):
```



The problem with inheritance

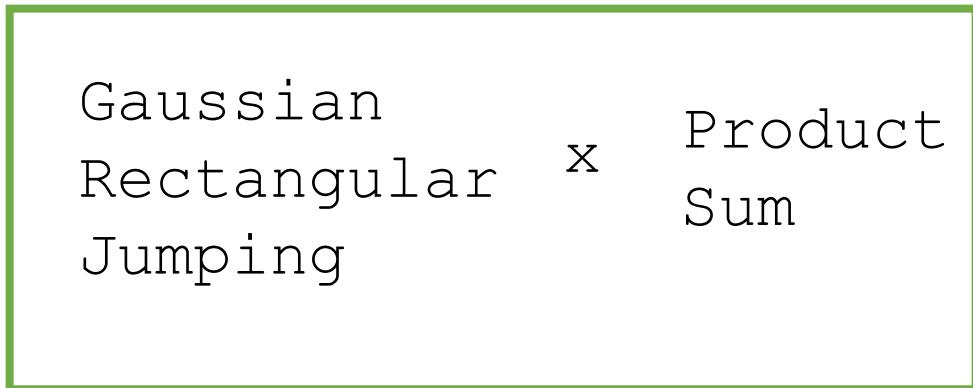
New requirement: the way of combining next step proposal and context map can also vary

```
class Walker:  
    # ...  
  
    def _compute_next_step_probability(self, next_step_map):  
        """ Compute the next step probability map from next step proposal and  
        context map. """  
        next_step_probability = next_step_map * self.context_map ←  
        next_step_probability /= next_step_probability.sum()  
        return next_step_probability
```

This combination can now be done through product, sum, or other

The problem with inheritance

The inheritance approach leads to a
combinatorial explosion of subclasses



```
class GaussianWalkerWithProductInteraction(Walker):
    def _next_step_proposal(self, current_i, current_j):
        # ...
    def _compute_next_step_probability(self, next_step_map):
        # ...

class GaussianWalkerWithSumInteraction(Walker):
    def _next_step_proposal(self, current_i, current_j):
        # ...
    def _compute_next_step_probability(self, next_step_map):
        # ...

class RectangularWalkerWithProductInteraction(Walker):
    def _next_step_proposal(self, current_i, current_j):
        # ...
    def _compute_next_step_probability(self, next_step_map):
        # ...

class RectangularWalkerWithSumInteraction(Walker):
    def _next_step_proposal(self, current_i, current_j):
        # ...
    def _compute_next_step_probability(self, next_step_map):
        # ...
```

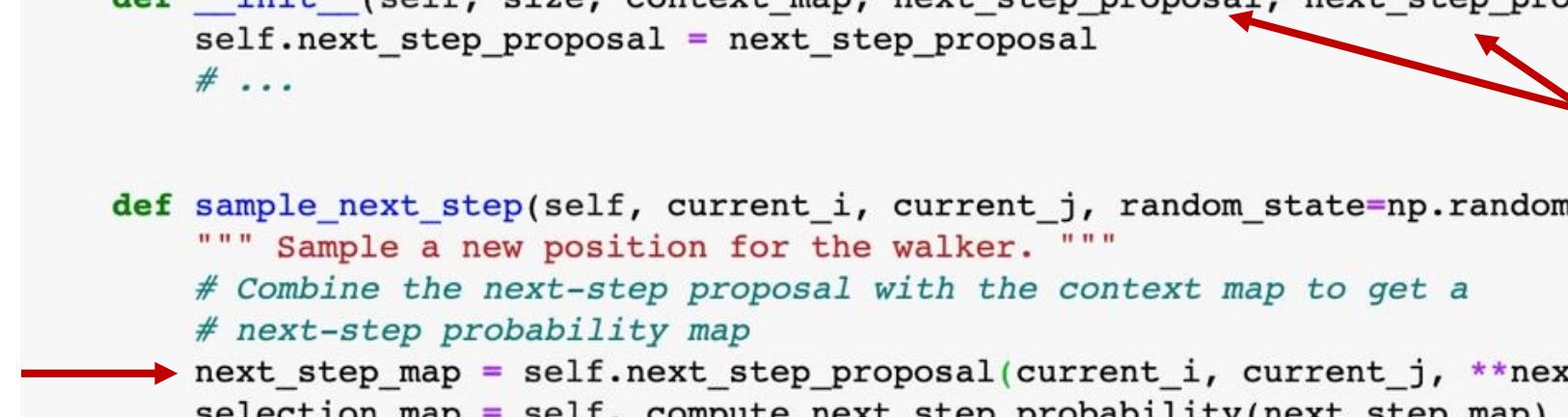
Passing varying behavior (e.g. functions) as an argument is often a better alternative

```
class Walker:
    def __init__(self, size, context_map, next_step_proposal, next_step_proposal_arguments):
        self.next_step_proposal = next_step_proposal
        # ...

    def sample_next_step(self, current_i, current_j, random_state=np.random):
        """ Sample a new position for the walker. """
        # Combine the next-step proposal with the context map to get a
        # next-step probability map
        next_step_map = self.next_step_proposal(current_i, current_j, **next_step_proposal_arguments)
        selection_map = self._compute_next_step_probability(next_step_map)

        # Draw a new position from the next-step probability map
        r = random_state.rand()
        cumulative_map = np.cumsum(selection_map)
        cumulative_map = cumulative_map.reshape(selection_map.shape)
        i_next, j_next = np.argwhere(cumulative_map >= r)[0]

    return i_next, j_next
```



In this way, we can **define the behavior independently** of the class, avoiding the combinatoric explosion

Hands-on

Implement two next-step proposals

- In a new module `next_step_proposals`, write two functions for two different next step proposals: a Gaussian one (it's the one currently in the Walker code) and a square one (you can find an example in the notebook)
- Modify the constructor of `Walker` to take a `next_step_proposal` function and a `next_step_proposal_arguments` dictionary as an input
- Modify the notebook to run a simulation first with the Gaussian proposal, then with the square one
- Submit a PR for Issue #4 on GitHub.

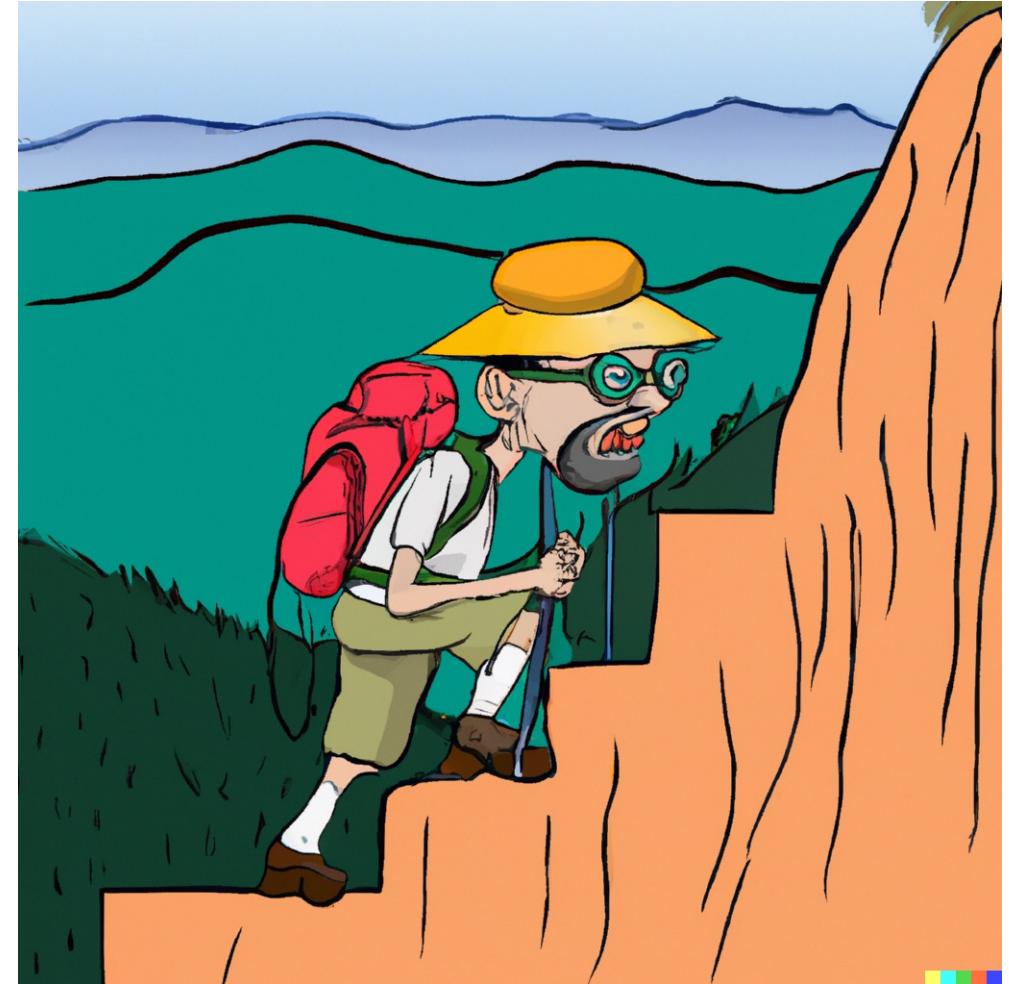


Exercise

walker/Step_4_break_out ...

The Walker: what have we achieved?

- We can run simulations with **different combinations** of context maps and next step proposals
- New context maps and next step proposals can be **contributed** by external people **without changing the code** in your package, or even knowing how it works
- We achieved **flexibility** and **openness to change**



Chapter 3: Separate what varies at the level of projects

The holy trinity of scientific computing

1. Provenance
2. Reproducibility
3. Organization



The holy trinity of scientific computing

1. Provenance
2. Reproducibility
3. Organization

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

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

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 specific for scientific use case.
 - Folder structure/Filenames?
 - Code generated meta-information files? (see next slide)
- In case of plots: data that generated the plots
 - For work in progress plots: plt.annotate()
 - Save .ipynb as pdf (with all the paths/version information in the notebook)
 - Could use metadata 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)

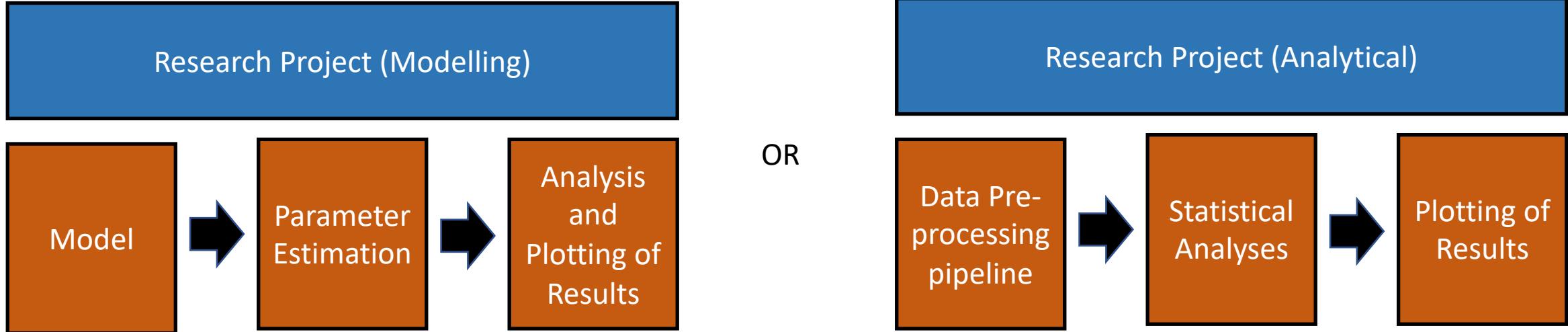
```
1 import git
2 import time
3
4 # lookup git repository
5 repo = git.Repo(search_parent_directories=True)
6 sha = repo.head.object.hexsha
7
8 # get current time
9 estim_time = (time.strftime("%Y%m%d-%H%M%S"))
10
11 with open('readme.txt', 'w') as f:
12     f.write(f'I estimated parameters at {estim_time}')
13     f.write(f'The git repo was at commit {sha}')
14
15
```

2. Reproducibility

- We also recommend using file names containing a version number or a time stamp, so that two subsequent runs do not overwrite previous results

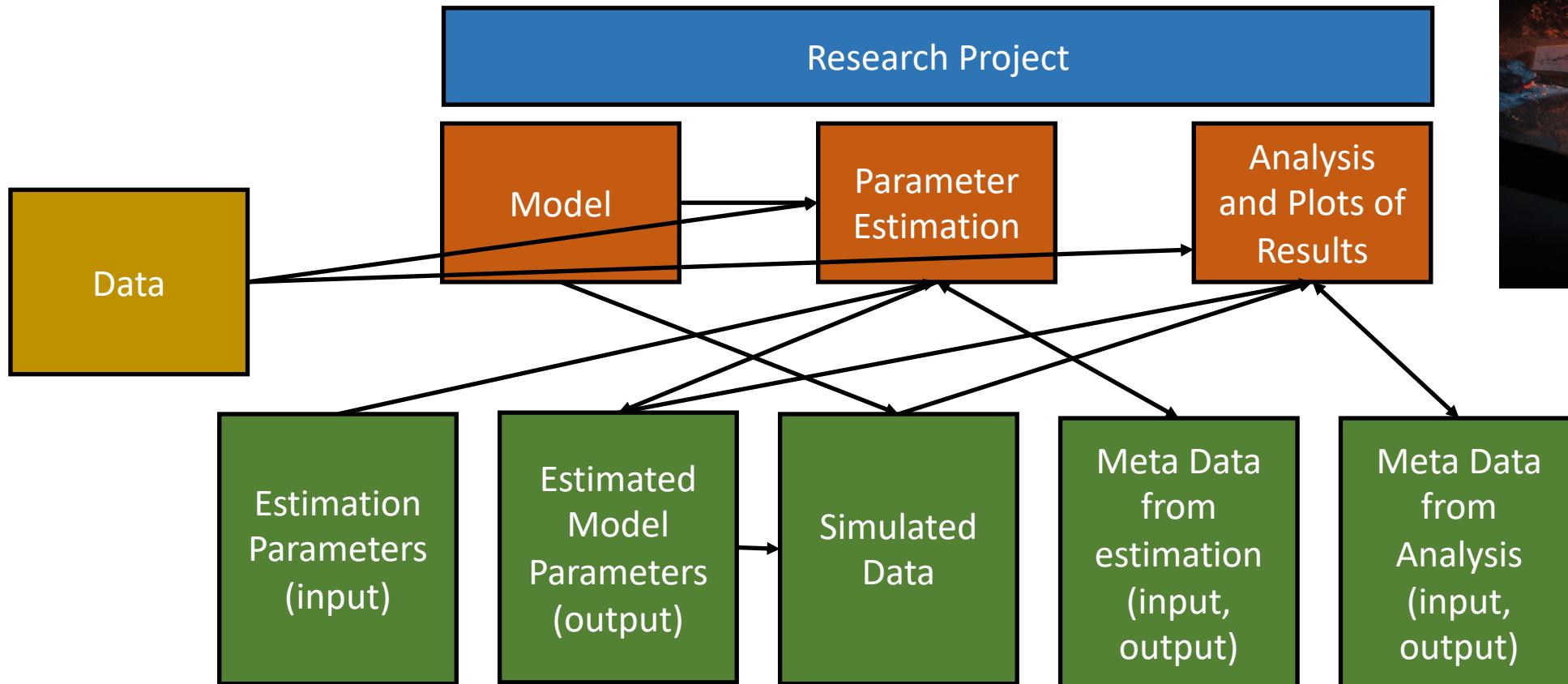
```
34     # STEP 4: Save the trajectory
35     curr_time = time.strftime("%Y%m%d-%H%M%S")
36     np.save(f"sim_{curr_time}", trajectory)
```

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

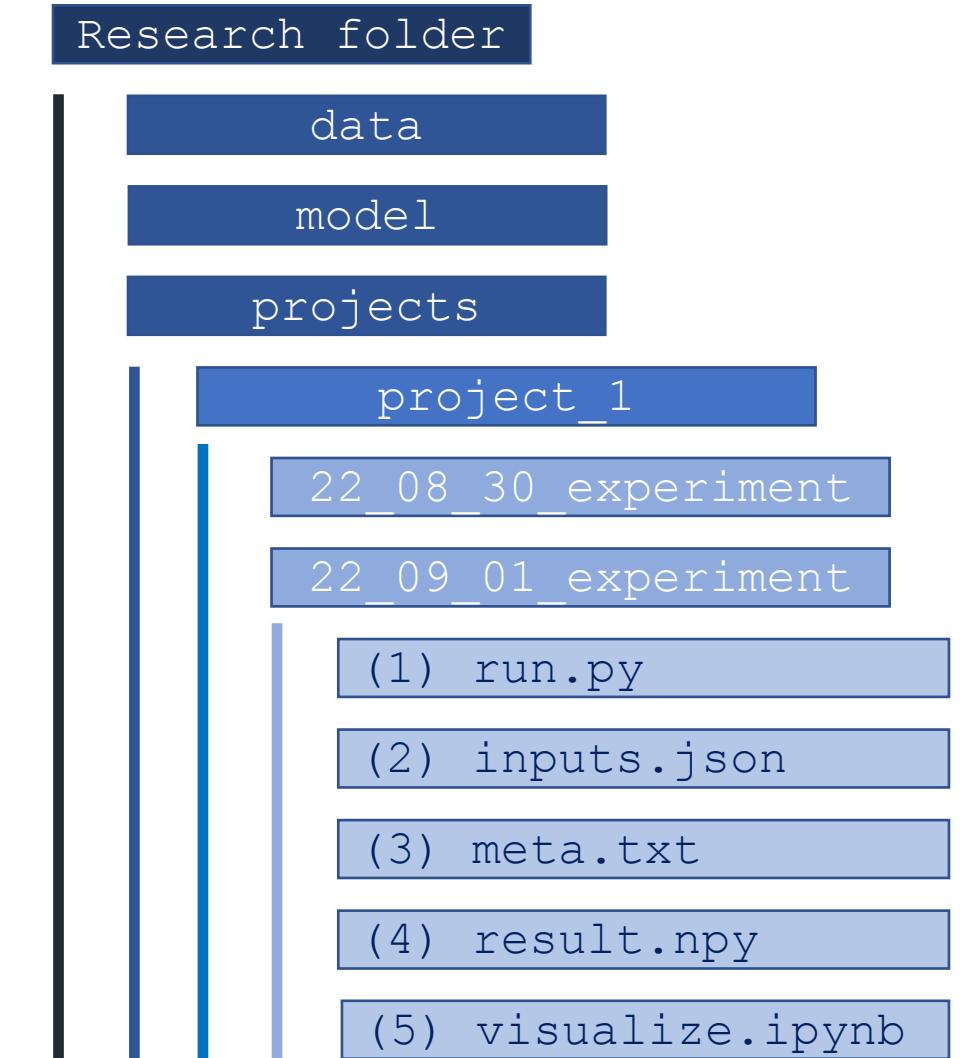


Each run has a bunch of associated data resulting folders and folders of data where no one knows which version of the code generated it or is using it!

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

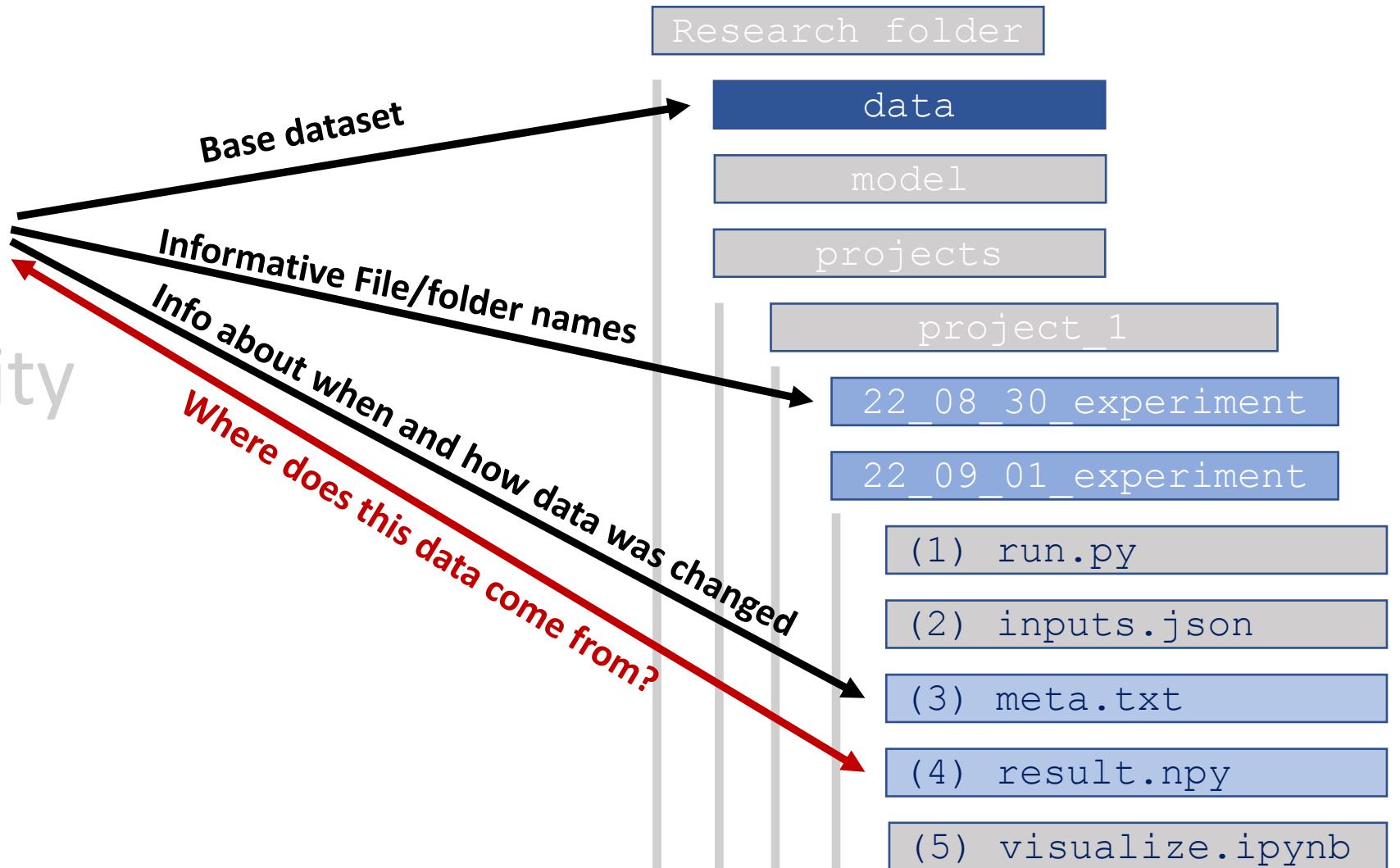


Organization

1. Provenance

2. Reproducibility

3. Organization

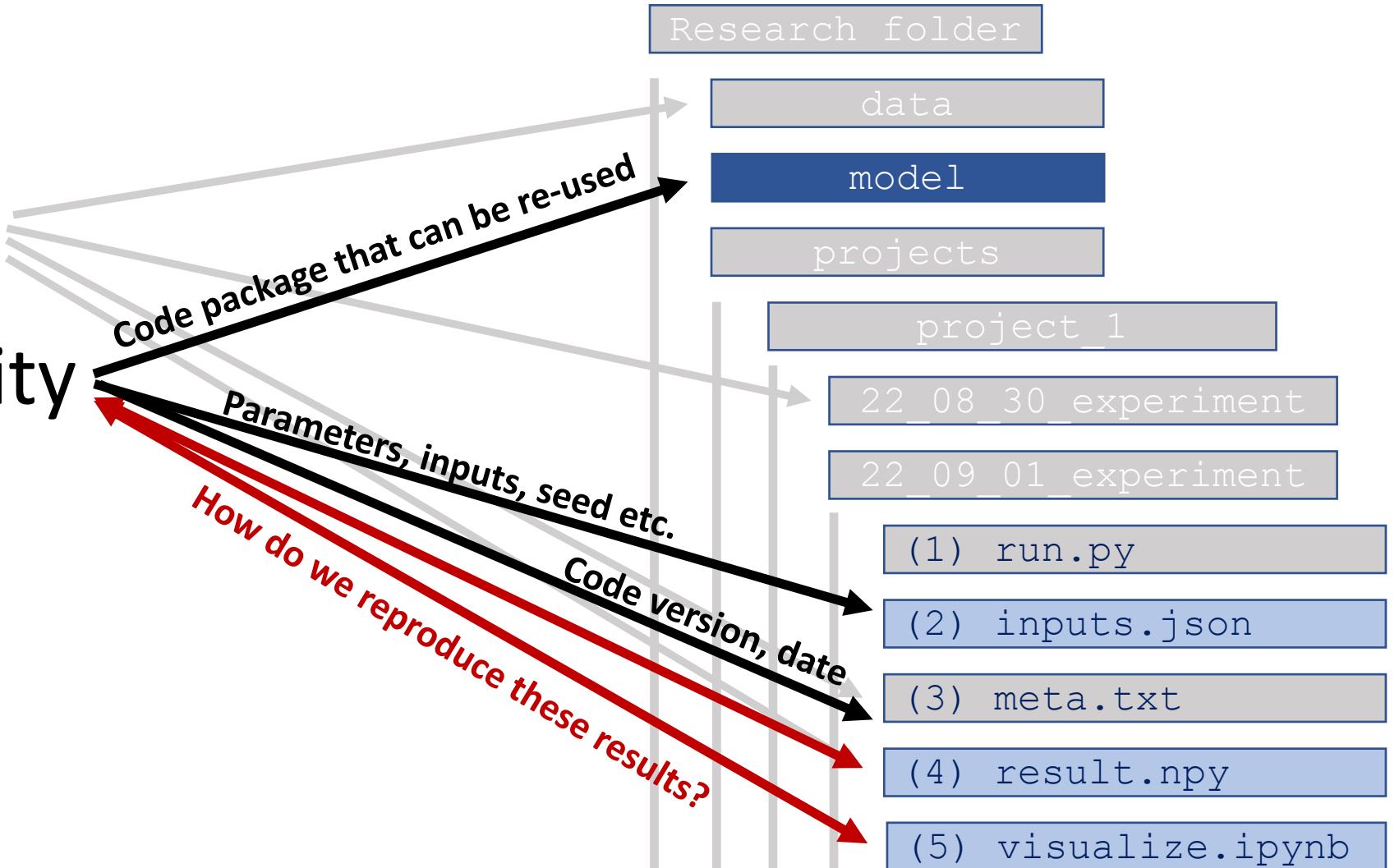


Organization

1. Provenance

2. Reproducibility

3. Organization

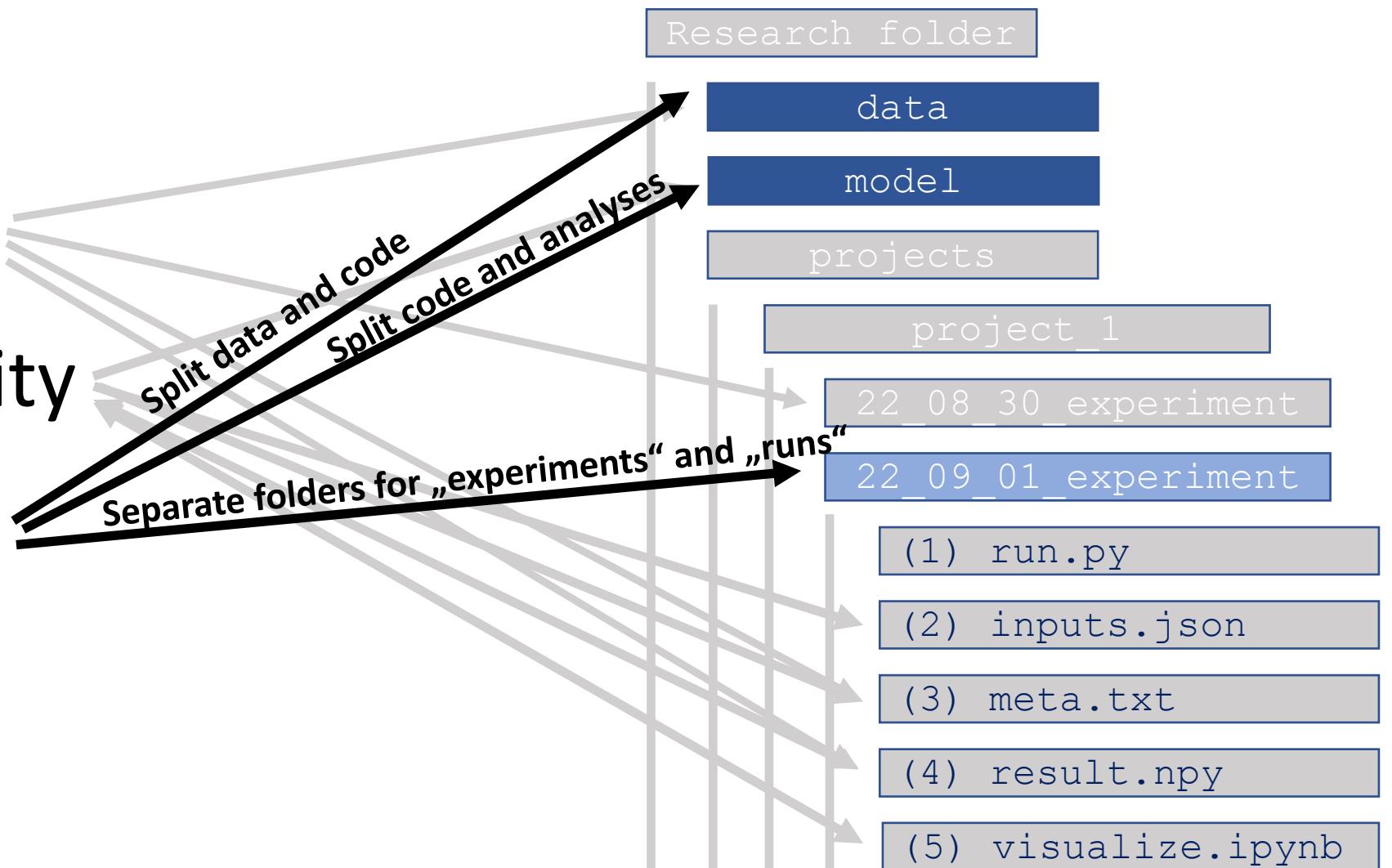


Organization

1. Provenance

2. Reproducibility

3. Organization



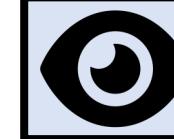
Hands On

Provenance and Reproducibility

Go through the following steps!

1. Submit PR for Issue #5 on GitHub
2. Complete the run.py script
 - In the file, at the top we give the desired parameters for the run
 - create a context map and walker (see previous exercises for reference)
 - simulate a trajectory (see previous exercises for reference)
3. Save the trajectory using `np.save()`, and also save some metadata
4. Run the run.py script twice and confirm the results are identical by plotting them using the provided notebook





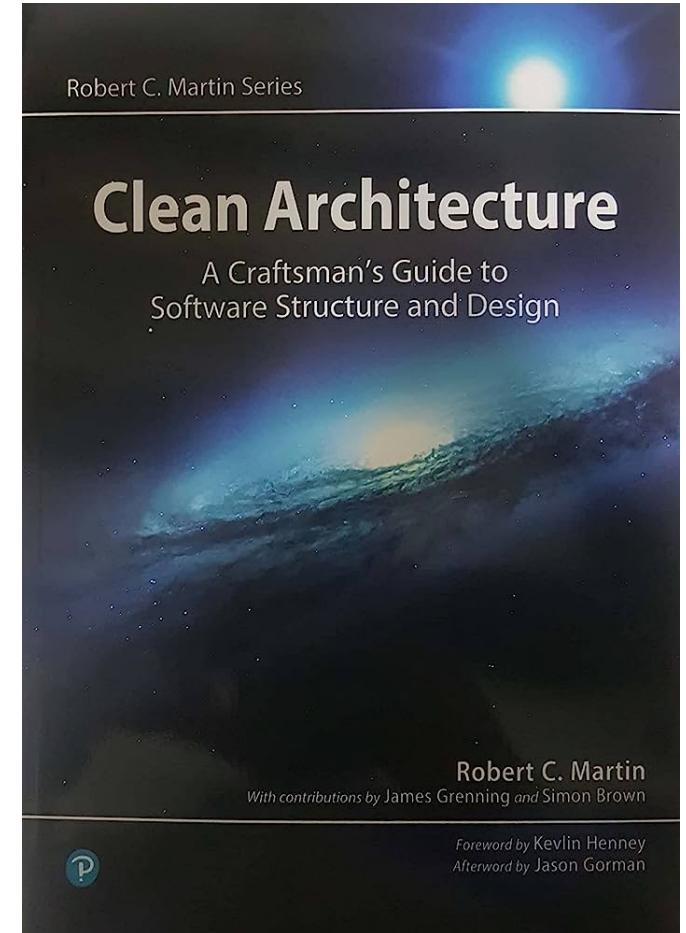
Live Coding

walker/Step_6_load_parameters_...

Provenance and Reproducibility

Where to go from here...

- Trust your nose! When your code smells, spend some time figuring out where the smell come from
- Don't get carried away: over-engineering counts as premature optimization



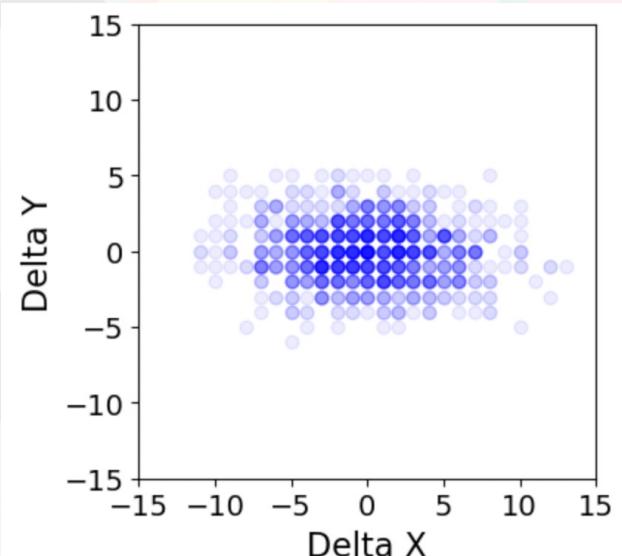
Thank you!



Hands-on

Move plotting code to a new module

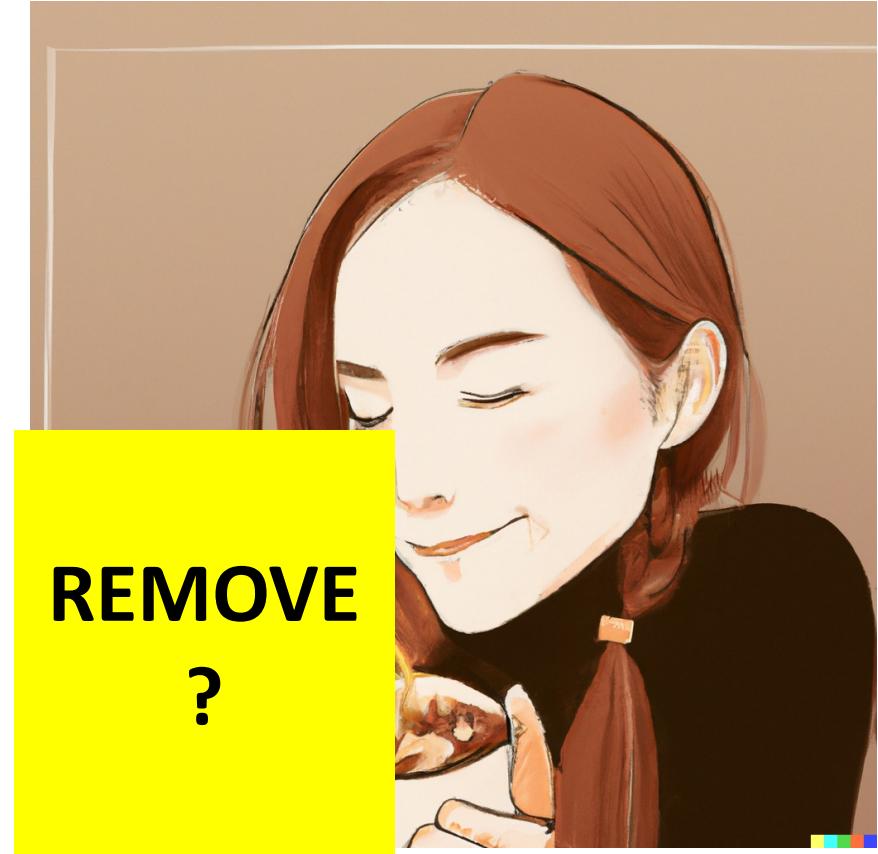
- Move plotting code to a separate plotting.py module
- Modify the notebook to import the plots and make sure it runs
- Add a new plotting function, `plot_delta_trajectory`, that plots a scatter plot of `delta_x` and `delta_y` for a (x, y) trajectory. Observe how we did not have to touch the `walker.py` file at all.
- Submit a PR for Issue #2 on GitHub.



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



Keep things open for

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 traversing, 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

