Lecture 08:

Scripts, Modules and Packages

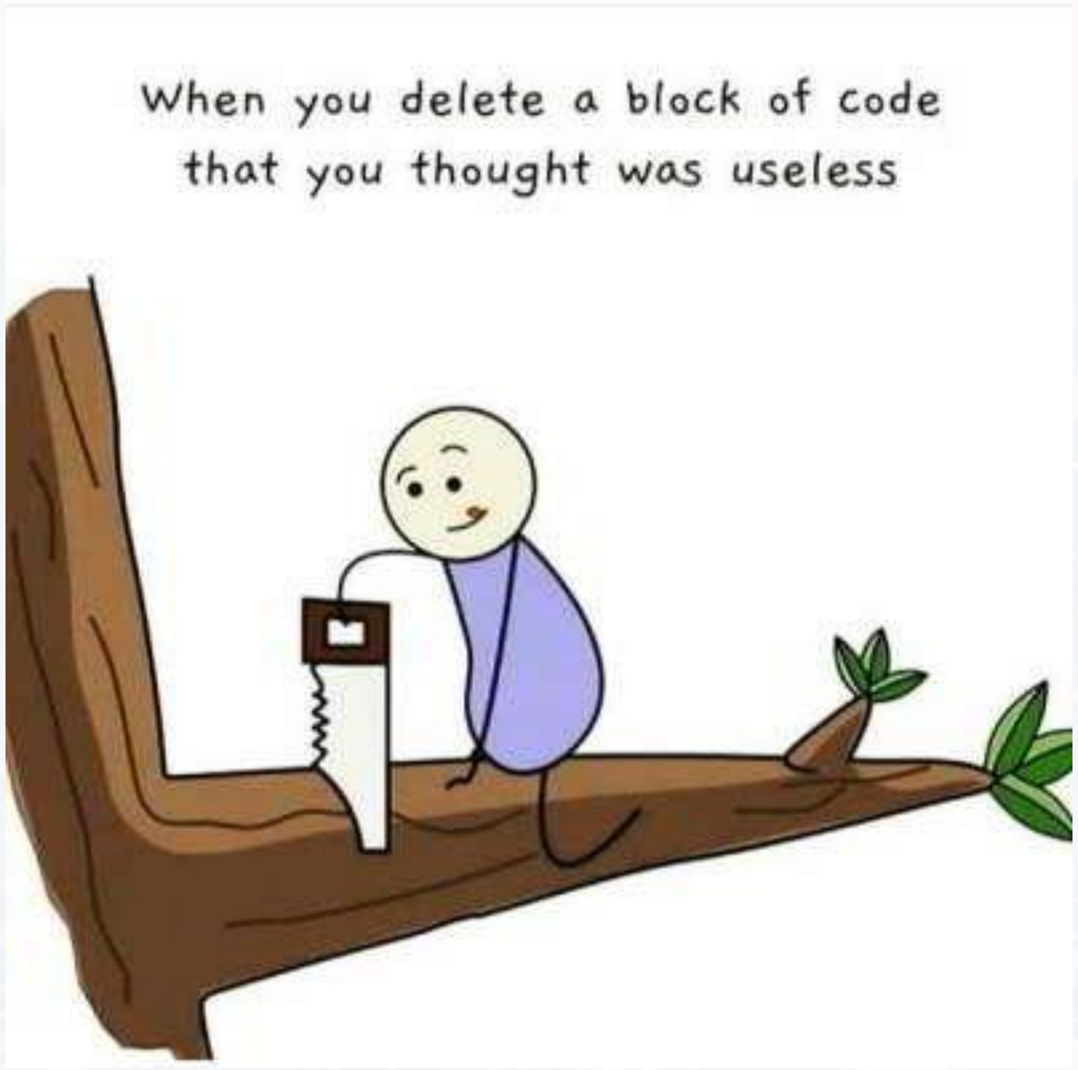
Markus Hohle

University California, Berkeley

Python for Molecular Sciences

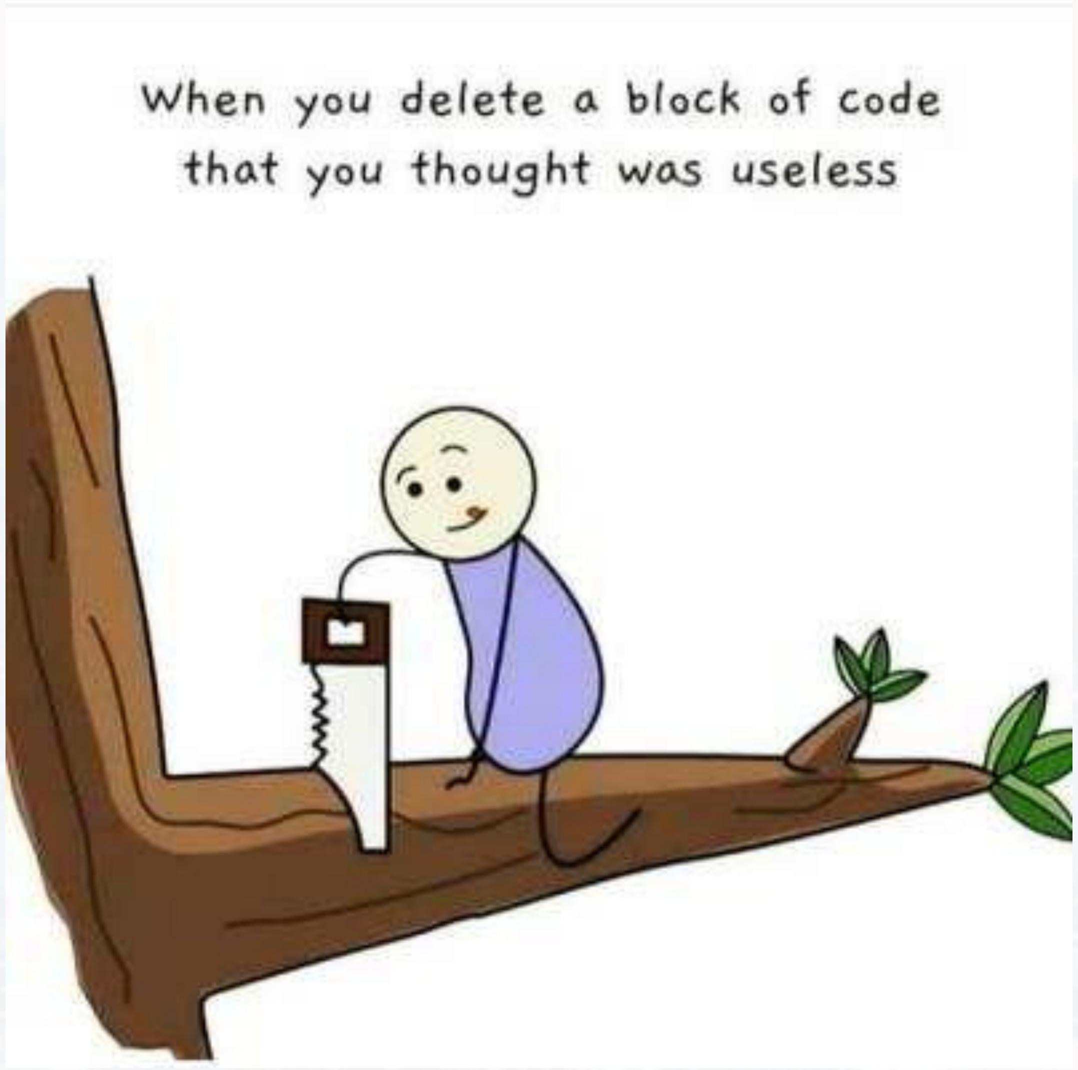
MSSE 272, 3 Units

Berkeley MSSE
MOLECULAR SCIENCE & SOFTWARE ENGINEERING


When you delete a block of code that you thought was useless

Outline

- Overall Structure
- `Classes`
- Let’s build a Package!


When you delete a block of code that you thought was useless

Outline

- **Overall Structure**
- `Classes`
- Let's build a Package!

Scripts, Modules and Packages

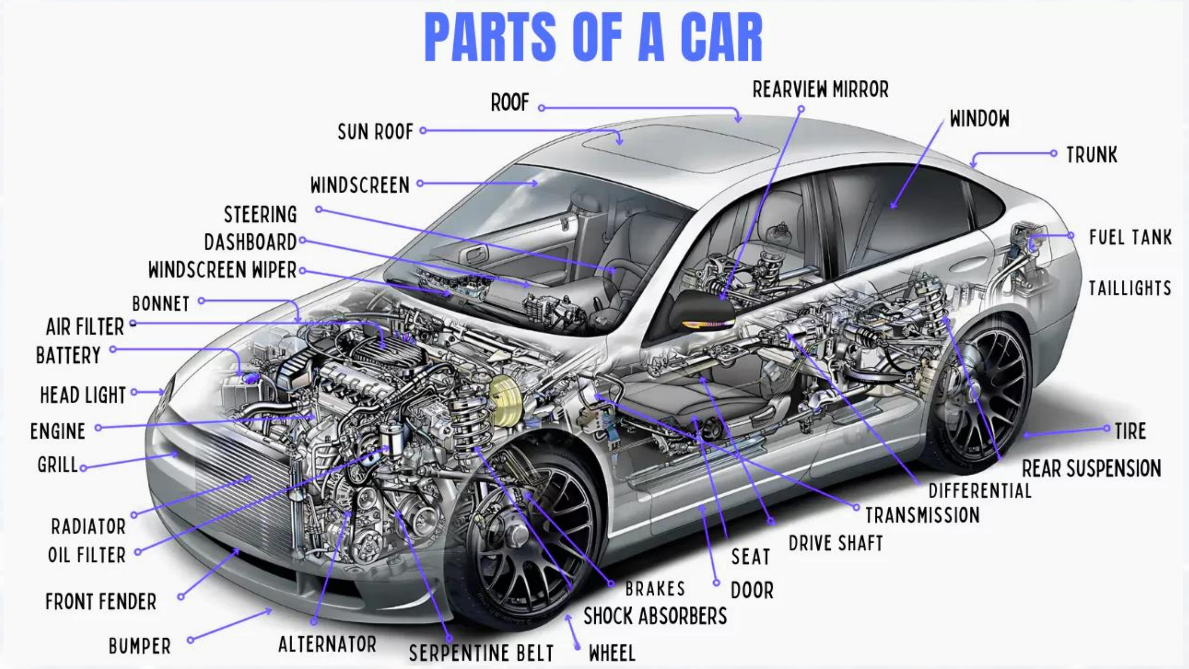
many different steps for a simulation or in a data analysis pipeline

- loading the data file
- extracting data
- analysis part
- visualization
- saving results

separate entities according to their functions

modularization or **refactoring**

advantages:
- removing redundancies
- maintainable
- readable
- scalable
- faster



credit: mendmotor

Scripts, Modules and Packages

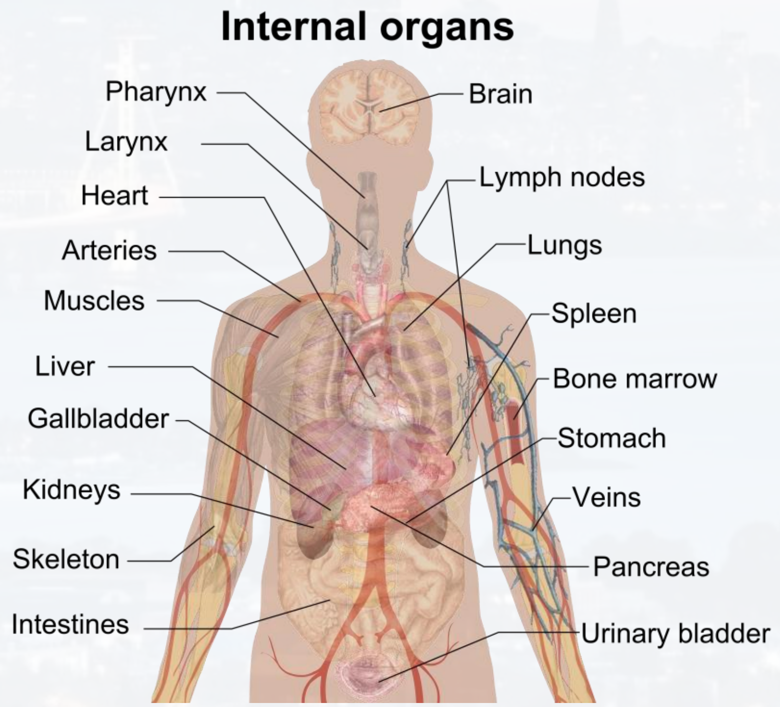
many different steps for a simulation or in a data analysis pipeline

- loading the data file
- extracting data
- analysis part
- visualization
- saving results

separate entities according to their functions

modularization or **refactoring**

advantages:
- removing redundancies
- maintainable
- readable
- scalable
- faster



credit: Wikipadia

Scripts, Modules and Packages

many different steps for a simulation or in a data analysis pipeline

for example: We need to call the same set of modules for many different tasks

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

idea: load libraries in the header of our package only once

We need to call the same plotting routine at different parts of the code

```
plt.scatter(x, y, size, marker = marker, color = color,\
            edgecolor = None, alpha = 0.2)
plt.xlim([-Lim*1.2, Lim*1.2])
plt.ylim([-Lim*1.2, Lim*1.2])
plt.title(title)
plt.show()
```

idea: write a separate plot routine

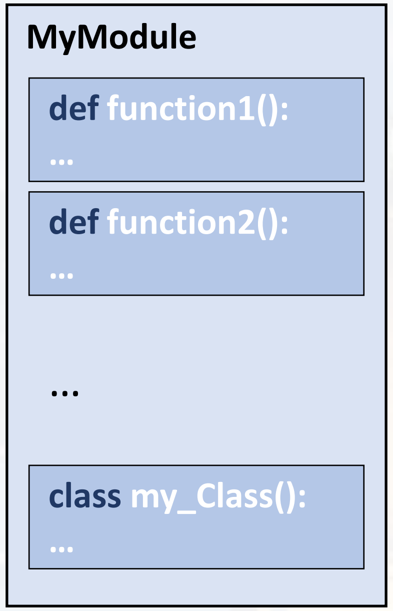
→ call it whenever needed

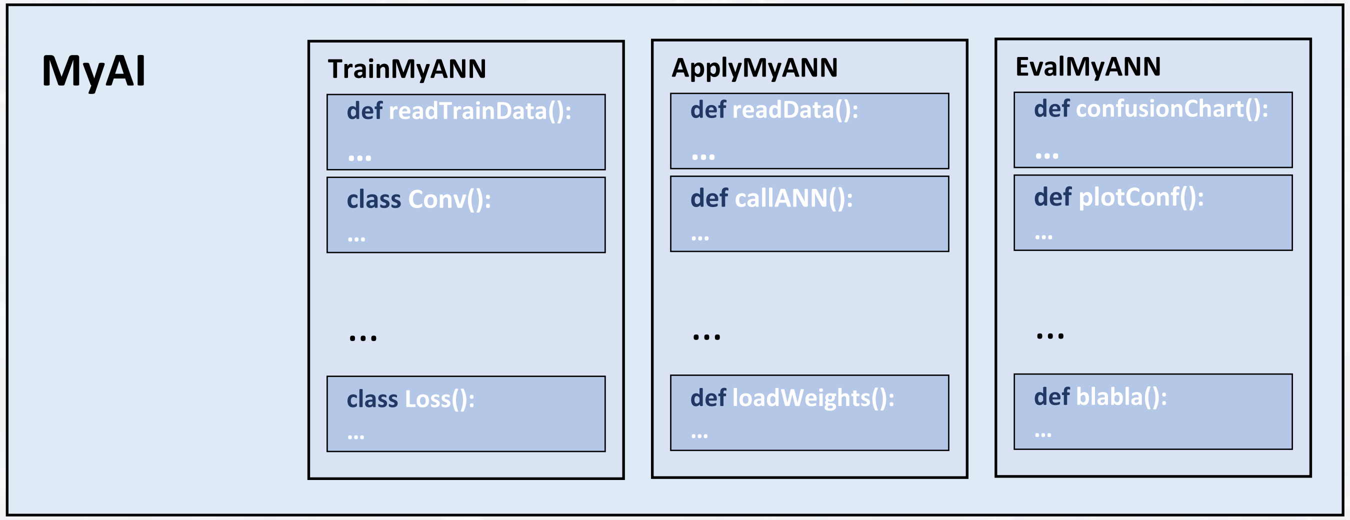
Scripts, Modules and Packages

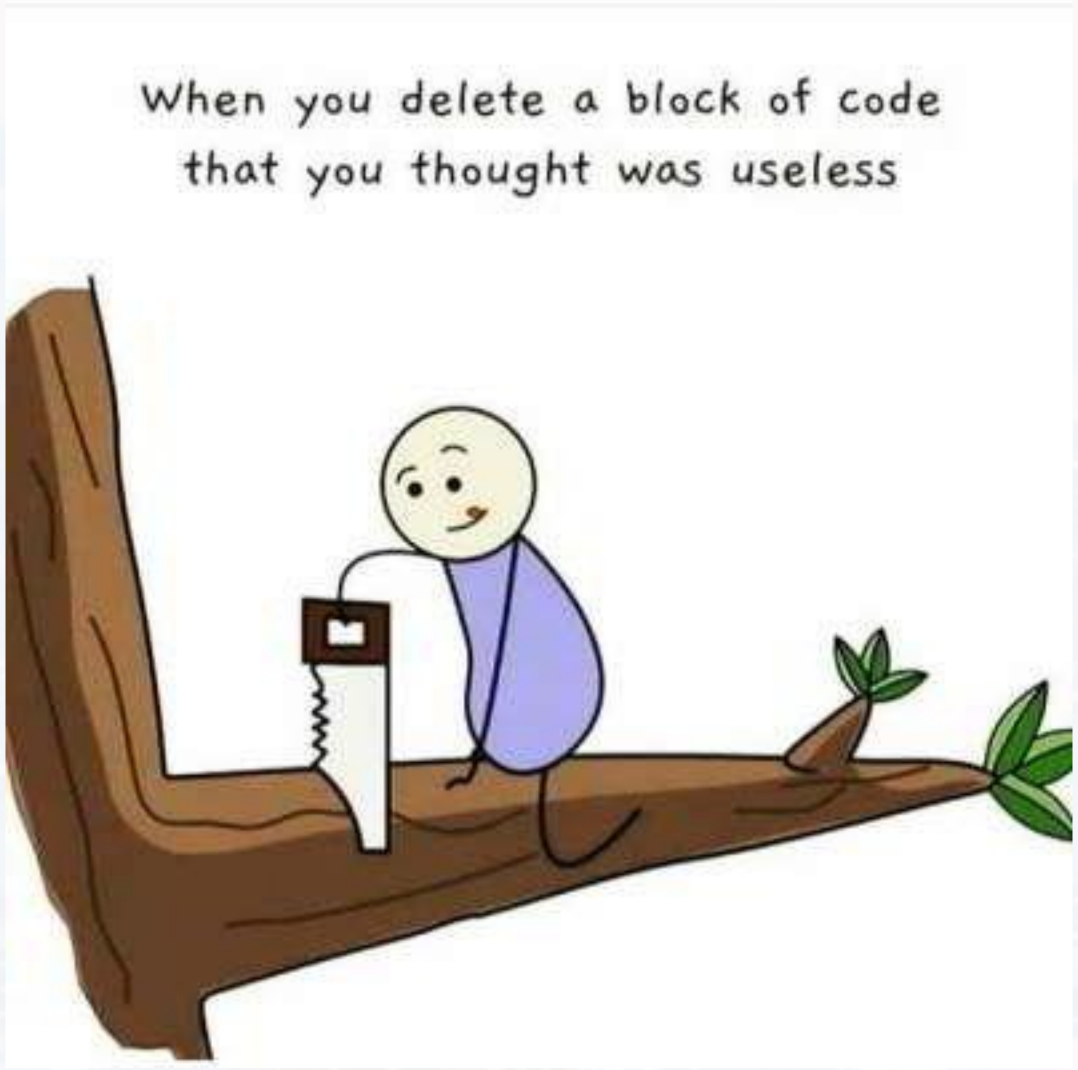
Overall Structure

many different steps for a simulation or in a data analysis pipeline

use `functions` and `classes` (see later)







Outline

- Overall Structure
- **Classes**
- Let’s build a Package!

Scripts, Modules and Packages

Task: writing a small package that one-hot encodes DNA and plots the result

recap: encoding a sequence with a dictionary

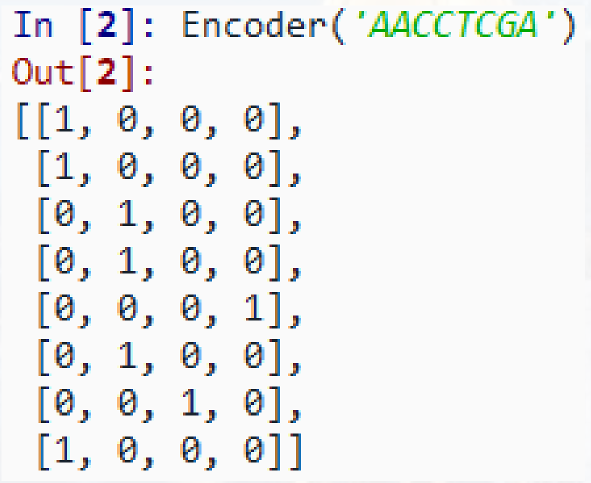
Before we introduce classes: let’s do it the old fashion way first!

Scripts, Modules and Packages

Task: writing a small package that one-hot encodes DNA and plots the result

recap: encoding a sequence with a dictionary

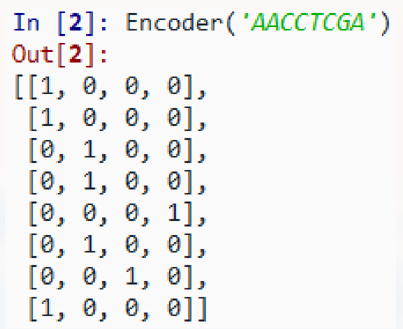
```
NT   = ['A', 'C', 'G', 'T']
Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]

Dict = {key: value for key, value in zip(NT, Code)}

Encoder = lambda Sequence: [Dict[s] for s in Sequence]
```



```
In [2]: Encoder('AACCTCGA')
Out[2]:
[[1, 0, 0, 0],
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 0, 1],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [1, 0, 0, 0]]
```

Scripts, Modules and Packages

Task: writing a small package that one-hot encodes DNA and plots the result



```
In [2]: Encoder('AACCTCGA')
Out[2]:
[[1, 0, 0, 0],
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 0, 1],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [1, 0, 0, 0]]
```

next step: plotting the one-hot encoded sequence

Seq = *'AACTCGTCTGACCTGTCGTATTGGCA'*

```
import numpy as np
import seaborn as sns

E = np.array(Encoder(Seq)).transpose()
```

1) calling the Encoder
2) for plotting:
   - turning output into an array
   - transposing it

```
sns.heatmap(E, cmap = "Blues", xticklabels = list(Seq), yticklabels = [None]*4)
```

Scripts, Modules and Packages

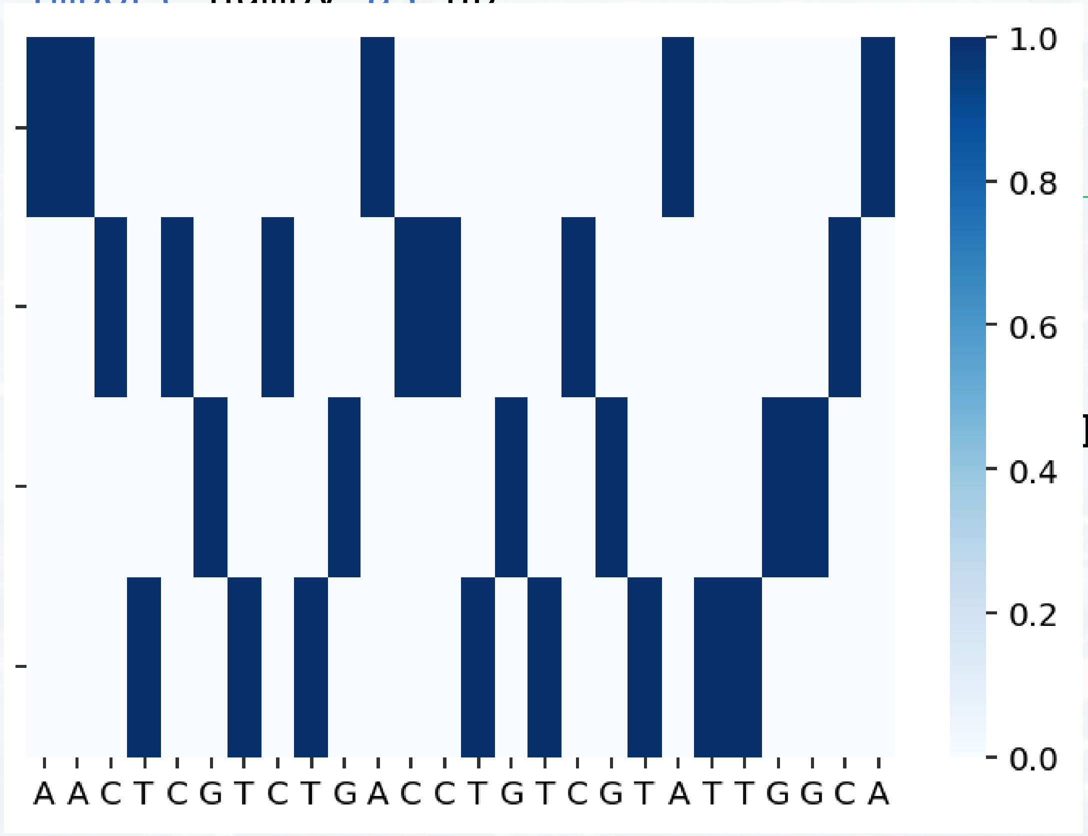
Task: writing a small package that one-hot encodes DNA and plots the result

next step: plotting the one-hot encoded sequence

Seq = *'AACTCGTCTGACCTGTCGTATTGGCA'*

```
In [2]: Encoder('AACCTCGA')
Out[2]:
[[1, 0, 0, 0],
 [1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 0, 1],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [1, 0, 0, 0]]
```

```
import numpy as np
```



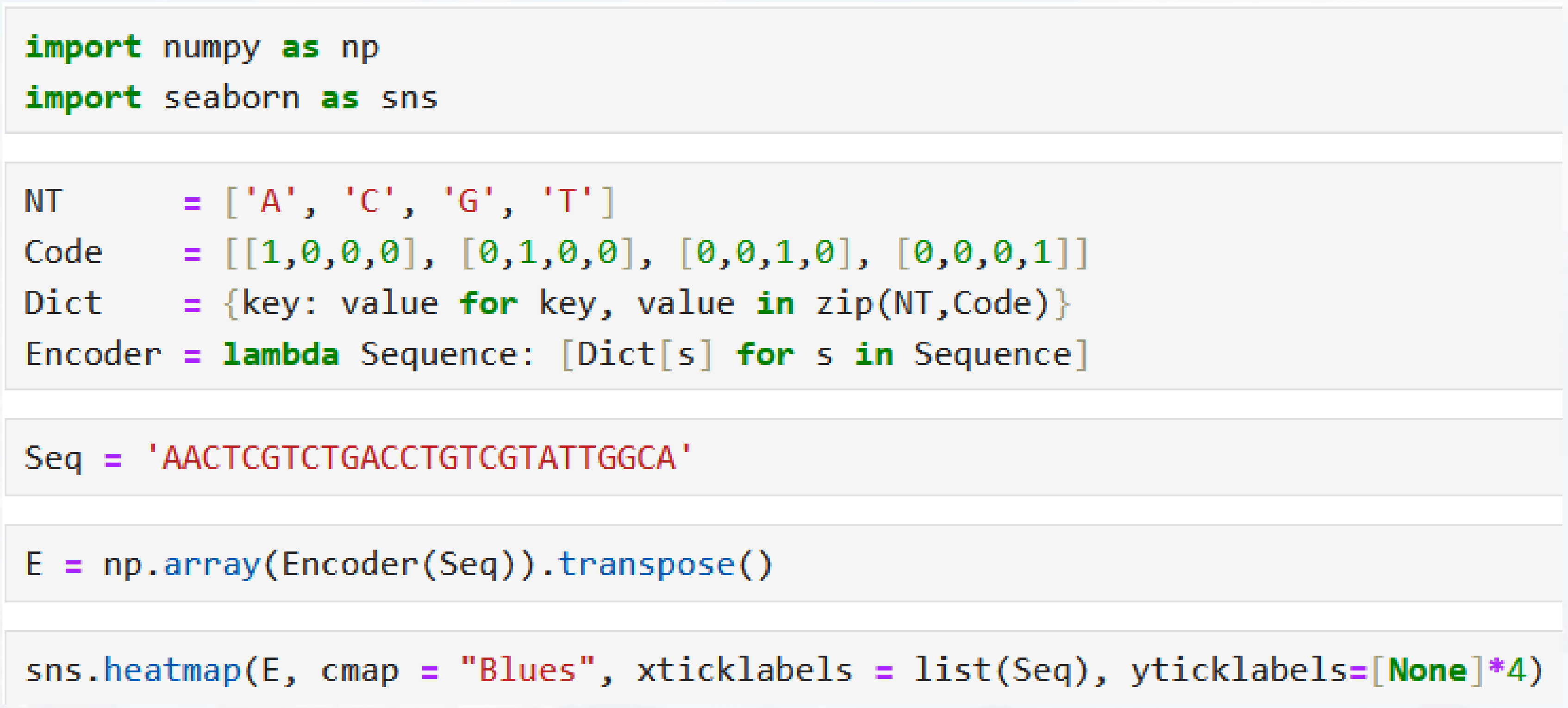
1) calling the Encoder
2) for plotting
   - turning output into an array
   - transposing it

```
ls = list(Seq), yticklabels = [None]*4)
```

Scripts, Modules and Packages

Task: writing a small package that one-hot encodes DNA and plots the result

If we ran this on a Jupyter Notebook:



```python
import numpy as np
import seaborn as sns
```

```python
NT      = ['A', 'C', 'G', 'T']
Code    = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]
Dict    = {key: value for key, value in zip(NT,Code)}
Encoder = lambda Sequence: [Dict[s] for s in Sequence]
```

```python
Seq = 'AACTCGTCTGACCTGTCGTATTGGCA'
```

```python
E = np.array(Encoder(Seq)).transpose()
```

```python
sns.heatmap(E, cmap = "Blues", xticklabels = list(Seq), yticklabels=[None]*4)
```

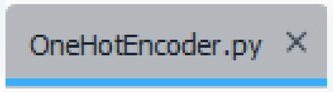
EncodeSequential.ipynb

Scripts, Modules and Packages

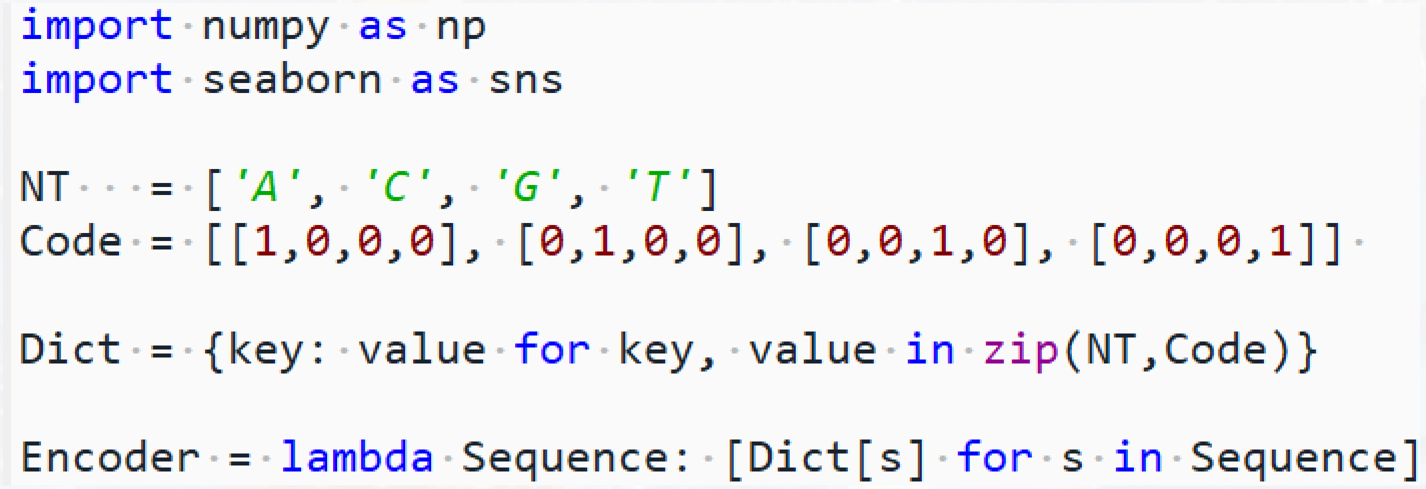
Task: writing a small package that one-hot encodes DNA and plots the result

Now using `classes`:

1) Open a Python Script
2) Save it under the name: `OneHotEncoder.py`


OneHotEncoder.py ×

3) refactoring: we need to call numpy and `seaborn` only once!
   we need to define our `dict` and the `Encoder` only once!



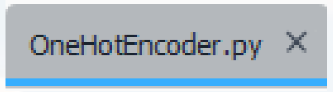
```python
import numpy as np
import seaborn as sns

NT   = ['A', 'C', 'G', 'T']
Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]] 

Dict = {key: value for key, value in zip(NT,Code)}

Encoder = lambda Sequence: [Dict[s] for s in Sequence]
```

Scripts, Modules and Packages

Task: writing a small package that one-hot encodes DNA and plots the result

Now using `classes`:

1) Open a Python Script
2) Save it under the name: `OneHotEncoder.py`



3) refactoring: we need to call numpy and `seaborn` only once!
   we need to define our `dict` and the `Encoder` only once!
   defining the plotting routine as independent `function` using `def`

```python
import numpy as np
import seaborn as sns

NT   = ['A', 'C', 'G', 'T']
Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]

Dict = {key: value for key, value in zip(NT,Code)}

Encoder = lambda Sequence: [Dict[s] for s in Sequence]

def MyPlotRoutine(E, Seq):
    sns.heatmap(E, cmap = "Blues", xticklabels = list(Seq),\
                ytickl‌abels = [None]*4)
```

don't forget the input arguments!

Scripts, Modules and Packages

Task: writing a small package that one-hot encodes DNA and plots the result

Now using classes: 4) writing a class

```
class EncodeMySeq():
```

reading and converting the sequence when we call the class

creating the plot by calling the plot routine

Scripts, Modules and Packages

Task: writing a small package that one-hot encodes DNA and plots the result

Now using classes: 4) writing a class

```
class EncodeMySeq():

    def __init__(    , Seq):

        E     = np.array(Encoder(Seq)).transpose()

    def PlotMySeq(     ):
        MyPlotRoutine(      E,    Seq)
```

reading and converting the sequence when we call the class

creating the plot by calling the plot routine

don't forget the input arguments!

How do we handover these arguments from **__init__** to **PlotMySeq**?

Scripts, Modules and Packages

Task: writing a small package that one-hot encodes DNA and plots the result

Now using `classes`: 4) writing a `class`

```
class EncodeMySeq():

    def __init__(self, Seq):

        self.E   = np.array(Encoder(Seq)).transpose()
        self.Seq = Seq

    def PlotMySeq(self):
        MyPlotRoutine(self.E,self.Seq)
```

reading and converting the sequence when we call the class

creating the plot by calling the plot routine

don't forget the input arguments!

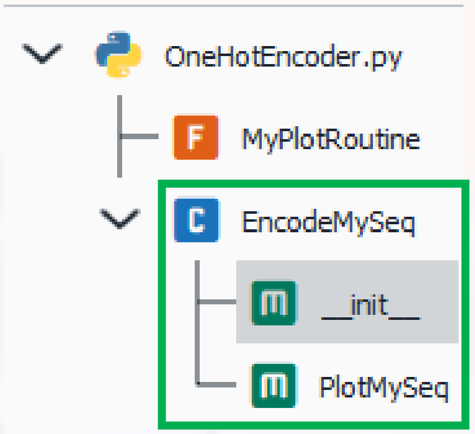
How do we handover these arguments from **__init__** to **PlotMySeq**?

→ using *self*

Scripts, Modules and Packages

our code now:

```
import numpy as np
import seaborn as sns

NT   = ['A', 'C', 'G', 'T']
Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]

Dict = {key: value for key, value in zip(NT, Code)}

Encoder = lambda Sequence: [Dict[s] for s in Sequence]

def MyPlotRoutine(E, Seq):
    sns.heatmap(E, cmap = "Blues", xticklabels = list(Seq),\
                ytickLabels = [None]*4)

class EncodeMySeq():

    def __init__(self, Seq):

        self.E   = np.array(Encoder(Seq)).transpose()
        self.Seq = Seq

    def PlotMySeq(self):
        MyPlotRoutine(self.E, self.Seq)
```



our class has two methods

Scripts, Modules and Packages

our code now:

a final adjustment!

```
import numpy as np
import seaborn as sns


NT   = ['A', 'C', 'G', 'T']
Code = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]

Dict = {key: value for key, value in zip(NT, Code)}

Encoder = lambda Sequence: [Dict[s] for s in Sequence]

def MyPlotRoutine(E, Seq):
    sns.heatmap(E, cmap = "Blues", xticklabels = list(Seq),\
                yticklabels = [None]*4)

class EncodeMySeq():

    def __init__(self, Seq):

        self.E   = np.array(Encoder(Seq)).transpose()
        self.Seq = Seq

    def PlotMySeq(self):
        MyPlotRoutine(self.E, self.Seq)
```

Scripts, Modules and Packages

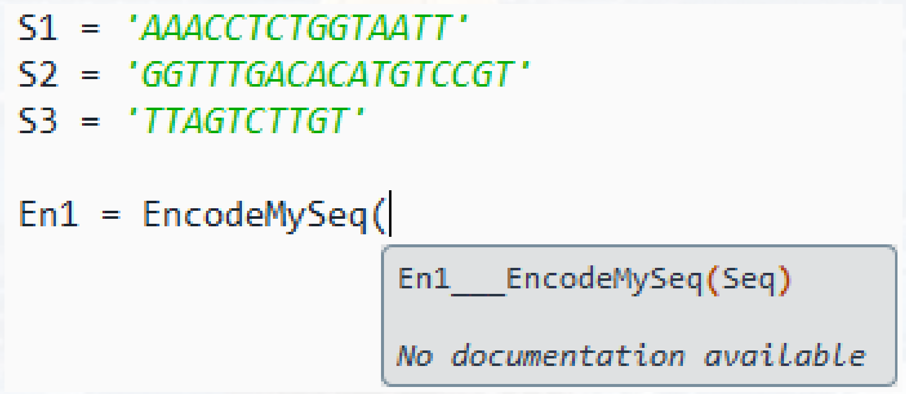
Let’s try to understand how a `class` works!

1) Compiling our package `OneHotEncoder.py`

2) Creating three sequences

```
S1 = 'AAACCTCTGGTAATT'
S2 = 'GGTTTGACACATGTCCGT'
S3 = 'TTAGTCTTGT'
```

3) **initializing** an instance (object) of the class `EncodeMySeq`



```
S1 = 'AAACCTCTGGTAATT'
S2 = 'GGTTTGACACATGTCCGT'
S3 = 'TTAGTCTTGT'

En1 = EncodeMySeq(
```

```
En1___EncodeMySeq(Seq)

No documentation available
```

```
En1 = EncodeMySeq(S1)
```

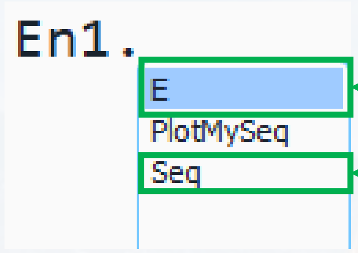
creating an instance (En1) of EncodeMySeq
→ runs the `__init__` method

Scripts, Modules and Packages

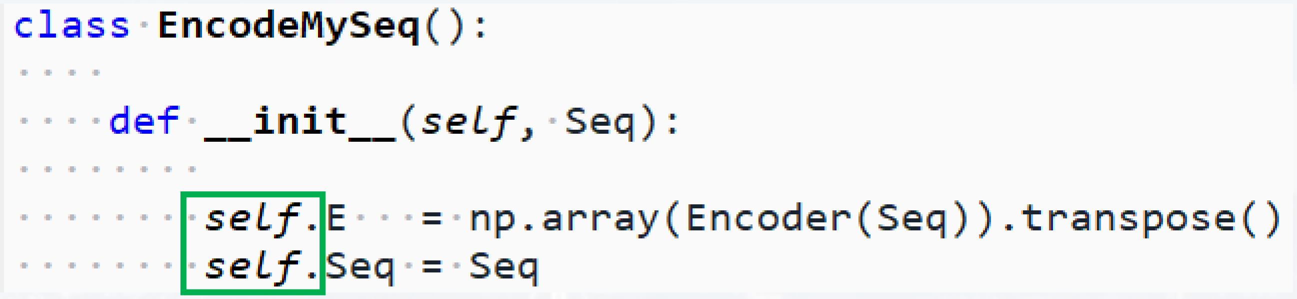
Let’s try to understand how a `class` works!

```
En1 = EncodeMySeq(S1)
```

creating an instance (En1) of EncodeMySeq
→ runs the `__init__` method



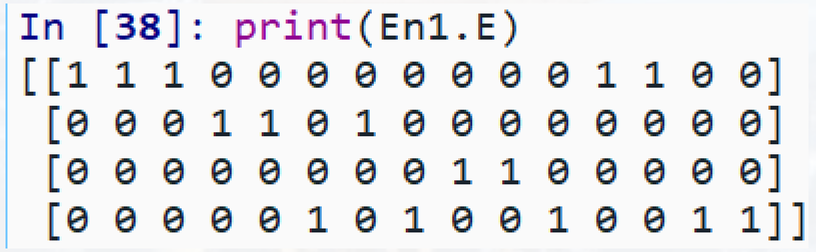
```
En1.
  E
  PlotMySeq
  Seq
```



```
class EncodeMySeq():

    def __init__(self, Seq):

        self.E   = np.array(Encoder(Seq)).transpose()
        self.Seq = Seq
```

`__init__` preformed the encoding!

E, `PlotMySeq` and Seq are so called **attributes** of the `class` EncodeMySeq



```
In [38]: print(En1.E)
[[1 1 1 0 0 0 0 0 0 0 0 1 1 0 0]
 [0 0 0 1 1 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 1 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 0 0 1 0 0 1 1]]
```

Scripts, Modules and Packages

E, PlotMySeq and Seq are so called **attributes** of the `class` EncodeMySeq

```
In [38]: print(En1.E)
[[1 1 1 0 0 0 0 0 0 0 0 1 1 0 0]
 [0 0 0 1 1 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 1 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 0 0 1 0 0 1 1]]
```

attributes can be functions or variables (and many other types too)

```
In [39]: dir(En1)
Out[39]:
['E',
 'PlotMySeq',
 'Seq',
```

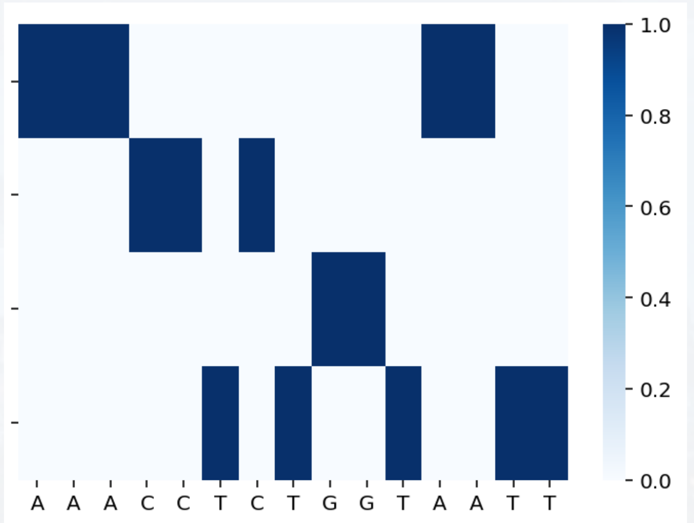
```
type(En1.Seq)
str
```

```
type(En1.E)
numpy.ndarray
```

```
type(En1.PlotMySeq)
method
```

running the plotting routine

```
En1.PlotMySeq()
```

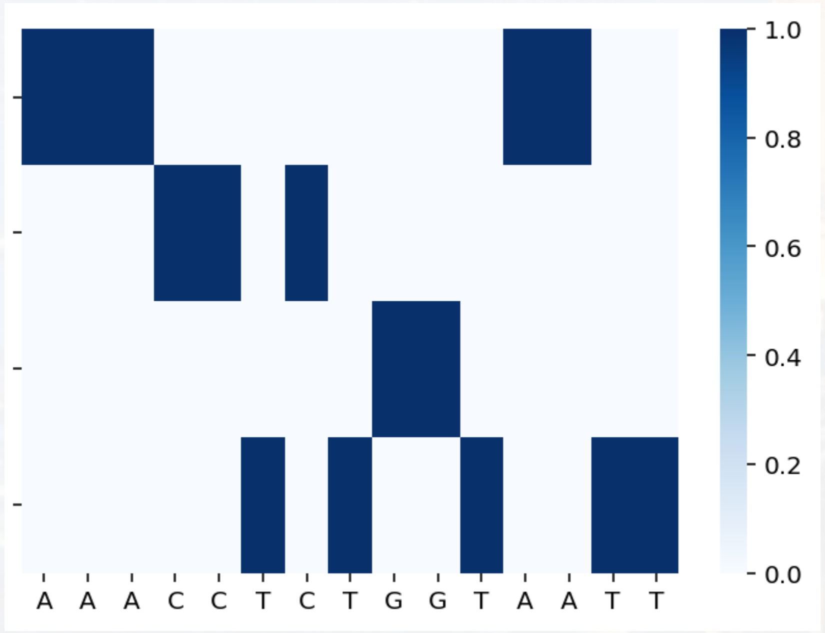


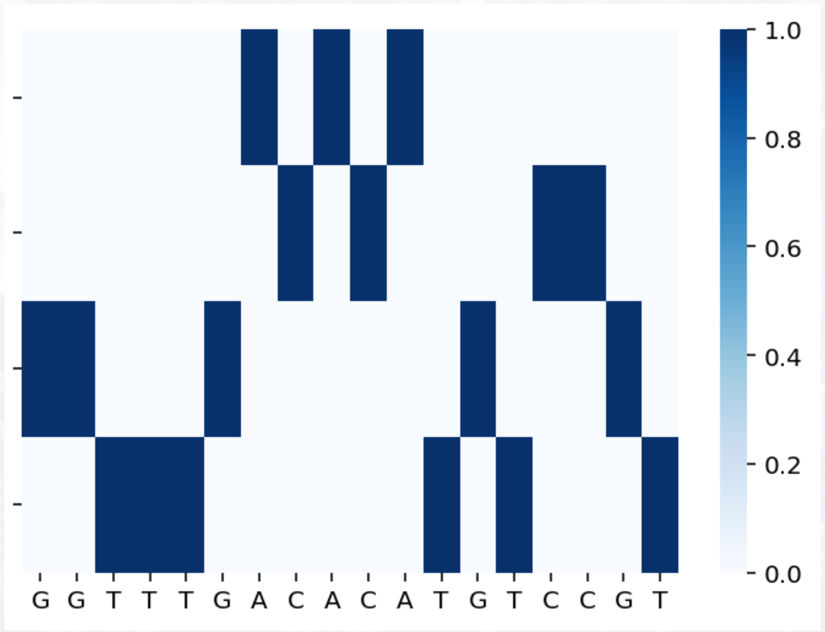
Scripts, Modules and Packages

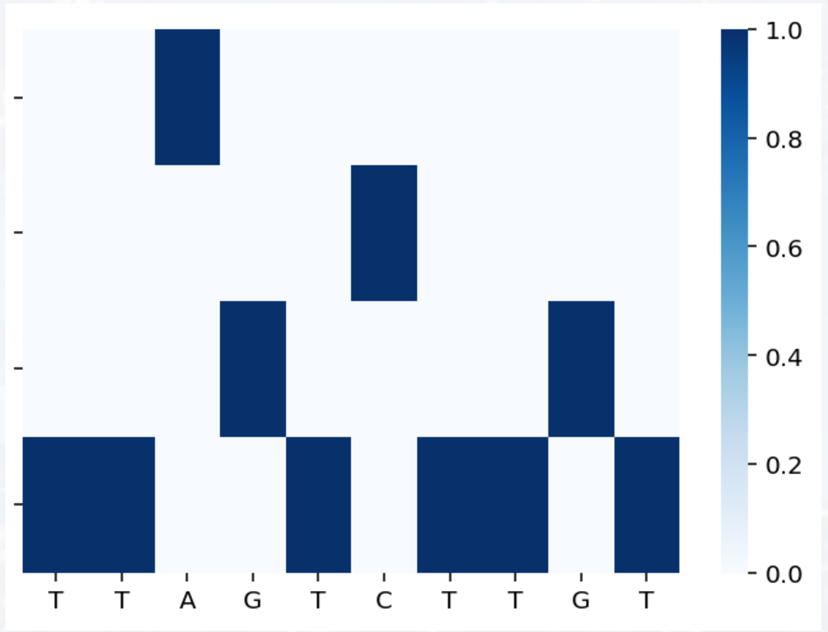
We can create as many **instance** of a `class` as we want!

```
En1 = EncodeMySeq(S1)
En2 = EncodeMySeq(S2)
En3 = EncodeMySeq(S3)

S = [S1, S2, S3]

[EncodeMySeq(s).PlotMySeq() for s in S]
```

Makes the code scalable! → parallelization!



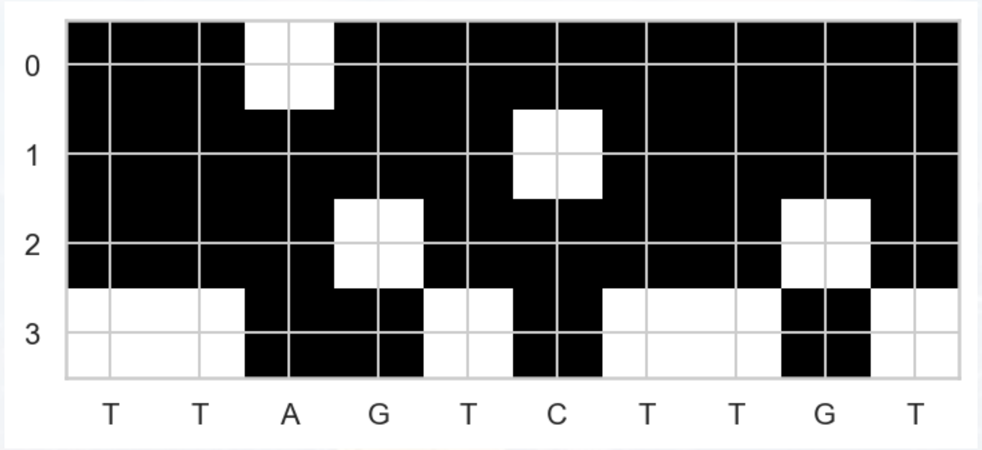




Scripts, Modules and Packages

Our package is now more maintainable!

Example: adding a **new plot routine**



1) adding the plot routine to the header

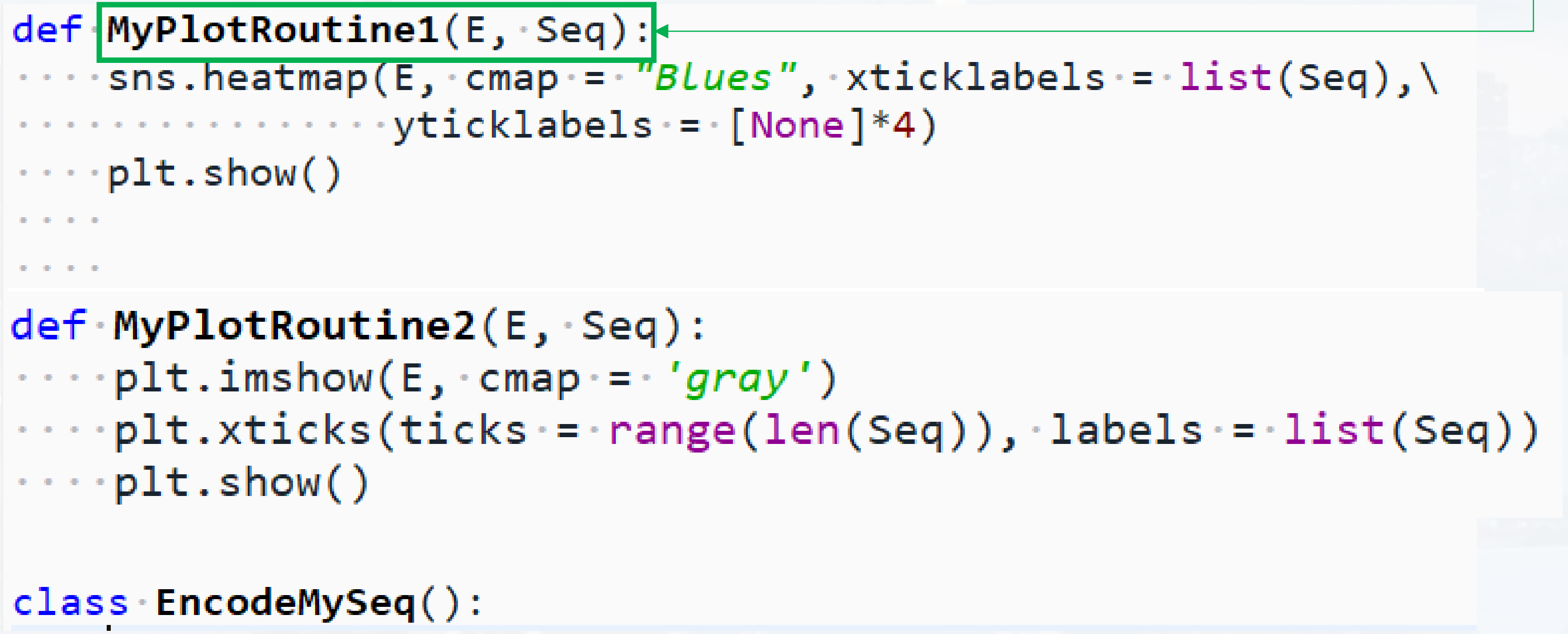
```
def MyPlotRoutine2(E, Seq):
    plt.imshow(E, cmap = 'gray')
    plt.xticks(ticks = range(len(Seq)), labels = list(Seq))
    plt.show()
```

Scripts, Modules and Packages

Our package is now more maintainable!

Example: adding a **new plot routine**

renaming the old plotting routine from **`MyPlotRoutine`** to **`MyPlotRoutine1`**

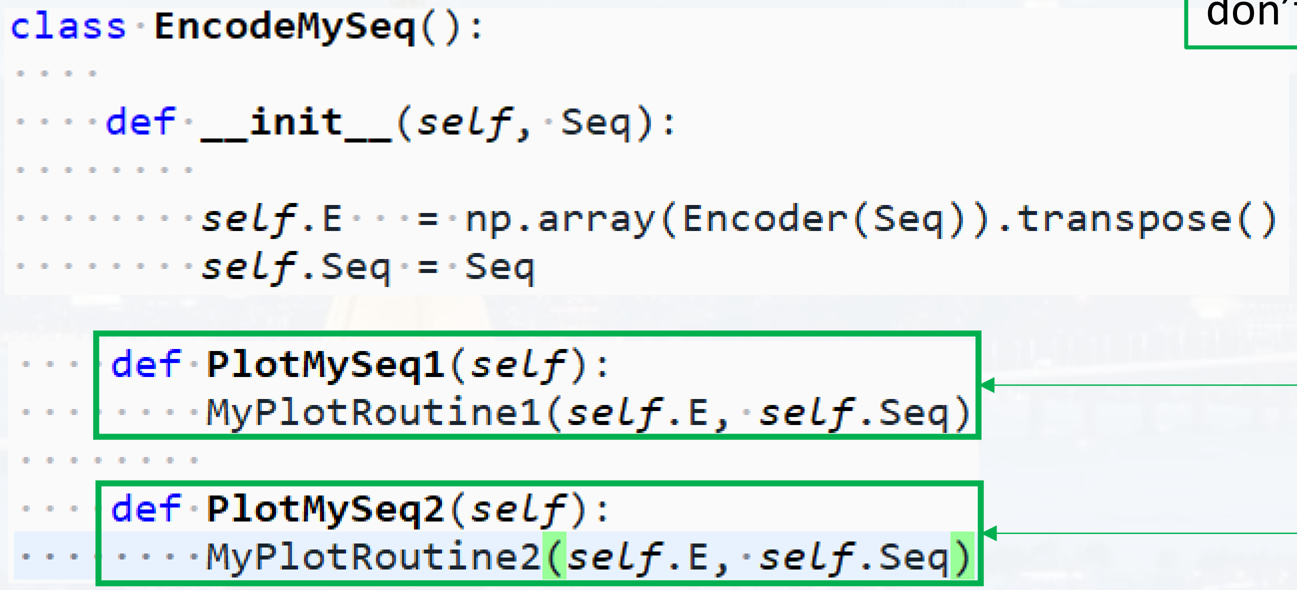


```
def MyPlotRoutine1(E, Seq):
    sns.heatmap(E, cmap = "Blues", xticklabels = list(Seq),\
                ytickslabels = [None]*4)
    plt.show()
    
    
def MyPlotRoutine2(E, Seq):
    plt.imshow(E, cmap = 'gray')
    plt.xticks(ticks = range(len(Seq)), labels = list(Seq))
    plt.show()


class EncodeMySeq():
```

Scripts, Modules and Packages

Our package is now more maintainable!

Example: adding a **new plot routine**



```
class EncodeMyClass():

    def __init__(self, Seq):

        self.E   = np.array(Encoder(Seq)).transpose()
        self.Seq = Seq

    def PlotMySeq1(self):
        MyPlotRoutine1(self.E, self.Seq)

    def PlotMySeq2(self):
        MyPlotRoutine2(self.E, self.Seq)
```

don’t forget to rename the old plot routine

2) adding the plot routine to the `class`

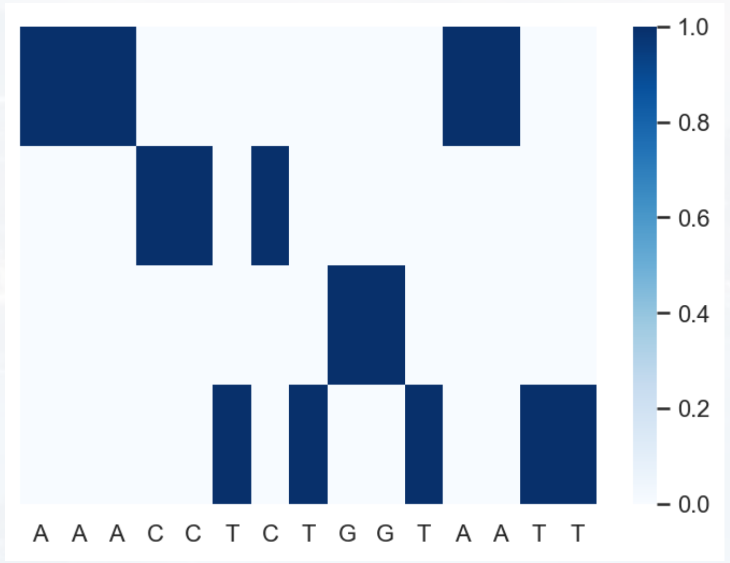
Scripts, Modules and Packages

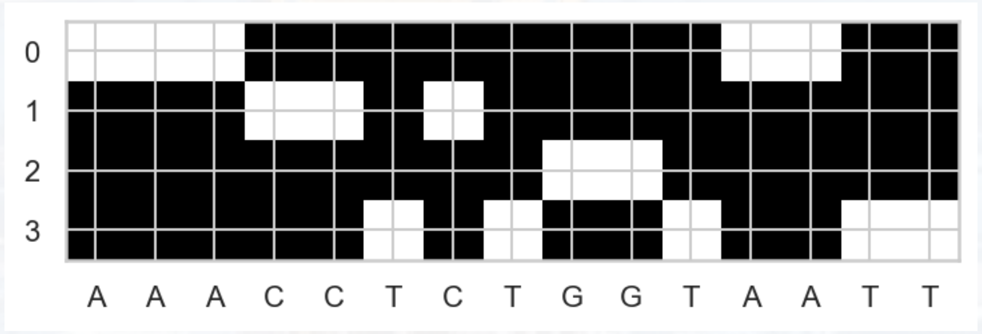
Our package is now more maintainable!

Example: adding a **new plot routine**

We can now choose a plot without reloading the data (saves time)!

```
E = EncodeMySeq(S1)

E.PlotMySeq1()

E.PlotMySeq2()
```





Scripts, Modules and Packages

We can also loop over all sequences **and** plot routines via looping over the attributes using `getattr`

Checking out `getattr` first:

```
E = EncodeMySeq(S1)
```

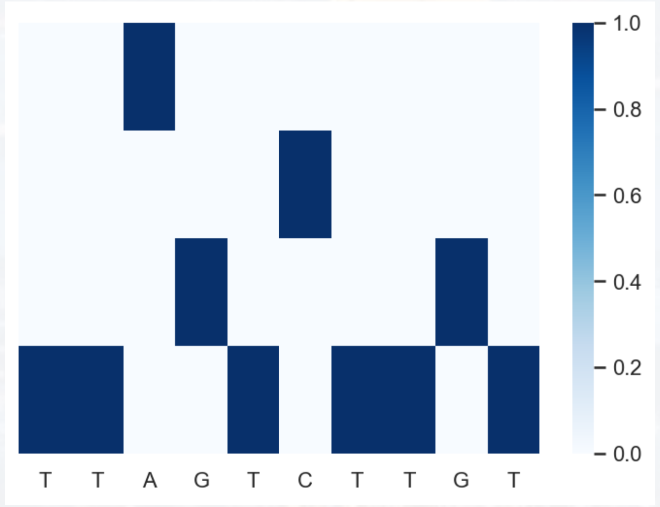
initializing an instance as before

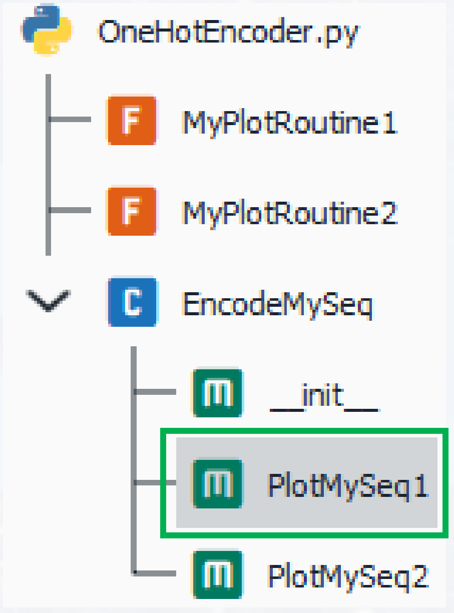
```
M = getattr(E,'PlotMySeq1')
```

retrieving the attribute *'PlotMySeq1'* which is a method

```
M()
```

running the method as before

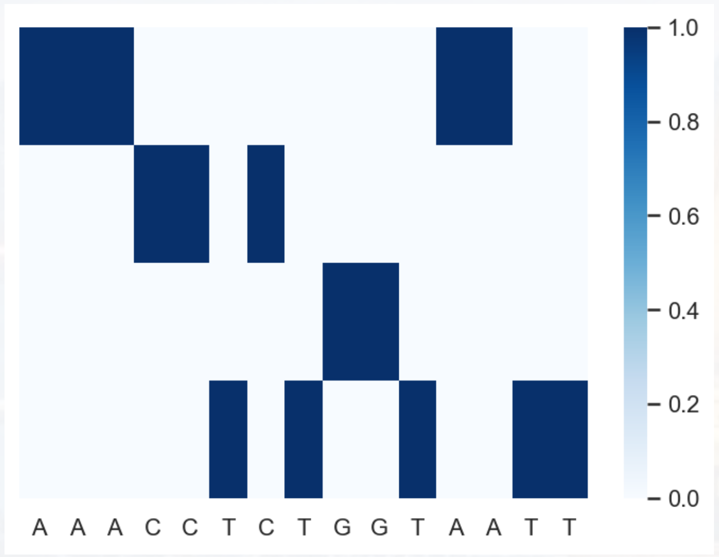


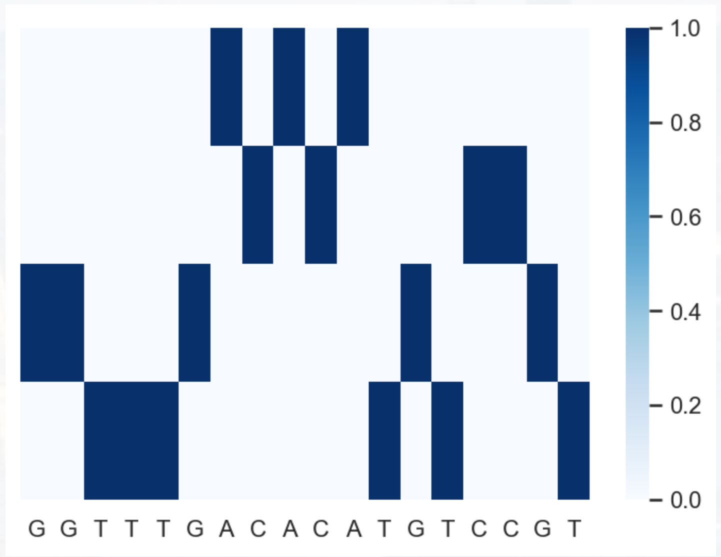


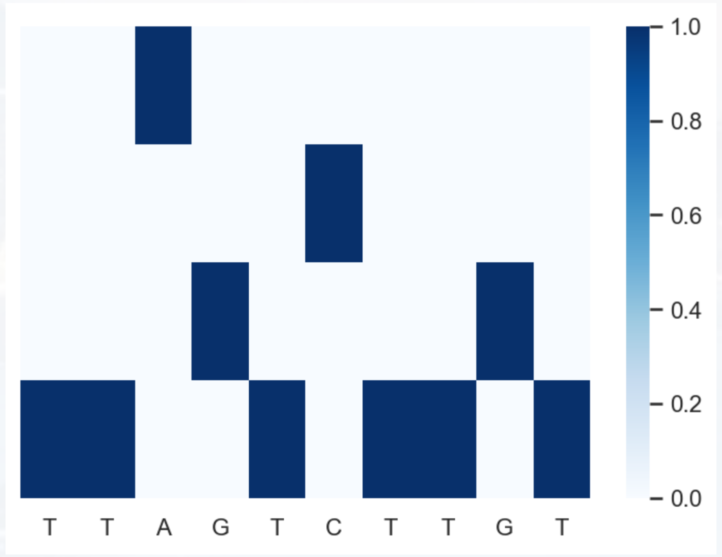
Scripts, Modules and Packages

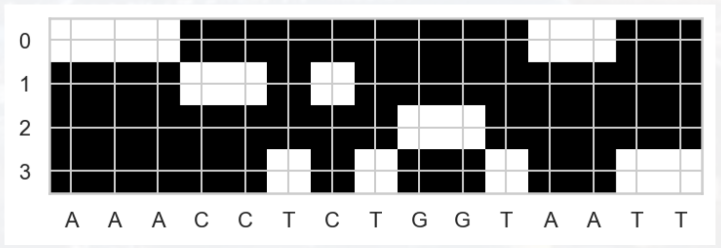
We can also loop over all sequences **and** plot routines via looping over the attributes using `getattr`

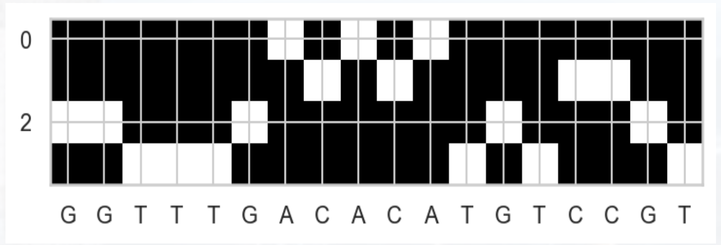
```
L = ['PlotMySeq1', 'PlotMySeq2']

[getattr(EncodeMySeq(s), M)() for s in S for M in L]
```

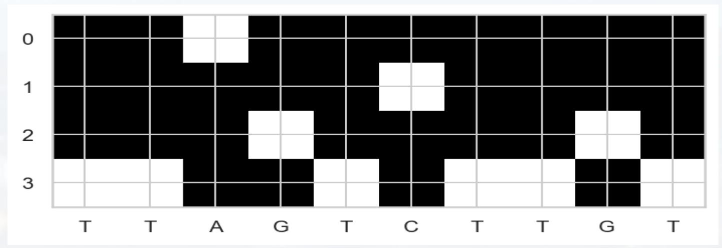


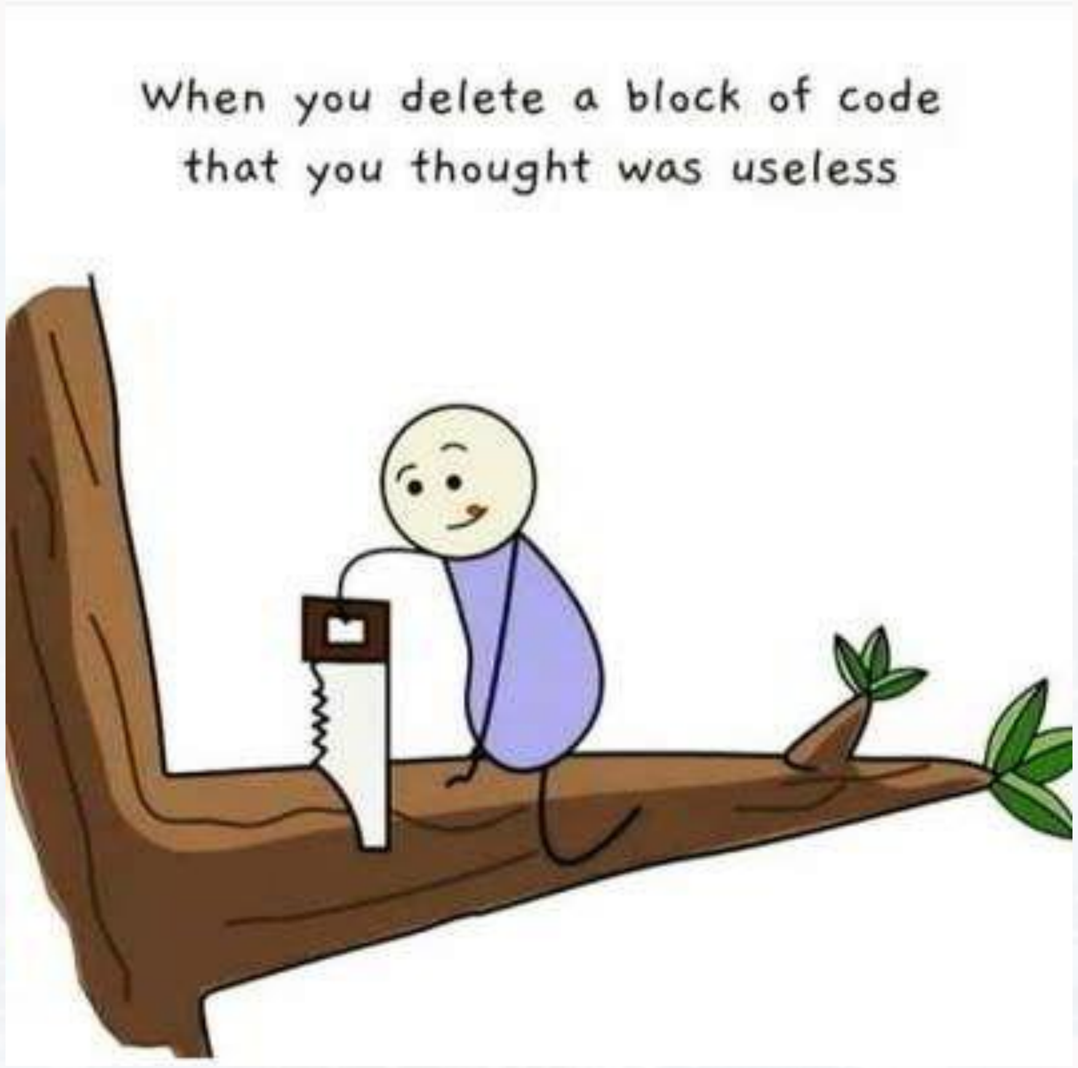











When you delete a block of code that you thought was useless

Outline

- Overall Structure
- Classes
- **Let's build a Package!**

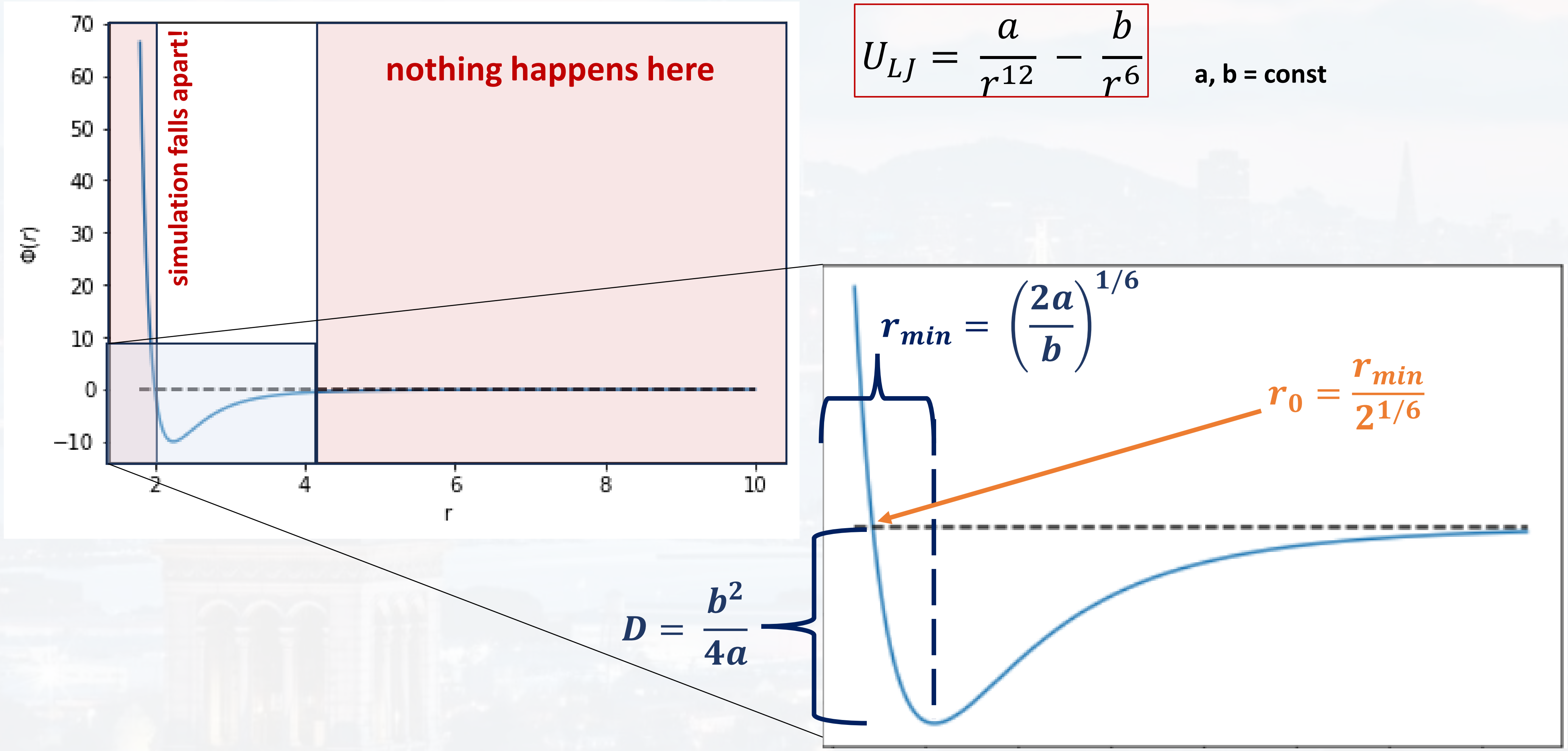
Let's build a Package!

We already built a full package, but now let us repeat the process, but this time working towards the **next project**.

Goal: modelling the **motion of particles** in a **Lennard-Jones** potential U_{LJ}

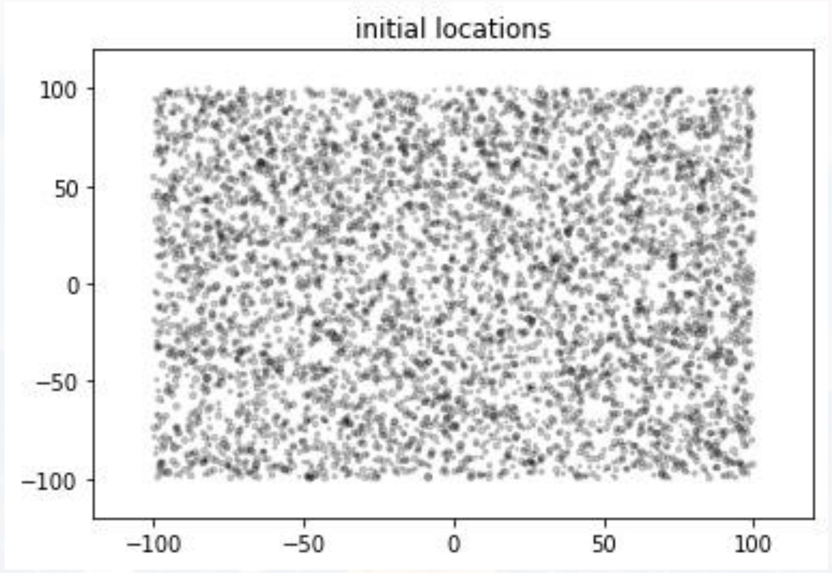
$$\boxed{U_{LJ} = \frac{a}{r^{12}} - \frac{b}{r^6}}$$

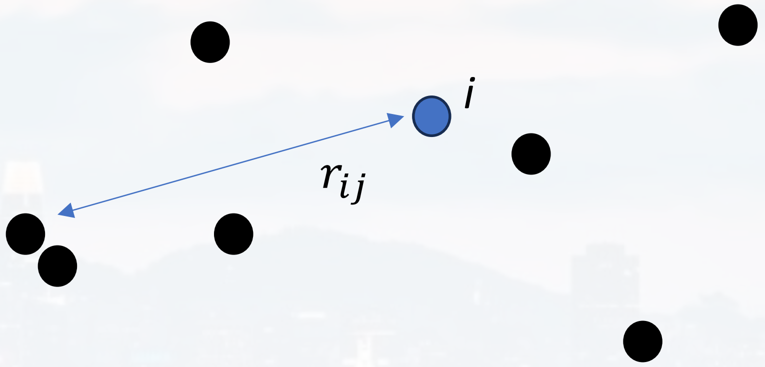
a, b = const



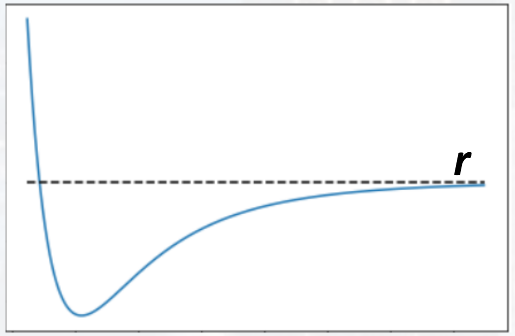
Let's build a Package!

Goal: modelling the motion of particle in a **Lennard-Jones** potential U_{LJ}





Each particle *i* feels the potential $U_{tot}(r_{ij})$ as sum of all the repulsion/attraction to the other particles

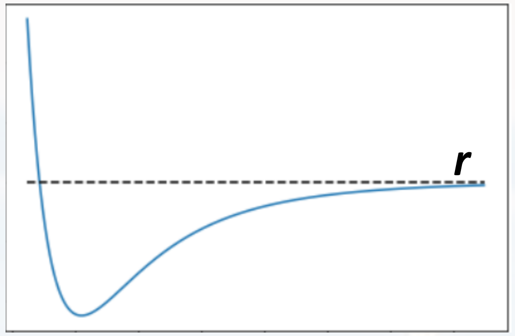


$$U_{tot}(i) = \sum_j \frac{a}{r_{ij}^{12}} - \frac{b}{r_{ij}^{6}}$$

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Scripts, Modules and Packages

Let’s build a Package!



$$U_{tot}(i) = \sum_j \frac{a}{r_{ij}^{12}} - \frac{b}{r_{ij}^{6}}$$

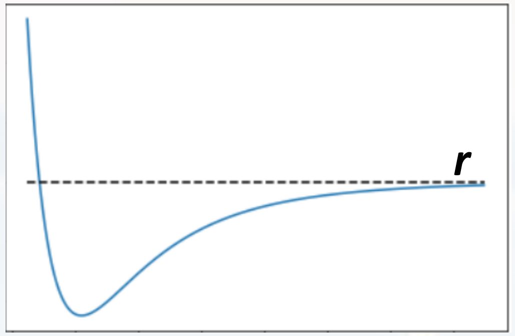
$$r_{ij} = \underbrace{\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}}$$

We need to calculate the **distance** between **each** of the particles

L:	number of particles
x:	vector with x coordinates
y:	vector with y coordinates

```
X       = np.tile(x, (L, 1))
Y       = np.tile(y, (L, 1))

Dx      = X - X.transpose()
Dy      = Y - Y.transpose()
```

Let's build a Package!



$$\underbrace{U_{tot}(i) = \sum_j \frac{a}{r_{ij}^{12}} - \frac{b}{r_{ij}^{6}}}$$

from the **distances,** we can calculate the **total potential**

$$\underbrace{r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}}$$

We need to calculate the **distance** between **each** of the particles

```
def Potential(Dx, Dy, a, b):
    r2 = Dx**2 + Dy**2

    return a/(r2**6) - b/(r2**3)
```

This function calculates the potential for all distances, but **does not** sum up to $U_{tot}(i)$ yet.

your task:

1) modify the function so that it calculates Utot

2) avoid $\mathrm{U} = \infty$ for $r = 0$

Scripts, Modules and Packages

Let's build a Package!

```
def Potential(Dx, Dy, a, b):
    r2 = Dx**2 + Dy**2

    return a/(r2**6) - b/(r2**3)
```

This function calculates the potential for all distances, but **does not** sum up to $U_{tot}(i)$ yet.

your task:

1) modify the function so that it calculates Utot

2) avoid $\mathrm{U} = \infty$ for $r = 0$

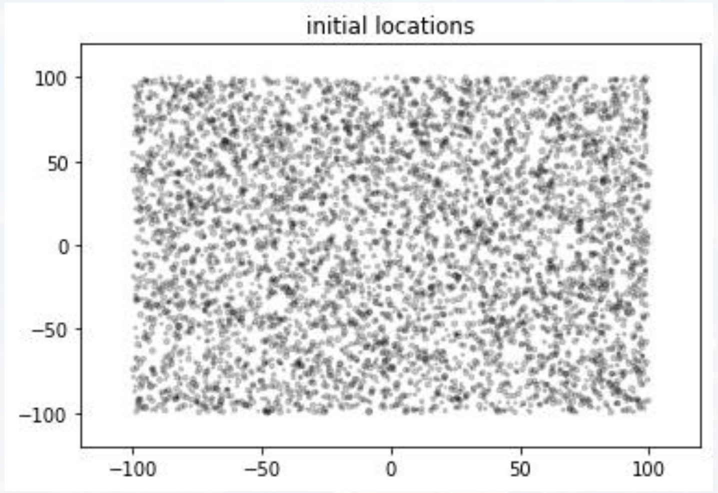
hint: **no loop is necessary** for any of these steps (use algebra with `np.eye`, `np.ones` and `np.dot`),
but you can run a loop if the matrix multiplications are too tricky!

Scripts, Modules and Packages

Let's build a Package!

Goal: modelling the motion of particle in a **Lennrd-Jones** potential U_{LJ}

We also want to plot the locations of the particles for each time step → add a plotting routine!



```
def PlotLocations(x, y, size, Lim, title, marker = 'o', color = 'black'):

    plt.scatter(x, y, size, marker = marker, color = color, edgecolor = None,\
                alpha = 0.2)
    plt.xlim([-Lim*1.2, Lim*1.2])
    plt.ylim([-Lim*1.2, Lim*1.2])
    plt.title(title)
    plt.show()
```

Scripts, Modules and Packages

How do the particles move?

Metropolis:

1) suggest a random move Δx and Δy for all particles

2) calculate $\Delta U_{tot}(x, y)_{LJ}$ based on Δx and Δy for **each particle**

3) move or not:

   a) move those particles where $\Delta U_{tot}(x, y)_{LJ} < 0$

   b) for those particles where $\Delta U_{tot}(x, y)_{LJ} > 0$
   - draw a **random number $\boldsymbol{\rho}$** from a **uniform distribution** in the interval $(\mathbf{0}, \mathbf{1})$
   - move those particles for which $\rho < \exp\left[-\frac{\Delta U_{tot}(x,y)_{LJ}}{T}\right]$

4) repeat

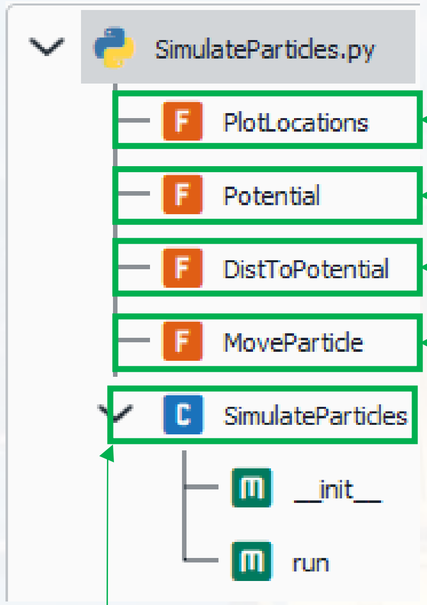
Note: 1) T is the temperature. Try T = 1 first. Vary T later, once the code works
2) Don't worry to much about this algorithm. You will learn **more about it in Chem 273**

Scripts, Modules and Packages

Let's build a Package!

Goal: modelling the motion of particle in a **Lennard-Jones** potential U_{LJ}

Your package could have this structure:



routine that plots **current locations** of the particles

Routine that **defines the potential**. Here: Lenard-Jones

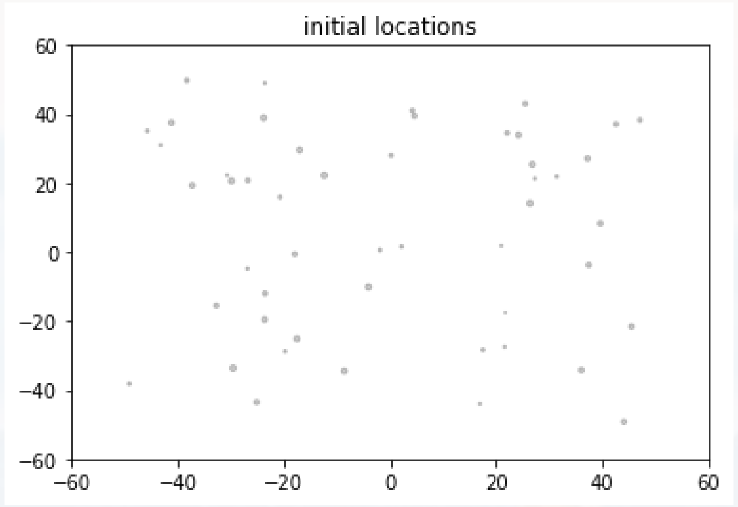
routine that **calculates the distances for each particle and the total potential**, calls method `Potential`

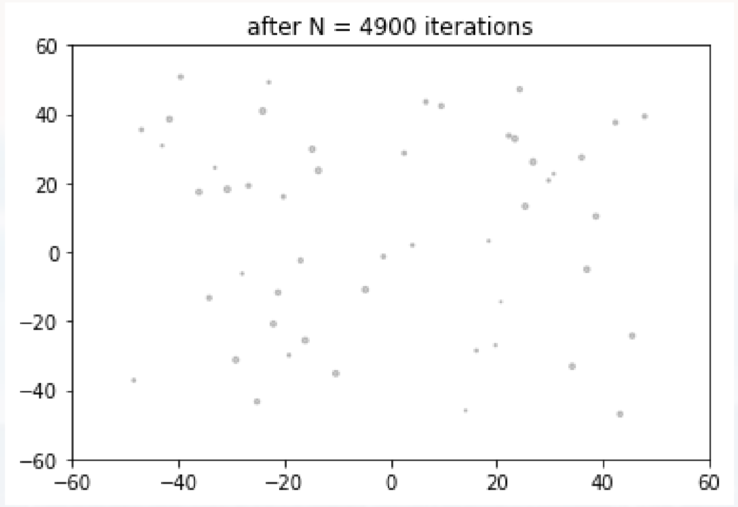
routine that calculates **if a particle is allowed to move** based on $\rho < \exp\left[-\frac{\Delta U_{tot}(x,y)_{LJ}}{T}\right]$

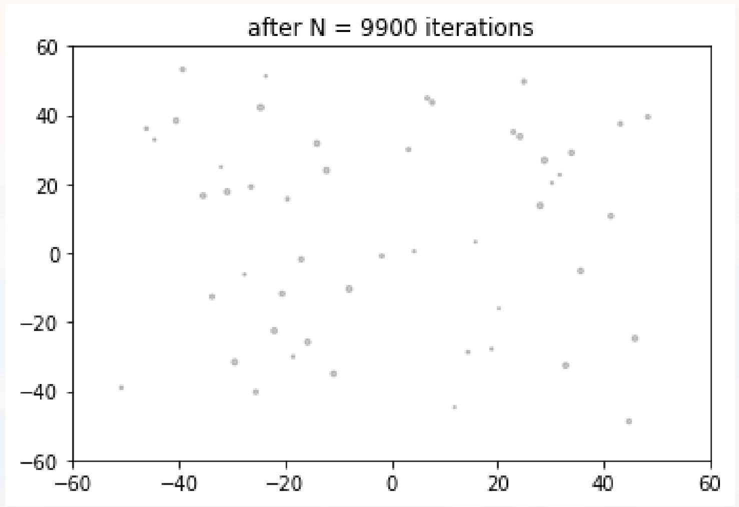
`class` that runs the simulation:
`__init__` takes **number of particles** and **generates initial vectors x and y** with random values
`run` **runs** the actual **simulation** by calling the routines from above. It takes number of **iterations** (say 1000), values for **a** and **b** and **T** (defaults are 1).

Scripts, Modules and Packages

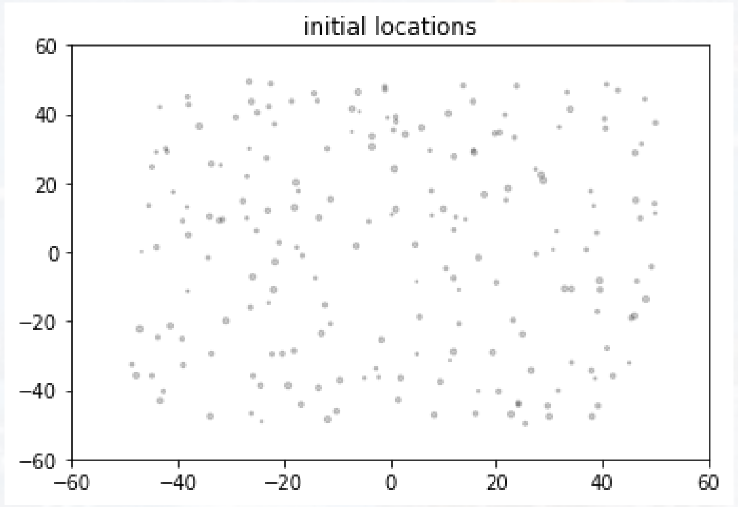
N = 50

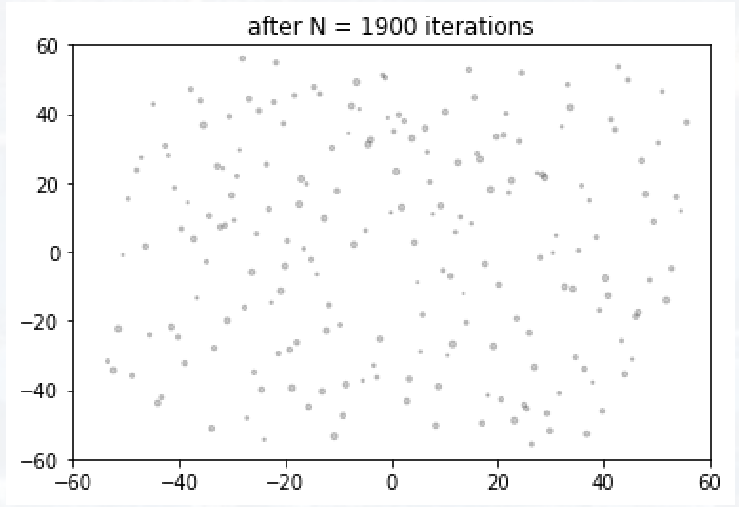


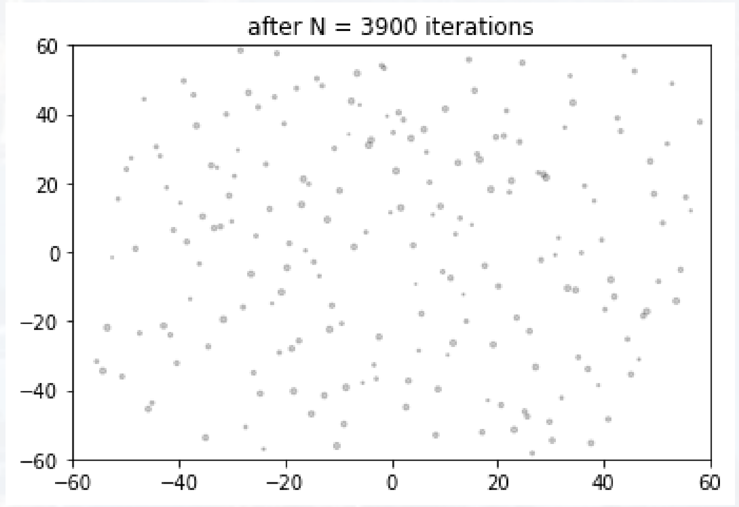




N = 200

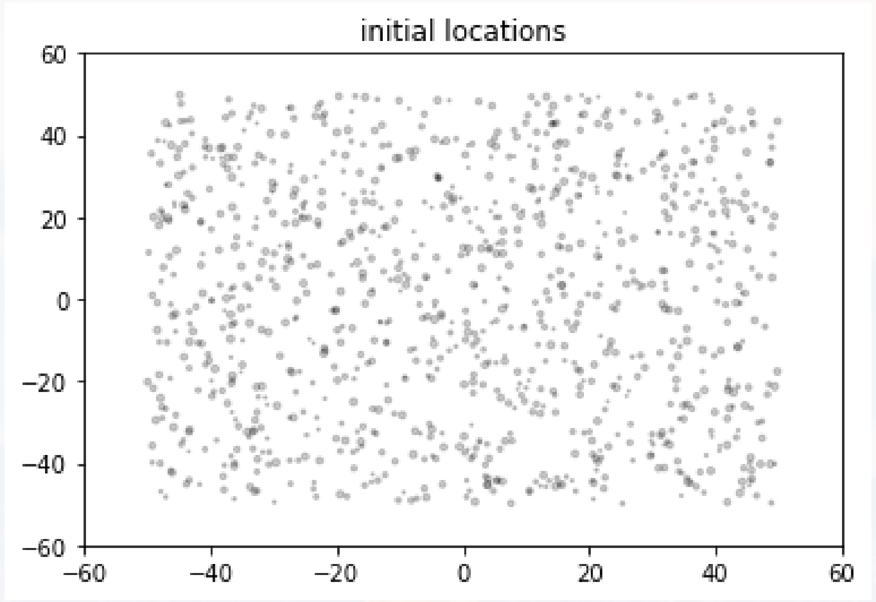




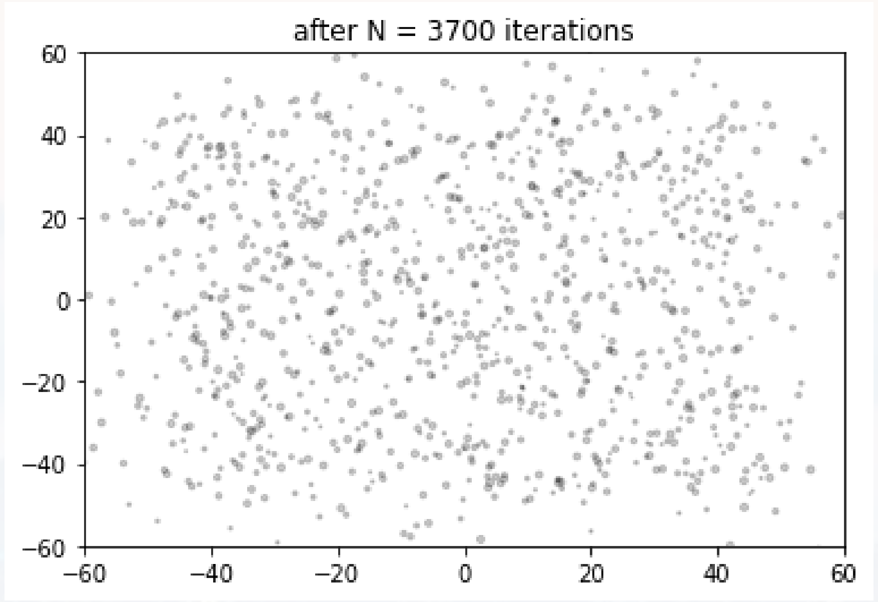


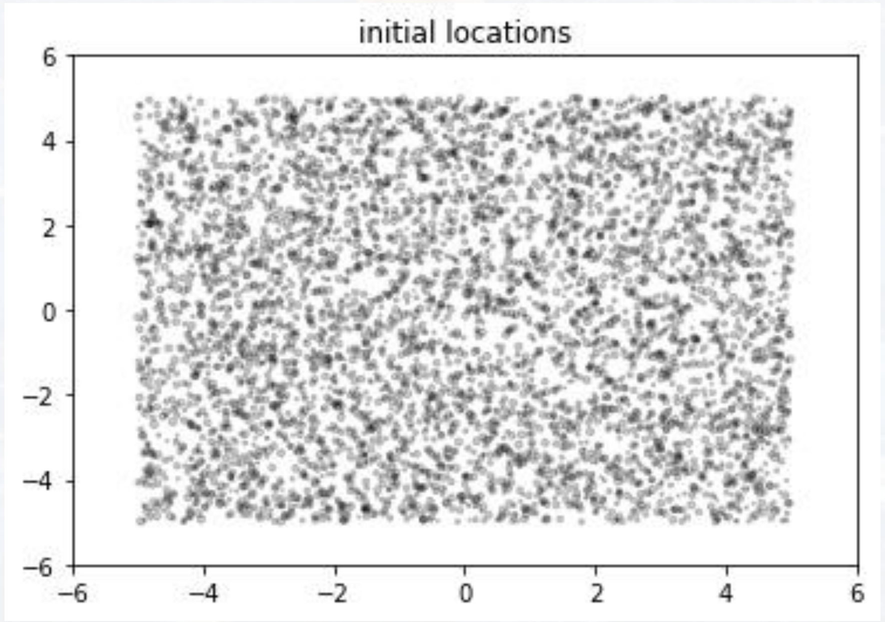
Scripts, Modules and Packages

Let’s build a Package!

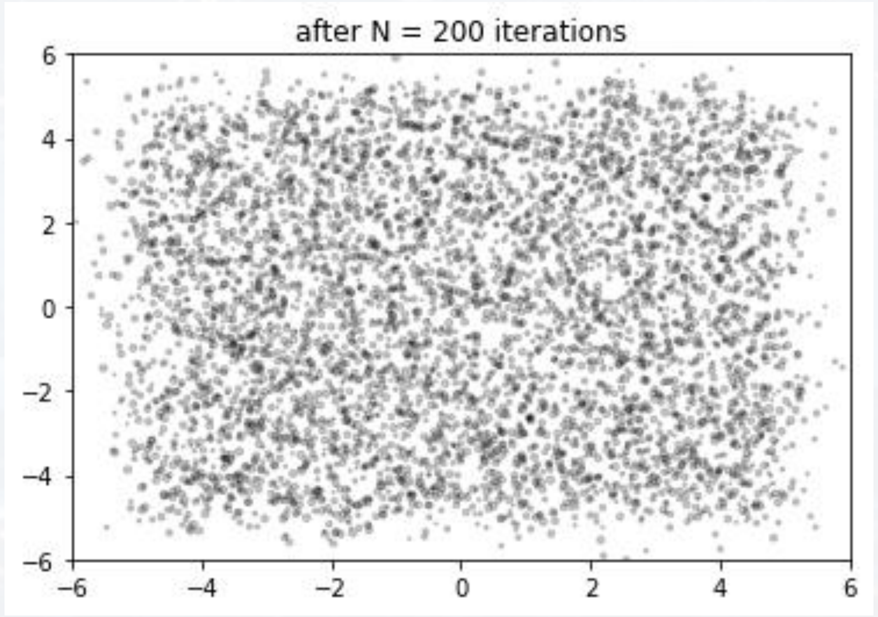


N = 1000





N = 5000



Thank you very much for your attention!