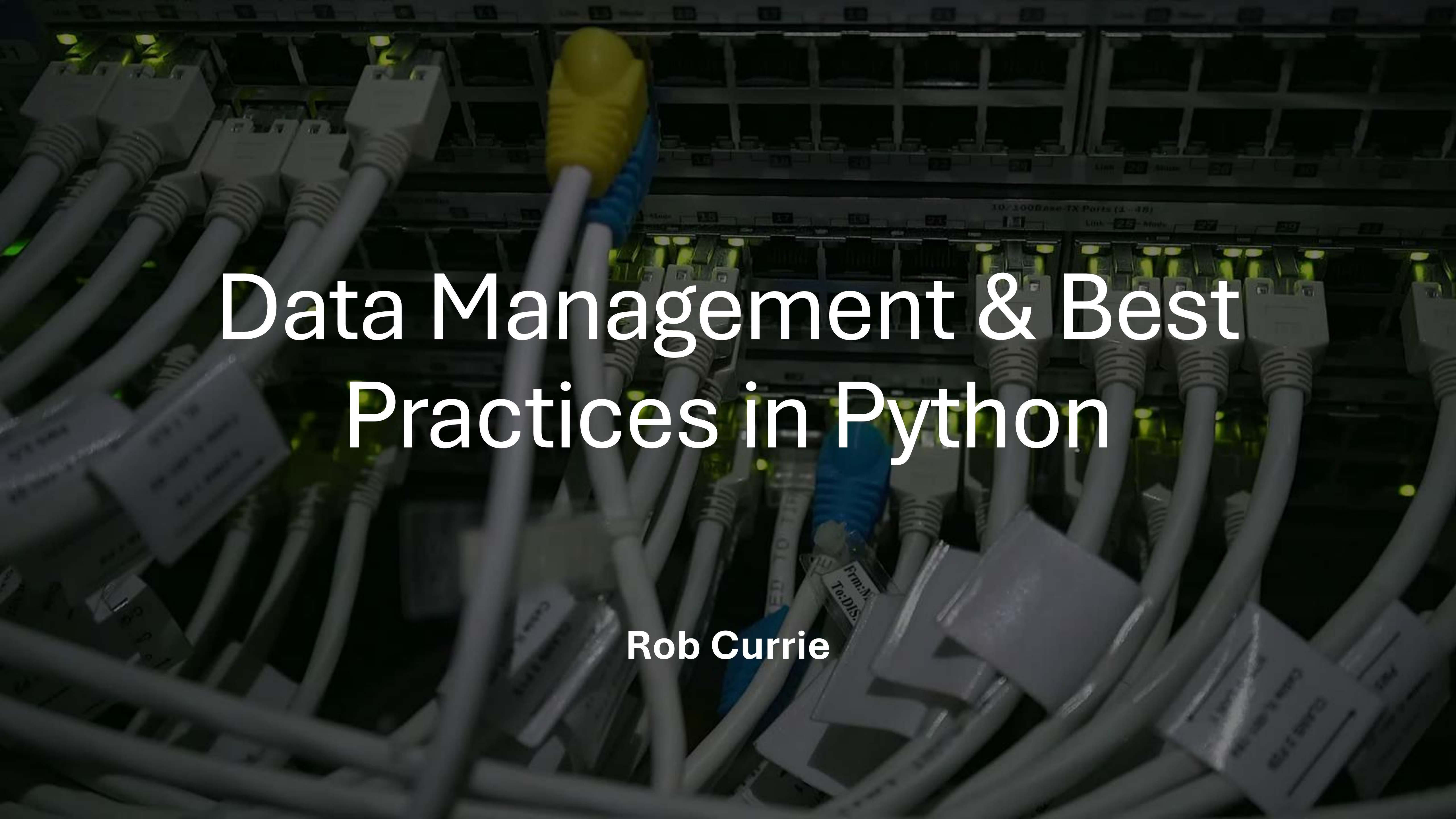


Data Management & Best Practices in Python

Rob Currie

Contents

Data Management 101

- Data is ultimately stored in binary format in computers
- Different storage media have different behaviours/performance
- What we see on the screen usually goes through multiple layers of transformations

ADDRESS	HEX	BINARY	CHAR
0x1000	68	01101000	'h'
0x1001	65	01100101	'e'
0x1002	6C	01101100	'l'
0x1003	6C	01101100	'l'
0x1004	6F	01101111	'o'
0x1005	00	00000000	'\0'

Data Management 101 – Storage Media

There are multiple types of storage media

Speed	Latency	Media	Good for	Cost
↓↓↓↓↓	↓↓↓↓↓ (minutes)	Tape	Archival storage	$
↓↓↓	↓↓↓↓ (seconds)	Disks	Long-term storage	$$
-	↓↓↓ (ms to s)	Flash media	Cameras, certain linear read/write	$
↑	↓↓ (<ms)	SSD	Laptop mobile storage, OS installs	$$
↑↑↑	↓ (<µs)	(G)DDR RAM	Data needed next in a computation	$$$$
↑↑↑↑↑	(ns)	CPU/GPU cache(s)	Extremely short-lived temporary objects	$$$$$$

Almost all of these have different levels of caching mechanisms to enhance their performance
or hide their specific performance features, keeping track of data effectively can be a full-time job(!)

Data Management 101 – Why does this matter?

- ML is a data driven field. It’s all about huge statistical model training and inference.
- If you want to perform a calculation on data, you need to first access it.

- Accessing lots of data requires a lot of time/effort/energy.
- Doing this as efficiently as possible speeds up processing.

- Big linear reads/writes allows buffers and machinery to accelerate everything as much as possible.

- Randomly accessing data out of order puts stress on the system and causes your CPU/GPU/… to have to wait before it can continue its processing.

How do I access data in raw Python

- Accessing data from big complex data structures is obviously the opposite of storing the data in some way.

- Accessing data in big linear reads is the ideal use-case.

- **Randomly** summing a list of 5M numbers takes ~**2s**

  vs

- **Linearly** processing the same 5M numbers takes ~**0.2s**

- This is an order of magnitude difference(!)

How do I store data in Python

- Raw Python has access to basic object types:

  bool, int, float, string

- These objects can be stored in containers to make managing them easier.

- The simplest example is that a string in python behaves like a list of char objects.

string	=	chars				
hello		h	e	l	l	o

How do I store/manage in Python

- The simplest containers for managing data in Python are **lists** and **dictionaries**.
- **Lists** store information in a big flat space in memory.
- **Dictionaries** store information by reference.
- **Sets** exist, and I'll discuss them briefly.
- **Really complex data** is sometimes stored/managed by **classes.**

How do I manage data in raw Python – Saving

- Storing data to RAM, or disk or paper sends the data through many levels of caching.

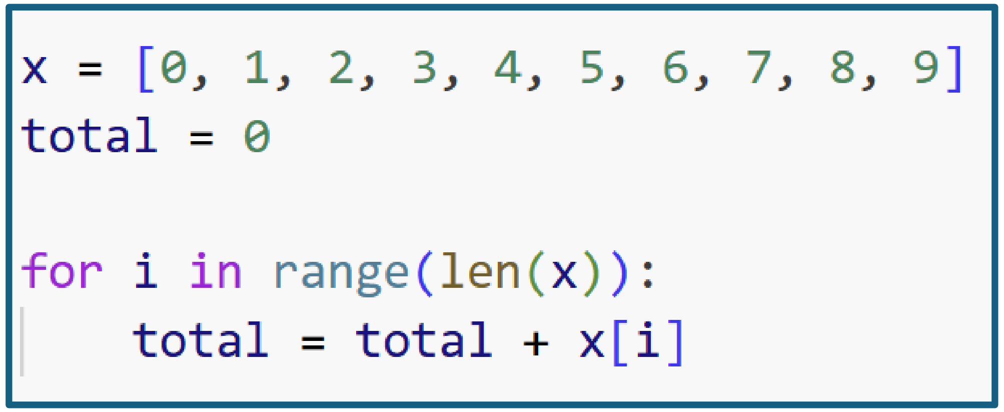
- When you want to make sure data is stored somewhere make sure to flush and close a file.

- I won’t talk about file-management too much here.

  This course is taught in Jupyter notebooks, as a result I’ll be trying to avoid too much direct discussion of file management.

How do I manage simple data in raw Python

- The simplest code to iterate through a list is via index
- e.g.

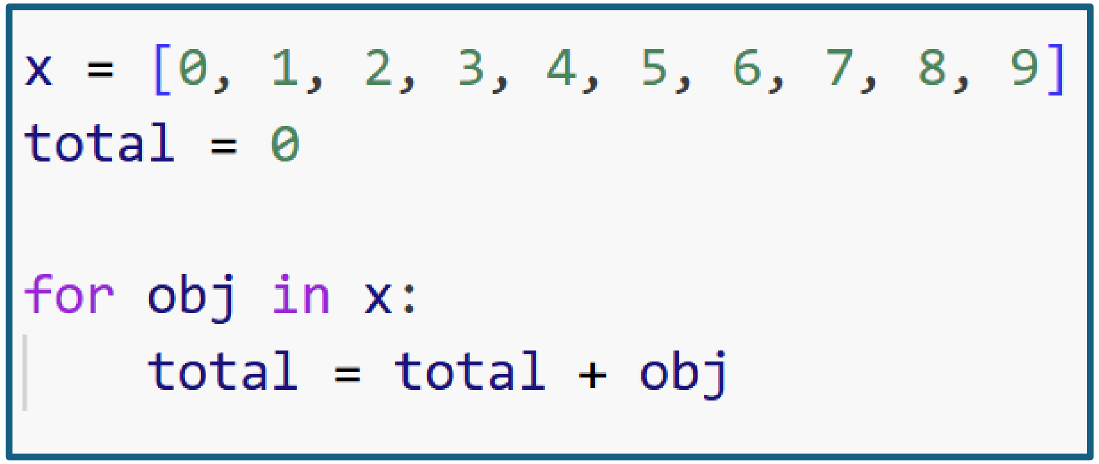


```python
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
total = 0

for i in range(len(x)):
    total = total + x[i]
```

- This is the conceptually simplest piece of code to understand when iterating.
- However, it creates a new temporary object i which is accessible after the loop has finished…

How do I manage data in raw Python

- The next to simplest code is the equivalent of “for-each” in most languages.
- e.g.
- This code is arguably the cleanest code which we can write.
- It’s fast and efficient as it accesses objects by reference
- But can this be shortened further?



```python
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
total = 0

for obj in x:
    total = total + obj
```

How do I access data in raw Python

- First, we can make use of an “in-place” **+=** operator for total…

```python
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
total = 0

for obj in x:
    total += obj
```

- Then we can push this to an extreme using **list comprehension,**

```python
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
total = 0
[total := total + _ for _ in x]
```

or even

```python
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
total = sum(x)
```

How do I access data in raw Python – Dictionaries

- Can be thought of a pauper's data-frame:

```
myDict = {“a”: 1,
           2  : [“z”, “y”, “x”],
          “c”: {“a”: 10,
                “b”: 20}
          }
```

- You can access data in complex ways with dictionaries which can be confusing and powerful:

```
> print(myDict[“a”])
1
```

```
> print(myDict[2][2])
“x”
```

```
> print(myDict[“c”][“b”])
20
```

How do I access data in raw Python – Dictionaries

- Dictionaries are key-value stores. This is excellent for sparse data(!)
- The mapping between key and value is achieved by mapping the hash of the key to a value.

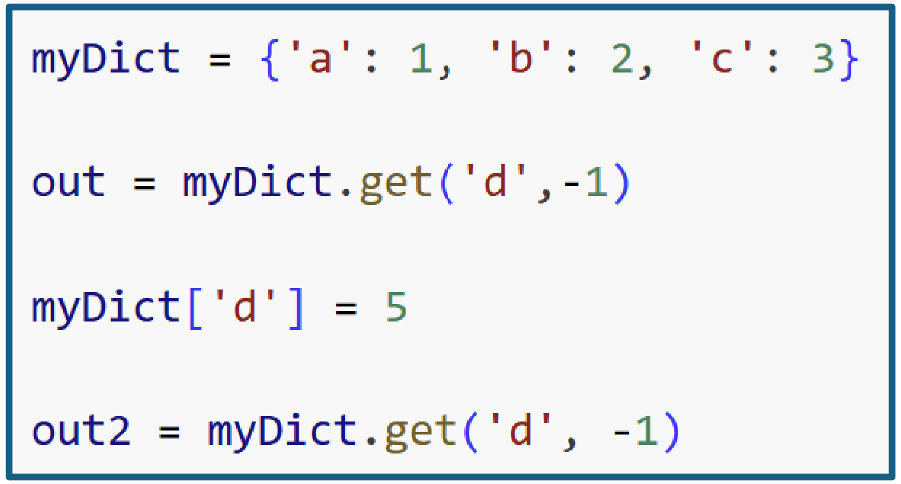
```python
for key in myDict.keys():
    print(f'key: {key}, value: {myDict[key]}')

for key, value in myDict.items():
    print(f'key: {key}, value: {value}')

for key in myDict:
    print(f'key: {key}, value: {myDict[key]}')

for value in myDict.values():
    print(f'value: {value}')
```

- NB: f-strings:

```python
someStr = “world”
print( f"hello {someStr}" )
```

How do I access data in raw Python – Dictionaries

- Dictionaries are very performant when considering sparse data.
- There are even helpful methods to make sure they return a default value when you request an entry that doesn't exist (yet).
- This is a very useful concept in large code-bases.



```
myDict = {'a': 1, 'b': 2, 'c': 3}

out = myDict.get('d',-1)

myDict['d'] = 5

out2 = myDict.get('d', -1)
```

- NB:
  There is a major caveat to dictionaries I've not mentioned yet.

- **The ordering of keys or values in memory is not guaranteed(!)**

  -> The order you get the keys back out is *__technically__* random(!)

How do I access data in raw Python – Sets

- In Python sets are somewhere between lists and dictionaries. They’re used less but are still powerful and worth mentioning.

- A Set has only unique values and is accessed like it’s a list.
  (*NB: dictionaries have unique* ***<u>keys</u>***)

- Compared to large lists sets are often faster for certain tasks as you can perform has-based lookups rather than having to make full computational comparisons.

- However this does mean these containers can only contain hashable objects(!)

How do I manage really complex data?

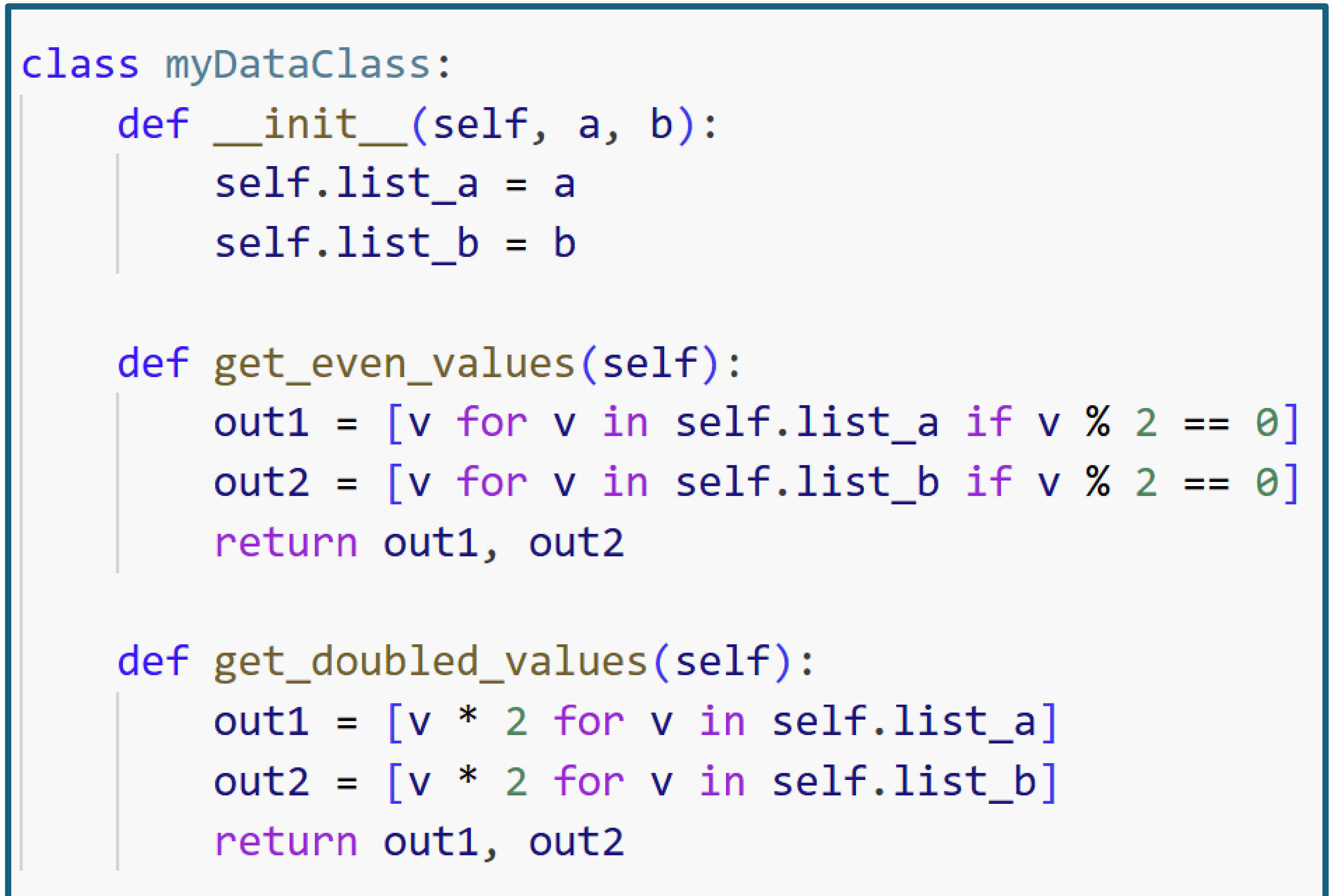
- Really complex data in Python can be managed by classes.

- This allows you to manipulate and store complex data on the fly

- Classes store more than just information.

- For writing good classes and class management I recommend amongst others:

https://docs.python-guide.org/



```python
class myDataClass:
    def __init__(self, a, b):
        self.list_a = a
        self.list_b = b

    def get_even_values(self):
        out1 = [v for v in self.list_a if v % 2 == 0]
        out2 = [v for v in self.list_b if v % 2 == 0]
        return out1, out2

    def get_doubled_values(self):
        out1 = [v * 2 for v in self.list_a]
        out2 = [v * 2 for v in self.list_b]
        return out1, out2
```

Data access in raw Python

- Python is slower at data access and performing numerical operations.
- It’s designed for flexibility and expandability.
- When you start to need to do many calculations a common trick is to move to perform vectorized calculations which are much faster.
- Raw Python → NumPy helps address this

Managing data with NumPy

NumPy

- NumPy is “*the*” package for scientific computing in Python.

  https://numpy.org/doc/

  This package underpins almost everything you’re likely to come across, scikit, PyTorch, pandas, …

- The core of the framework is that data is stored in **np.array** objects and that all of the “heavy lifting” is done using low-level optimized libraries written in C.

- This is probably one of the best examples of:
  “*performance metaprogramming*”

Managing data with NumPy – Types

- NumPy comes with additional types:

  np.float, np.(u)int, np.bool, np.time, np.complex, np.str

- These all come in different precisions from 8bit unit to complex256.

- When working with NumPy these types take the place of common types you encounter in raw Python.

- Technically all data stored in a NumPy container is converted to a type recognised by NumPy.

Managing data with NumPy – Arrays

- Data stored in arrays in NumPy are:

  - more efficient because they use less memory per object
  - have access to optimized functions

- It’s common convention to use

```
import numpy as np
```

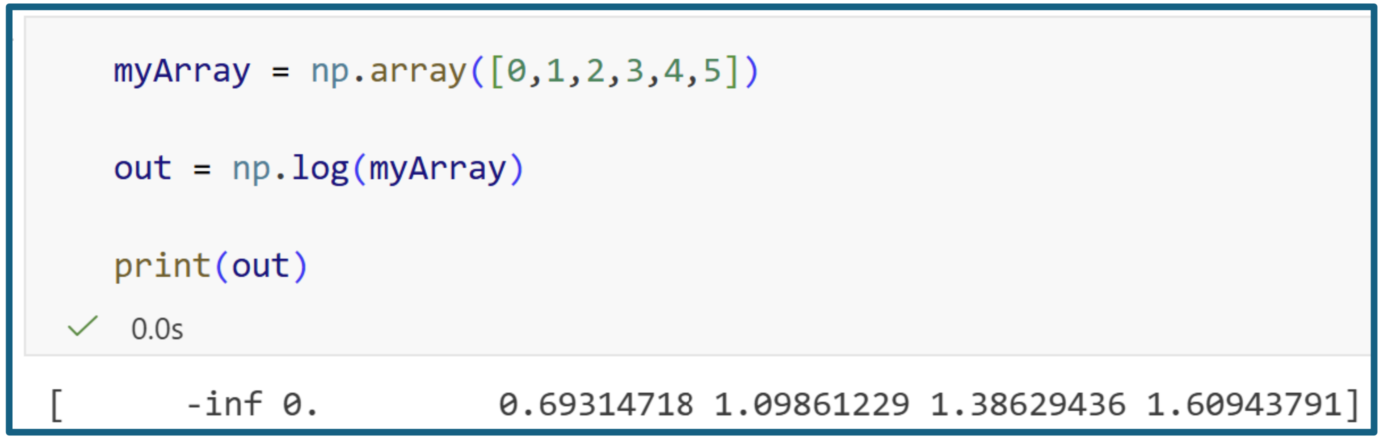
- This makes it possible to access all of the built-in functions from NumPy using **np.xxx**

```
np.array()
```

Managing data with NumPy – Arrays

- As a general rule of thumb, if you're using NumPy objects you want to try and use NumPy methods to get work done.
- Arrays allow you to perform bulk actions in NumPy for short-hand code. Useful to (data) scientists.
- Combining **np.log** and **np.array** compared to the Python built-ins can gain you up to a whole order of magnitude in performance.

e.g.



```
myArray = np.array([0,1,2,3,4,5])

out = np.log(myArray)

print(out)
```

✓ 0.0s

```
[       -inf 0.         0.69314718 1.09861229 1.38629436 1.60943791]
```

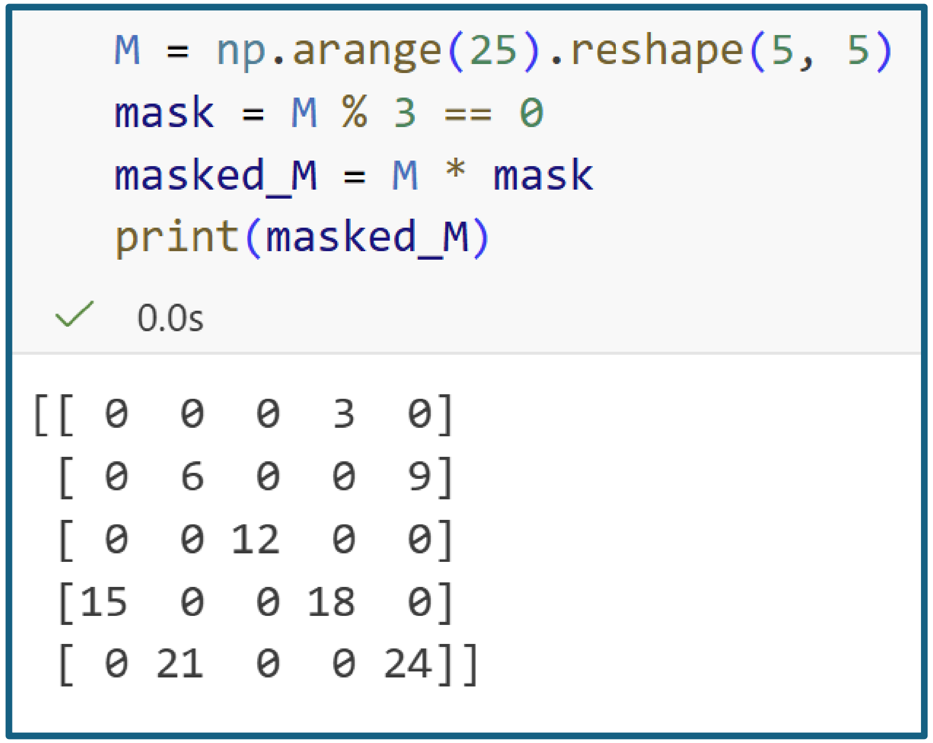
Managing data with NumPy – Masks

- One of the most common ways to manipulate a NumPy array is via masks.

- This is a common concept in building data processing pipelines.

 1) Build a mask to describe what entries to want to keep or throw away.
 2) Build another mask for another description.
 …
 n) combine masks so that we're only left with what we want to keep

- Finally create a new output based on the masks.



```
M = np.arange(25).reshape(5, 5)
mask = M % 3 == 0
masked_M = M * mask
print(masked_M)
```

✓ 0.0s

```
[[ 0  0  0  3  0]
 [ 0  6  0  0  9]
 [ 0  0 12  0  0]
 [15  0  0 18  0]
 [ 0 21  0  0 24]]
```

Managing Data with NumPy – Masks

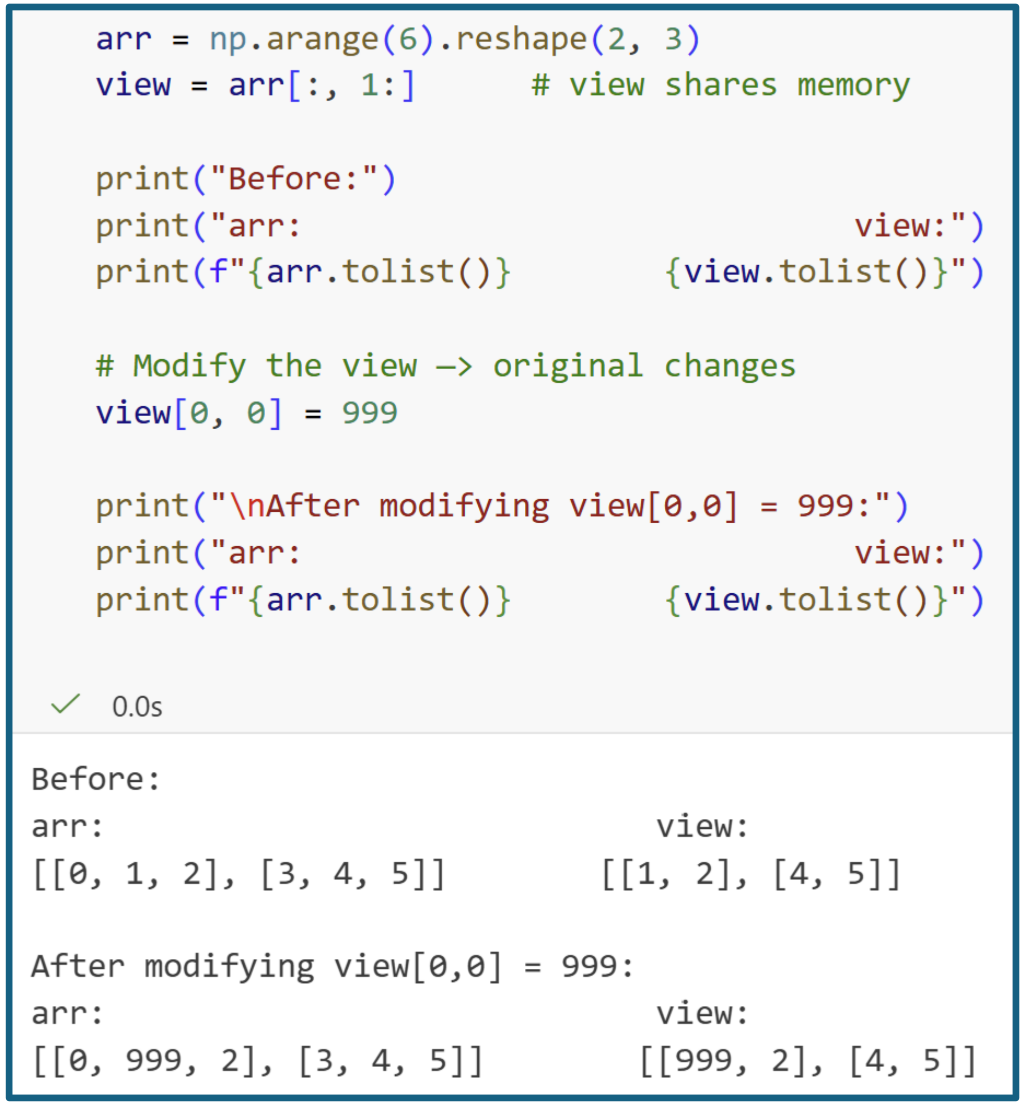
- Using masks in this way has several key advantages:

- Changing or modifying a decision in isolation is easy to understand the impact
- Data is unchanged whilst you're building a list of filters. (also stays in the same place)
- Masks allow us to create temporary "views" of the dataset for working.
- Masks should be Boolean so they can be combined using bitwise logic.

- However, you need to be good at bitwise logic, and familiar with "views" in Python to take advantage of this I claim…

Managing data with NumPy – Views

- Views are extremely powerful.
- They allow you to grab a section of a container without having to copy data, create a full new object or have to use more memory.
- This is one of the key features that makes large data manipulation performant in NumPy.
- The data stays in-place (often in-RAM) for as long as possible until it's needed.



```
arr = np.arange(6).reshape(2, 3)
view = arr[:, 1:]      # view shares memory

print("Before:")
print("arr:                              view:")
print(f"{arr.tolist()}        {view.tolist()}")

# Modify the view -> original changes
view[0, 0] = 999

print("\nAfter modifying view[0,0] = 999:")
print("arr:                              view:")
print(f"{arr.tolist()}        {view.tolist()}")
```

✓ 0.0s

```
Before:
arr:                                view:
[[0, 1, 2], [3, 4, 5]]        [[1, 2], [4, 5]]

After modifying view[0,0] = 999:
arr:                                view:
[[0, 999, 2], [3, 4, 5]]        [[999, 2], [4, 5]]
```

Managing data with NumPy – Views

- Views are very powerful. They’re fast, they’re flexible, but they require you really know what’s going on behind the scenes…

  Their use assumes that you understand memory management asif you’re programming in a hard/compiled language like c/c++ …

- ***As a general rule of thumb.***

  If you suspect you have a container of some-sort that contains important data and you plan to make changes.

  Always take a copy first with **.copy()**

- This isn’t perfect advice and it has draw-backs. But it’s the safest way of writing code and that’s more important to start with.

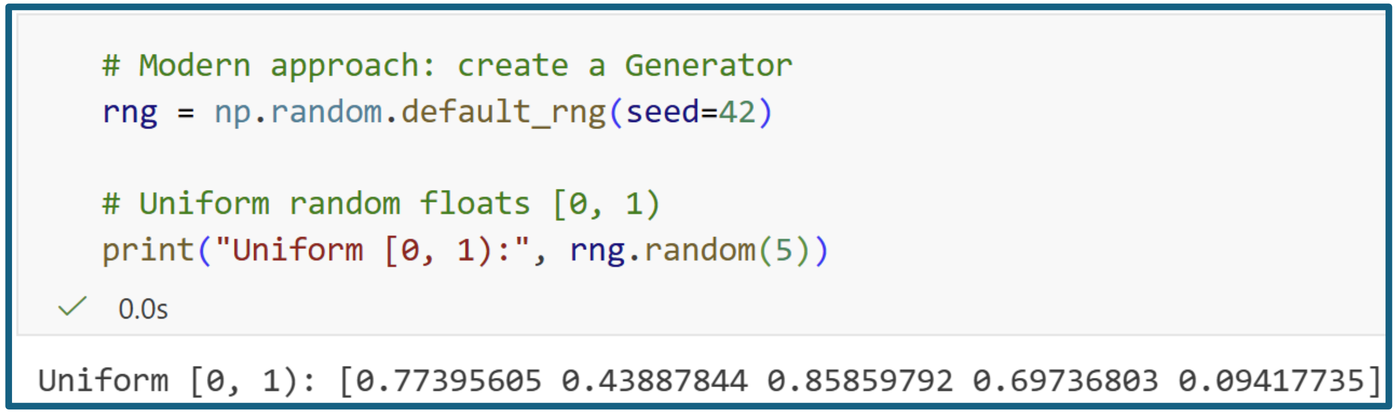
Managing data with NumPy – Other

- There are other features to working with NumPy arrays which are worth mentioning.

- NumPy arrays allow you to ***broadcast*** operations.
- This means you can take actions such as add 10 to every cell in an n-dim array with 1 call to ‘**+**’ without the need to make nested for loops…

- In addition to handling data, NumPy also has a modern and well-designed random number generator and library.

- This can be used to generate new values useful for all sorts of computing.



```
# Modern approach: create a Generator
rng = np.random.default_rng(seed=42)

# Uniform random floats [0, 1)
print("Uniform [0, 1):", rng.random(5))
```

✓ 0.0s

```
Uniform [0, 1): [0.77395605 0.43887844 0.85859792 0.69736803 0.09417735]
```

Managing larger and larger datasets

- As data becomes larger and larger part of the problem becomes the usability aspect of data management
- One of the best ways to manage this is to move to start using a tabular approach to data management.
- This is where Pandas is dramatically better than NumPy

Managing data with Pandas

- The Pandas framework is probably the largest and best-known data management framework in Python

  https://pandas.pydata.org/docs/getting_started/index.html

- The main feature of this framework is the Pandas DataFrame itself.

- This is used almost universally in Data Science and Industry and is probably one of the most compatible/widely-used data manipulation frameworks ever.

- It's compatible with a huge amount of other data container systems such as:

  Excel, XML, HTML, LaTeX, csv, json, sql support, …

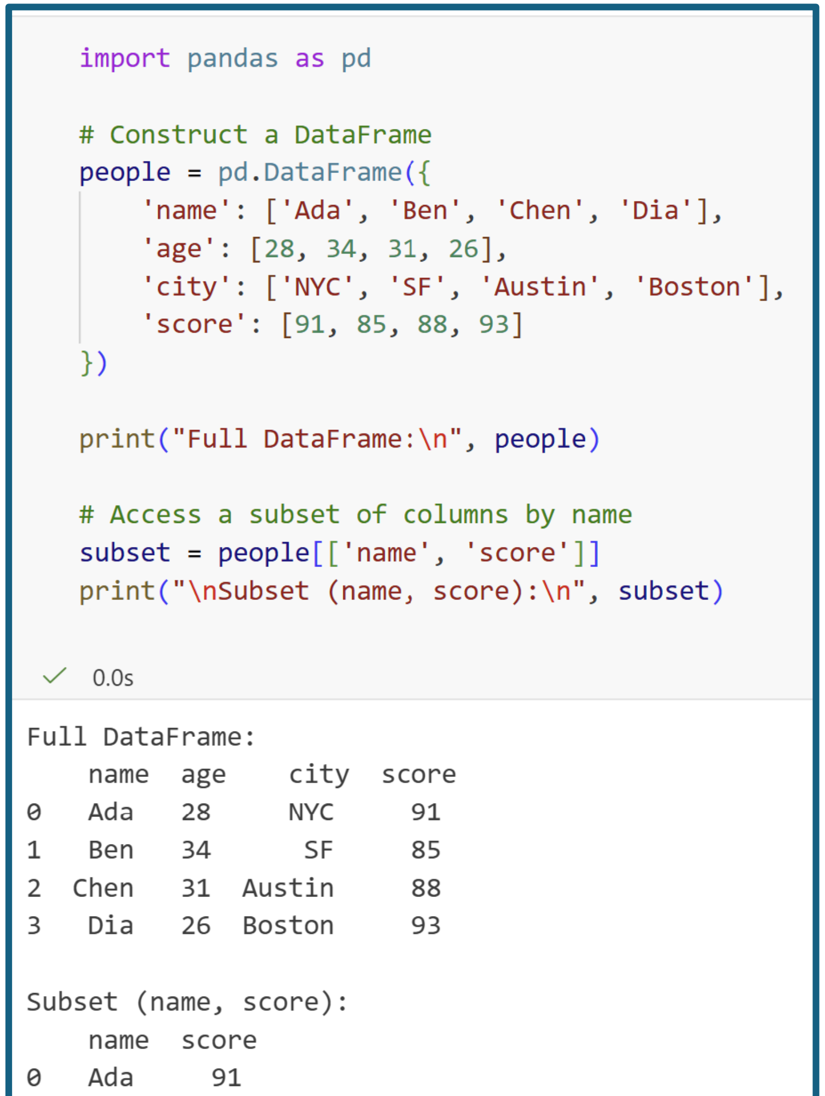
Managing data with Pandas – DataFrames

- Pandas Dataframes are close to “Excel spreadsheets for Python”.
- Data is stored in named columns (keys).
- Pandas is faster at extreme scale (1M+ rows) compared to raw NumPy.
- Accessing data by named columns is more friendly to humans.
- Pandas allows for complex views and slices to be taken.

Managing data with Pandas – Dataframes

- Similar to NumPy, convention is to import Pandas as “**pd**”
- Pandas allows for accessing data from a dataframe using complex calls to the ‘[]’ operator.
- It’s possible to use a lot of different types of calls in pandas that you can’t use elsewhere, such as:

  multiple lists, arrays, short-hand lists,
  mixed lists between columns and rows, …



```python
import pandas as pd

# Construct a DataFrame
people = pd.DataFrame({
    'name': ['Ada', 'Ben', 'Chen', 'Dia'],
    'age': [28, 34, 31, 26],
    'city': ['NYC', 'SF', 'Austin', 'Boston'],
    'score': [91, 85, 88, 93]
})

print("Full DataFrame:\n", people)

# Access a subset of columns by name
subset = people[['name', 'score']]
print("\nSubset (name, score):\n", subset)
```

✓ 0.0s

```
Full DataFrame:
    name  age    city  score
0   Ada   28     NYC     91
1   Ben   34      SF     85
2  Chen   31  Austin     88
3   Dia   26  Boston     93

Subset (name, score):
    name  score
0   Ada     91
```

Managing data with Pandas – Dataframes

- There are a few common patterns/recipes when working with Dataframes is to perform data analysis and filtering input data.
- Dataframes supports masks in the same way as NumPy and builds extra features on top.
- Dataframes support sparsely populated datasets, but by the time data is passed to an ML model the data needs to be in a uniform shape.

Managing data with Pandas – Pipelines

A data management pipeline doesn’t exist as a class/concept in Pandas, but people build them with Pandas quite frequently.

1. First, we load our data.
   (Pandas supports all sorts of data formats and filetypes)

2. Then we calculate masks to apply to our input data.
   (This reduces the amount of data we’re processing)

3. We then identify any rows containing invalid (Nan or inf) values and remove them.
   (This reduces the amount of processing next) (potentially pad missing data here!)

4. Now we pre-process our data in some way. Take logs, multiply, normalize, scale, transform…

5. Finally check the output from above for Nan or inf values and tend to drop the rows containing this.

Managing data with Pandas – Pipelines

- One of the biggest patterns is that should you ever encounter a value you don’t expect (e.g. inf) or a value way out of a range, is to replace it with a Nan (Not a number)

- This is so important and widely used in pandas that this is the main difference between Pandas and NumPy that pandas effectively adds a new data type:

  pd.NA

- Combined with methods such as **.dropna()** the pandas framework makes it possible to:

  - Load data
  - Pre-Filter data
  - Pre-process data
  - Transform data
  - Filter data

- Guaranteeing that when we move on to do an analysis with this data only “good” values are used.

Managing data with Pandas – Pipelines

- If you’re talking about real-world datasets people tend to accept a certain amount of pre-processing and pipeline in-efficiencies when handling an input dataset.

- Common mistakes are things like we call **log(0)** which then returns **inf** which is then filtered out from our dataset.

  Instead, we should have called **log(0+1e-8)** to make sure we keep this row and the rest of the information (it could just be this value in the row which is off)…

- There is no universal approach to building a data analysis pipeline, that is why pandas is so flexible.

Doing Machine Learning with data

- To use the data, we have now processed to place this data somewhere useful and in a format which is useful.
- Pandas ultimately stores all of the data in a usually flat way which is best for access by just CPUs.
- If we want to use our data with ML it needs to be accessible via our accelerator, and it has to be in a batched format which allows us to do the calculations we want.
- To do this we need to put our data in PyTorch.

Managing data with PyTorch

- Data management in PyTorch is designed with a different goal in mind to NumPy or Pandas.

- Feed the data into the ML framework for training an ML model as efficiently as possible.

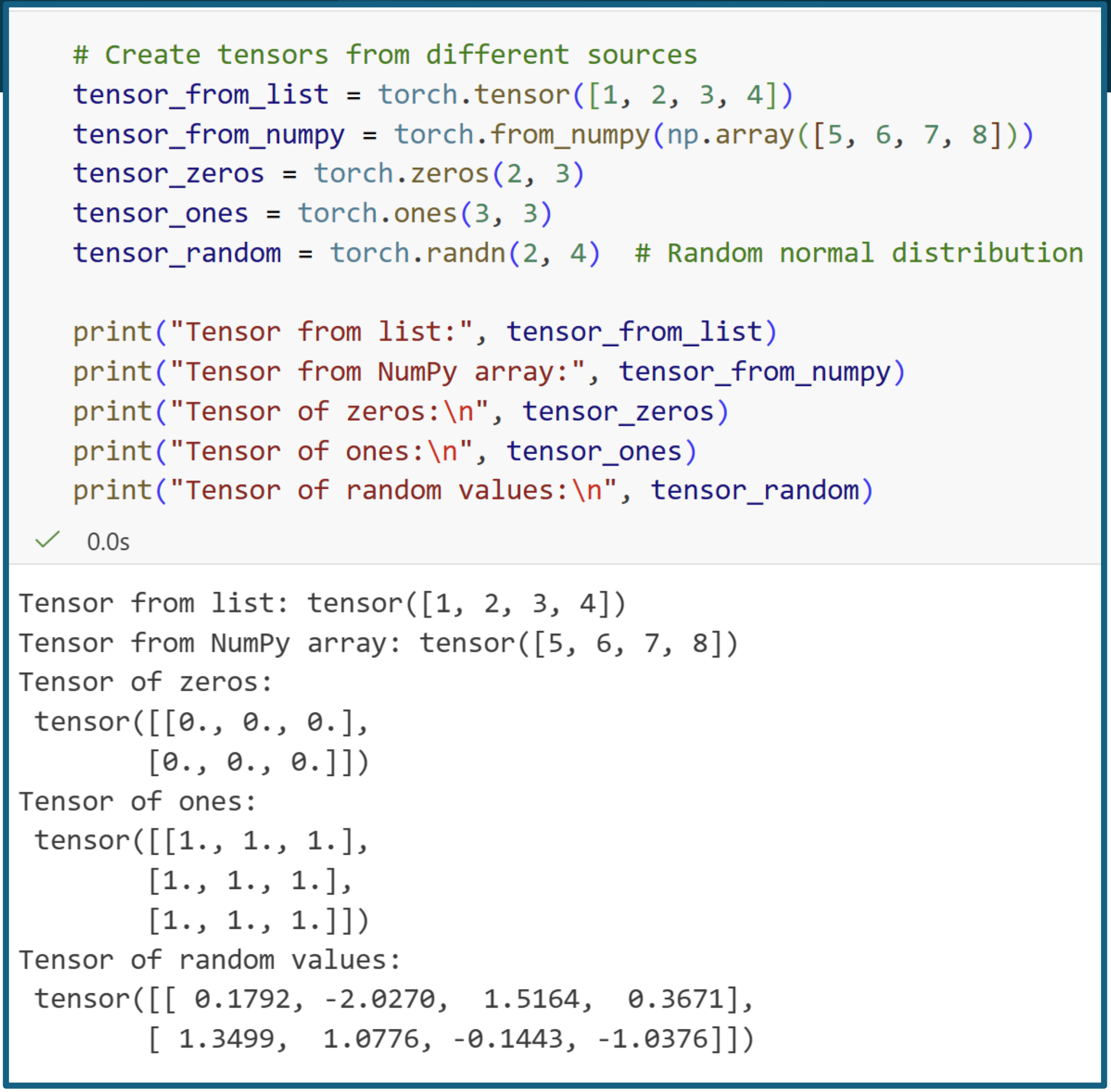
- All data for ML needs to be in a numerical format which is stored as a Tensor object.

- One of the explicit key features with data in PyTorch is that a tensor may be moved between CPU/GPU/NPU/FPGA/… as required to train our model efficiently.

Managing data with PyTorch – Tensors

- Creating a Tensor is relatively easy.
- PyTorch Tensors are still effectively interoperable with NumPy arrays, but its own types.
- PyTorch data types are similar to those in NumPy but with more limited precision.

  Most of ML training is done with 32-bit precision or potentially much less.



```
# Create tensors from different sources
tensor_from_list = torch.tensor([1, 2, 3, 4])
tensor_from_numpy = torch.from_numpy(np.array([5, 6, 7, 8]))
tensor_zeros = torch.zeros(2, 3)
tensor_ones = torch.ones(3, 3)
tensor_random = torch.randn(2, 4)  # Random normal distribution

print("Tensor from list:", tensor_from_list)
print("Tensor from NumPy array:", tensor_from_numpy)
print("Tensor of zeros:\n", tensor_zeros)
print("Tensor of ones:\n", tensor_ones)
print("Tensor of random values:\n", tensor_random)
```

✓ 0.0s

```
Tensor from list: tensor([1, 2, 3, 4])
Tensor from NumPy array: tensor([5, 6, 7, 8])
Tensor of zeros:
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
Tensor of ones:
 tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
Tensor of random values:
 tensor([[ 0.1792, -2.0270,  1.5164,  0.3671],
        [ 1.3499,  1.0776, -0.1443, -1.0376]])
```

Managing data with PyTorch – Dataset

- The Dataset class in PyTorch is a wrapper around a collection of smaller Tensors which represent individual measurements/events which are to be used/generated/compared to an ML model.

- The Dataset class both does a huge amount and looks like it does nothing.

- You can iterate through a given index on a large Tensor (like a Dataset) or can access subsections of a larger Tensor without the need to use a Dataset…

- However,

  Dataset can lazy-load data from disk. Dataset can transform data on access. Dataset can pre-process data on the fly.

Managing data with PyTorch – DataLoader

- When training a ML model 99% of the time it’s impractical to load all of the data through the model at the same time.

  This is true for the small examples in this course and is true for the multi-TB datasets used for training large LLM models.

- When you want to train a model, you need to batch up your dataset during training.
- In a lot of situations, you will also want to randomize the order of data within batches as it’s presented to the model.

- A DataLoader does batching, calling processing and shuffling all for us in an automated way.

Managing data with PyTorch – DataLoader

- The final "form" the data should take for training an ML model is a DataLoader.
- It seems like a lot of effort to go from
  "*I have my data*" to "*I want to use my data with my model*".
- Keeping the loading/processing/generating of the data separate to the training is good for code clarity and design.
- It forces us to worry about what we're doing during training at the end of the pipeline and only how to read/pre-process the data when loading it from disk.

Managing data with PyTorch

- In the simplest terms, it’s useful to think of an individual Tensor as a single datapoint/image/entry/…
- A DataSet can be thought of as being a list of Tensors closer to a big list of the datapoints.
  (Technically the DataSet can transform data on access…)
- A DataLoader is the tool which auto-batches the Dataset for us so we can do training of an ML model, so kinda think of it as a list of (Data subsets) lists.

Code Management in Python

- For the workshops in this course, we will just be using Jupyter notebooks.

- Jupyter notebooks are great for Data science for a few reasons:

  - They allow us to store graphical output alongside the code to generate it

  - Allows us to track what the current state of an analysis at a high-level is

  - Provide a common platform to see the output from our model training

  - Keeps our results comparable with each other to make it easier to learn/understand

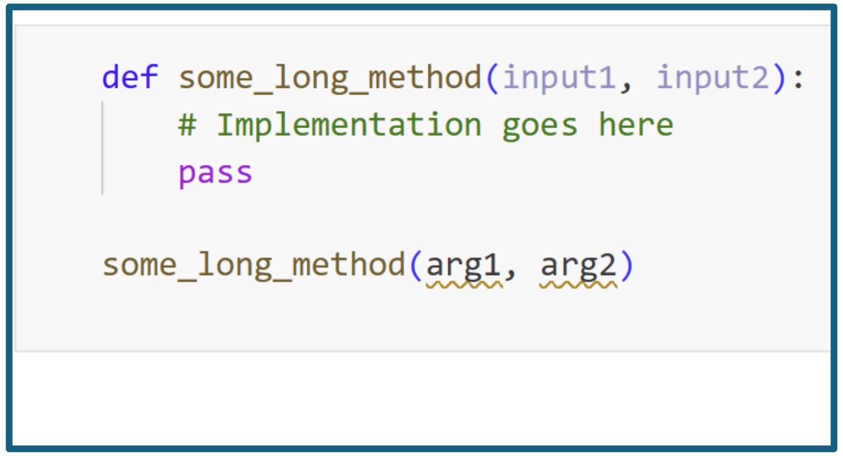
- Unfortunately for a large project who wants to scroll for 5min to get to "the last/latest" output/plot.

Code Management in Python

- A good way of arranging code in a notebook I try to promote is to have each cell be callable and re-producible.

- This is closer to older computing paradigms of “do one thing and do it well”.

- It helps separate (as much as possible), different ideas and concepts.



```
def some_long_method(input1, input2):
    # Implementation goes here
    pass

some_long_method(arg1, arg2)
```

Code Management in Python

- In the “real world” so considering programming in large projects beyond this course Jupyter notebooks are excellent for doing science.

- i.e. I’m a scientist trying to solve some problem or explore some data-analysis or simulation.

- Typically I have lots of possible combinations of inputs/outputs to keep track of with plots and conditions, etc…

- Jupyter notebooks are excellent for this. Once cell per config (with notes) and the output from calling some external tool/code saved alongside the notes and config used to generate it.

Code Management in Python

- By the time you have moved to the point a large jupyter notebook is useful for tracking your results, you're starting to move into proper programming territory.

- At this point you will want to start to think less about having lots and lots of **def**-*ined* methods and will want to start thinking about grouping concepts together in **classes**.

- At the point you need to start factoring your code into classes there are several pieces of good advice to try and remember:

  - One core concept per class as much as possible
    (i.e. a student class and a course class separating learning from student grades)

  - "***Rule of 7***"
    Try as much as possible to keep methods short and simple to keep code legible

  - One class per file
    Once you're branching out to use multiple classes consider breaking them into multiple files.

Code Management in Python – Requirements

- We already encountered this in the last workshop.

- To run our code, we need the code our project relies on installed.

- Keeping track of that can be tricky.

- You can do it by an anaconda env (dificult)… or a requirements.txt (also difficult) …

  basically, if you ever need to install an extra package (for whatever reason) at least write it down and try to keep track of what version you used.

  Your future self will thank you(!)

Code Management in Python – Coding

- Good programming is difficult.

- A good tool to help you is to consider asking a generative AI tool how to re-factor your code.
  (ONLY if you're allowed!)

- There's a huge amount to "good" Python programming which I can't cover in just 1hr.

Further Python programming examples

- Looking beyond Workshop2

- I have put together a higher-level Python tutorial to learn some of the more complex concepts around Python programming such as:

  - exception handling
  - writing new classes
  - importing classes
  - handling multiple files
  - …

  If anyone wants to attempt this, I'm happy to help by email

- Here → https://github.com/rob-c/farm