Introduction to Programming in Python

Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN

Working with PyCharm

There are no slides for this bit. Some nice tutorials are available on the web, like this one (https://www.mygreatlearning.com/blog/pycharm-tutorial/#runningacodeinpycharm).

PyCharm also has extensive documentation:

- https://www.jetbrains.com/help/pycharm/creating-and-running-your-first-python-project.html#summary (https://www.jetbrains.com/help/pycharm/creating-and-running-your-first-python-project.html#summary)
- https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html#summary (https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html#summary)

Some important packages

Numpy

- Numpy is the most fundamental package for numerical computations in Python ([user guide (https://docs.scipy.org/doc/numpy/user/index.html)](https://docs.scipy.org/doc/numpy/user/index.html)).
- Basically, it provides a datatype `ndarray` and defines mathematical functions for it
- An array is similar to a `list`, except that
    - it can have more than one dimension;
    - its elements are homogeneous (they all have the same type).
- NumPy provides a large number of functions (*ufuncs*) that operate elementwise on arrays. This allows *vectorized* code, avoiding loops (which are slow in Python).

Constructing Arrays

- Arrays can be constructed using the `array` function which takes sequences (e.g, lists) and converts them into arrays. The data type is inferred automatically or can be specified.

In [ ]:
```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a)
```

In [ ]:
```
a = np.array([1, 2, 3, 4], dtype='float64')  # or np.array([1., 2., 3., 4.])
print(a)
```

- NumPy uses C++ data types which differ from Python's (though `float64` is equivalent to Python's `float`).

- Nested lists result in multidimensional arrays. We won't need anything beyond two-dimensional (i.e., a matrix or table).

In [ ]:
```
a = np.array([[1., 2.], [3., 4.]]); a
```

In [ ]:
```
a.shape # number of rows and columns
```

- Other functions for creating arrays include:

```
In [ ]: np.ones([2, 3])  # there's also np.zeros, and np.empty (which results in an uninitialize
        d array).
```

```
In [ ]: np.arange(0, 10, 2)  # like range, but creates an array instead of a list.
```

Indexing

- Indexing and slicing operations are similar to lists:

In [ ]:
```
a = np.array([[1., 2.], [3., 4.]])
print(a)
a[0, 0] # [row, column]
```

In [ ]:
```
b = a[:, 0]; b # entire first column. note that this yields a 1-dimensional array (vecto
r), not a matrix with one column.
```

- Apart from indexing by row and column, arrays also support *Boolean* indexing:

In [ ]:
```
a = np.arange(10); a
```

In [ ]:
```
ind = a < 5; ind
```

In [ ]:
```
a[ind]
```

A shorter way to write this is

In [ ]:
```
a[a<5]
```

This is useful for selecting elements according to some condition

Arithmetic and `ufuncs`

- NumPy `ufuncs` are functions that operate elementwise:

In [ ]:
```
a = np.arange(1, 5); np.sqrt(a)
```

- Other useful ufuncs are exp, `log`, `abs`, and `sqrt`.
- Basic arithmetic on arrays works elementwise:

In [ ]:
```
a = np.arange(1, 5); b = np.arange(5, 9); a, b, a+b, a-b, a/b.astype(float)
```

Broadcasting

- Operations between scalars and arrays are also supported:

In [ ]:
```
np.array([1, 2, 3, 4]) + 2
```

- This is a special case of a more general concept known as *broadcasting*, which allows operations between arrays of different shapes.
- NumPy compares the shapes of two arrays dimension-wise. It starts with the trailing dimensions, and then works its way forward. Two dimensions are compatible if
    - they are equal, or
    - one of them is 1 (or not present).
- In the latter case, the singleton dimension is "stretched" to match the larger array.

- Example:

In [ ]:
```
x = np.arange(6).reshape((2, 3)); x  # x has shape (2,3).
```

In [ ]:
```
m = np.mean(x, axis=0); m  # m has shape (3,).
```

In [ ]:
```
x-m  # the trailing dimension matches, and m is stretched to match the 2 rows of x.
```

Array Reductions

- *Array reductions* are operations on arrays that return scalars or lower-dimensional arrays, such as the `mean` function used above.
- They can be used to summarize information about an array, e.g., compute the standard deviation:

In [ ]:
```
a = np.random.randn(300, 3)  # create a 300x3 matrix of standard normal variates.
a.std(axis=0)  # or np.std(a, axis=0)
```

- By default, reductions operate on the *flattened* array (i.e., on all the elements). For row- or columnwise operation, the `axis` argument has to be given.
- Other useful reductions are `sum`, `median`, `min`, `max`, `argmin`, `argmax`, any, and `all` (see help).

Saving Arrays to Disk

- There are several ways to save an array to disk:

In [ ]:
```
np.save('myfile.npy', a) # save `a` as a binary .npy file
```

In [ ]:
```
import os
print(os.listdir('.'))
```

In [ ]:
```
b = np.load('myfile.npy')  # load the data into variable b
os.remove('myfile.npy')  # clean up
```

In [ ]:
```
np.savetxt('myfile.csv', a, delimiter=',')  # save `a` as a CSV file (comma seperated va
lues, can be read by MS Excel)
```

In [ ]:
```
b = np.loadtxt('myfile.csv', delimiter=',')  # load data into `b`.
os.remove('myfile.csv')
```

Pandas Dataframes

Introduction to Pandas

- pandas (from *p*anel *d*ata) is another fundamental package ([user guide (http://pandas.pydata.org/pandas-docs/stable/overview.html)](http://pandas.pydata.org/pandas-docs/stable/overview.html)).
- It provides a number of datastructures (*series*, *dataframes*, and *panels*) designed for storing observational data, and powerful methods for manipulating (*munging*, or *wrangling*) these data.
- It is usually imported as pd:

In [ ]:

```
import pandas as pd
```

Series

- A pandas `Series` is essentially a NumPy array, but not necessarily indexed with integers.

In [ ]:
```
pop = pd.Series([5.7, 82.7, 17.0], name='Population'); pop  # the descriptive name is optional.
```

- The difference is that the index can be anything, not just a list of integers:

In [ ]:
```
pop.index=['DK', 'DE', 'NL']
```

- The index can be used for indexing (duh...):

In [ ]:
```
pop['NL']
```

- NumPy's `ufuncs` preserve the index when operating on a `Series`:

```
In [ ]: gdp = pd.Series([3494.898, 769.930], name='Nominal GDP in Billion USD', index=['DE', 'N
        L']); gdp
```

```
In [ ]: gdp / pop
```

- One advantage of a `Series` compared to NumPy arrays is that they can handle missing data, represented as `NaN` (not a number).

Dataframes

- A `DataFrame` is a collection of `Series` with a common index (which labels the rows).

In [ ]:
```
data = pd.concat([gdp, pop], axis=1); data  # concatenate two Series to a DataFrame.
```

- Columns are indexed by column name:

In [ ]:
```
data.columns
```

In [ ]:
```
data['Population']  # data.Population works too
```

- Rows are indexed with the `loc` method:

In [ ]:
```
data.loc['NL']
```

- Unlike arrays, dataframes can have columns with different datatypes.
- There are different ways to add columns. One is to just assign to a new column:

In [ ]:
```
data['Language'] = ['German', 'Danish', 'Dutch']  # add a new column from a list.
```

- Another is to use the `join` method:

In [ ]:
```
s = pd.Series(['EUR', 'DKK', 'EUR', 'GBP'], index=['NL', 'DK', 'DE', 'UK'], name='Curren
cy')
data.join(s)  # add a new column from a series or dataframe.
```

- Notes:
    - The entry for 'UK' has disappeared. Pandas takes the *intersection* of indexes ('inner join') by default.
    - The returned series is a temporary object. If we want to modify `data`, we need to assign to it:

In [ ]:

```
data = data.join(s)
```

- To add rows, use `loc` or append:

In [ ]:

```
print(data.loc["DE"])
data.loc['AT'] = [386.4, 8.7, 'German', 'EUR']  # add a row with index 'AT'.
s = pd.DataFrame([[511.0, 9.9, 'Swedish', 'SEK']], index=['SE'], columns=data.columns)
data = data.append(s)  # add a row by appending another dataframe. May create duplicate
s.
data
```

- The dropna method can be used to delete rows with missing values:

In [ ]:
```
data = data.dropna(); data
```

- Useful methods for obtaining summary information about a dataframe are `mean`, `std`, `info`, `describe`, `head`, and `tail`.

In [ ]:
```
data.describe()
```

In [ ]:
```
data.head()  # show the first few rows; data.tail shows the last few
```

- To save a dataframe to disk as a csv file, use

In [ ]:
```
data.to_csv('myfile.csv')  # to_excel exists as well.
```

- To load data into a dataframe, use `pd.read_csv`:

In [ ]:
```
pd.read_csv('myfile.csv', index_col=0)
```

In [ ]:
```
os.remove('myfile.csv')  # clean up
```

Pandas can also open CSV files directly from a URL:

In [ ]:
```
URL = "https://covid.ourworldindata.org/data/owid-covid-data.csv"
df = pd.read_csv(URL)
df.head()
```

Working with Time Series

Data Types

- Different data types for representing times and dates exist in Python.
- The most basic one is `datetime` from the eponymous package:

In [ ]:
```
from datetime import datetime
datetime.today()
```

- `datetime` objects can be created from strings using `strptime` and a format specifier:

In [ ]:
```
datetime.strptime('2017-03-31', '%Y-%m-%d')
```

- Pandas uses `Timestamps` instead of `datetime` objects. Unlike timestamps, they store frequency and time zone information. The two can mostly be used interchangeably.

In [ ]:
```
pd.Timestamp('2017-03-31')
```

- A time series is a `Series` with a special index, called a `DatetimeIndex`; essentially an array of `Timestamps`.
- It can be created using the `date_range` function.

In [ ]:
```
myindex = pd.date_range(end=pd.Timestamp.today(), normalize=True, periods=100, freq='B')
P = 20 + np.random.randn(100).cumsum()  # make up some share prices.
aapl = pd.Series(P, name="AAPL", index=myindex)
aapl.tail()
```

- As a convenience, Pandas allows indexing timeseries with date strings:

In [ ]:
```
aapl['4/11/2022']
```

In [ ]:
```
aapl['4/11/2022':'4/12/2022']
```

Reading (recommended)

- [https://python-course.eu/numerical-programming/ (https://python-course.eu/numerical-programming/)](https://python-course.eu/numerical-programming/) 1-14, 23-28, 33-34

Homework

- Ex. 1-16 of https://github.com/rougier/numpy-100/blob/master/100_Numpy_exercis (https://github.com/rougier/numpy-100/blob/master/100_Numpy_exercises.md). Ski in PyCharm.
- Read https://python-course.eu/numerical-programming/pandas-groupby.php (https:/ course.eu/numerical-programming/pandas-groupby.php), then do https://github.com/guipsamora/pandas_exercises/tree/master/01_Getting_%26_Kno (https://github.com/guipsamora/pandas_exercises/tree/master/01_Getting_%26_Kno