Python Cookbook and Quick Reference

One of the most essential things to learn in python is about the various libraries, which are very general and powerful, although you might not see these at first on an introductory python course. For machine learning the most essential ones to know about are: numpy, matplotlib, pandas, seaborn, sklearn, tensorflow and pytorch (though we don"t cover the latter in this course, as it is more advanced).

There is a lot of very useful code in the workshops – do make sure you look at these – but here is a collection of many useful bits of code together in one document for convenient reference.

Most of the following is illustrated with specific examples just to give an indication of how they work, but they can often be used much more generally. There are *many* other things that are also useful within these libraries, but the pieces included here are a good starting point.

1. Fundamental python

F-Strings

```
  # Useful strings (the letter f followed by quotes) that allow variable substitution within {}: e.g.
fname = 'file.txt'    # this is not an f-string, just a normal string
msg1 = f'Filename is {fname}'    # this is now a string = 'Filename is file.txt'
x = 3.427
msg2 = f'X = {x}'   # this is now a string = 'X = 3.427'
msg3 = f'X = {x:.2f}'   # this is now a string = 'X = 3.43' – as :.2f = two decimal places
```

Lists

```
  # Create a list
mlist = [ 1.2, 2.0, -3.7 ]
mlist2 = [ [ 1.1, 2.2 ] , [ 3.3, 4.4 ] ]    # list of lists
  # Length of list
len(mlist)
  # Access list elements
mlist[0]     # first element (1.2 from above)
mlist[2]     # third element (-3.7 from above)
mlist[-1]    # last element (in this case, same as third element)
mlist[:2]    # first two elements (defined by the slice notation)
mlist[1:]    # second element through to the last
mlist2[1]   # access second list ([3.3, 4.4] from above)
mlist2[0][1]   # access first row (list), second column (value is 2.2 from above)
  # Many other options exist for accessing and manipulating lists …
```

Dictionaries

```
  # Create a dictionary
mdict={"a":3, "b":5}   # set of key-value pairs (types can be anything)
mdict["a"]      # access value via key
mdict["c"] = 1   # assign new entry
mdict.keys()   # provides set of keys (arbitrary order)
for k in mdict:    # acts as shorthand for k in mdict.keys()
mdict.values()   # get set of values
  # Many other options exist for accessing and manipulating dictionaries …
```

Copy

```
  # Be warned: default use of references in python means explicit copies are needed now and then
a = [1, 2]
b = a
a[0] = 5
print(b)     #  gives [5,2] as b is a reference to a
import copy
copy.copy()  # shallow copy
copy.deepcopy()   # deep copy (recursive copy of all parts, sub-parts, etc) – e.g. for list of arrays
b = copy.copy(a)
a[0] = 7
print(b)    # gives [5,2] as b and a are now separate copies
```

2. Numpy (numerical data, vectors, arrays)

```
import numpy as np
  # Create n-dimensional array (ndarray - NB: np.array creates an ndarray object)
mdata = np.array([1, 10, 4.3, -0.5])    # 1D array (filled with specific numbers from a list)
mdata = np.zeros((nelements))    # 1D array of specified size (filled with zeros)
mdata = np.zeros((nrows, ncols))    # 2D array of specified size (filled with zeros)
mdata = np.zeros((nx, ny, nz))    # 3D array of specified size (filled with zeros)
  # Access elements (counting starts at zero)
mdata[2]   # third element of 1D array
mdata[1] = 7.2  # assign to second element (assignments possible for all selections shown here)
mdata[4, 1]   # element in fifth row, second column
mdata[4][1]   # element in fifth row, second column (less efficient)
mdata[4]        # whole of fifth row
mdata[:, 1]   # all elements in second column (1D array)
mdata[1, :]   # all elements in second row (1D array)
mdata[:3, 5:10]   # submatrix - 3 by 5 - first three rows and five columns (5 to 9 inclusive)
mdata[bselect,:]  # use a Boolean vector (bselect) to pick out rows (or columns) where value is True
  # Size of array
mdata.shape
mdata.reshape([-1, 1])    # turn a 1D vector – of shape (N,) – into an N by 1 (column) vector
mdata.reshape([1, -1])    # turn a 1D vector – of shape (N,) – into a 1 by N (row) vector
mdata.reshape([nx,ny,…])  # reshape into any size, any dimensions, with same number of elements
  # Check for NaNs (not a number)
np.isnan(mdata)   # creates a binary array (one binary value per element) - can use to select
rows/cols
  # Other operations (illustrated for 2D arrays, but can be applied to other dimensionalities)
np.concatenate((arr1, arr2, …), axis=0)  # Use axis=1 to extend columns, or axis=0 to extend rows
np.sum(mdata, axis=1)    # sum across columns to get single row (use axis=0 to sum across rows)
     # there are also mean, std, and lots of other options available
  # select columns that do not have any NaN entries (could be done for columns instead)
mdata[:, ~np.any(np.isnan(mdata, axis=0))]
  # load and save csv files (or more general text files)
mdata = np.loadtxt("datafile.csv", delimiter=",")
np.savetxt("foo.csv", mdata, delimiter=",")
  # load and save in binary format
mdata = np.load("foo.npy")
np.save("foo.npy", mdata)
```

3. Plotting with matplotlib

```
%matplotlib inline
  # the line above is only needed within a notebook, not within general python scripts
import matplotlib.pyplot as plt
plt.figure()   # make a new figure
plt.figure(figsize=(12,10))   # specify the required size of the figure, in inches – this one is quite large
x_values = np.linspace(10000, 55000, 1000)   # make regularly spaced values: start, end, step
pltv1 = plt.plot(x_values, y_values, "r")   # last arg is red; other matlab-style colours/styles
available
plt.show()   # nothing is displayed until this is called

  # display image
plt.imshow(x_img, cmap="gray")
  # save figure to file (pdf or other formats)
pltv1.figure.savefig("myfig.pdf", format='pdf', dpi=300)
```

4. Pandas (high-level data manipulation)

```
import pandas as pd
  # Create a pandas dataframe from an n-dimensional array
mdf = pd.DataFrame(data=mdata, columns=['blah', 'blah', …])   # mdata is a numpy array
  # Read from a csv file
mdf = pd.read_csv("lifesat.csv")
  # Report basic info and summary info
mdf.info()
mdf.describe()
  # List of column names
list(mdf.columns)

# Access a column via its name
mdf["Country"]
  #  … or the same thing can be done as an attribute
mdf.Country
  # Access a row via its positional number/index
  #   Note there is iloc[] and loc[]; loc[] is original and iloc[] is most recent ordering (e.g. after sorting)
mdf.iloc[3]
  # Access a subset of rows and columns
mdf.iloc[2:5,3:8]
  # Access a single element (two options – though there are more than this)
mdf.iloc[3,5]
mdf["Country"][3]
  # select parts of table based on search
mdf[mdf["GDP per capita"]<10000]
mdf[(mdf["GDP per capita"]<10000) & (mdf["Life satisfaction"]==6.0)]
  # select based on set of options (useful for categorical variables)
mdf[mdf["ocean_proximity"].isin(["NEAR BAY","NEAR OCEAN"])]

# Dealing with invalid / non-numeric data
mdf.isna()   # returns boolean array, depending on whether values are valid or null/NaN
mdf[~mdf.median_house_value.isna()]   # select rows without null/NaN values
  # remove rows (or columns) that contain NaN/null values
mdf.dropna()
mdf.dropna(subset=["total_rooms", "total_bedrooms"])   # only consider some columns

# combining options (often like database operations)
mdf_new = pd.concat([mdf1, mdf2])      # extend by adding rows (use axis=1 for column-wise)
pd.merge(mdf1, mdf2, on='ID', how='outer')      # combine using column named 'ID' to align
  # an outer join uses all rows, from both dataframes, regardless of whether they match
pd.merge(mdf1, mdf2, on='ID', how='inner')      # combine using column named 'ID' to align
  # an inner join only uses rows that match
  # other options for merging also exist


# convert to numeric values as much as possible
data.apply(pd.to_numeric, errors="coerce")
  # quick printout of the total of values that are missing/NaN/strings/dashes/etc
np.sum(np.isnan(data.apply(pd.to_numeric, errors="coerce")))

  # convert to numpy array
mdata = mdf.to_numpy()
```

5. Plotting with seaborn (pandas)

```
import seaborn as sns; sns.set()
sns.scatterplot(data=mdf, x="population", y="median_income", hue="ocean_proximity")
  # other seaborn plot options (for some types) inclue style, palette, legend and s (for size)
sns.jointplot(data=mdf, x="population", y="median_income")
sns.violinplot(data=mdf, x="ocean_proximity", y="median_income", cut=0)
  # use cut=0 to stop extrapolation past the min/max data limits
  # still need plt.show() to display
```

6. Machine Learning Models (traditional)

```
import sklearn
from sklearn import linear_model
# Select a linear model  (can replace "LinearRegression" with many other options)
model = sklearn.linear_model.LinearRegression()
# Train the model
model.fit(X, y)
# Apply the trained model to make predictions
Y_new = model.predict(X_new)
# Can also access method-specific properties (these are just for LinearRegression)
intercept, slope = model.intercept_[0], model.coef_[0][0]
```

7. Machine Learning Models (deep learning)

Keras Fundamentals

```
import tensorflow as tf
from tensorflow import keras
```

Sequential Style (Dense Network example)

```
# Build a network using the sequential style (output of one layer becomes input to the next)
# Start a model, with the selected style
model = keras.models.Sequential()
# Add layers to the model
model.add(keras.layers.Flatten(input_shape = [28, 28, 1]))    # always have same sized inputs
model.add(keras.layers.Dense(200, activation = "relu", kernel_initializer="he_uniform"))
model.add(keras.layers.Dense(10, activation = "softmax"))  # always have N output classes
# Compile the model (specifying the loss function, optimiser, metrics, etc.)
model.compile(loss="sparse_categorical_crossentropy", optimizer="Adam",
              metrics=["accuracy"])
# Train the model (specifying training data, number of epochs, callbacks, etc.)
history = model.fit(X_train, y_train, epochs=n_epochs, callbacks = [early_stopping_cb],
                    validation_data=(X_valid, y_valid),verbose=1)

# Look in history object for training curves and information
max_val_acc = np.max(history.history["val_accuracy"])
plt.plot(np.arange(0,n_epochs),history.history["accuracy"], color="orange")
plt.plot(np.arange(0,n_epochs)+0.5,history.history["val_accuracy"],"r")  # offset validation curves

# Run trained model on new (test) examples
testres = model.evaluate(X_test, y_test, verbose=0)
```

Additional Options

```
# Early stopping callback
early_stopping_cb = keras.callbacks.EarlyStopping(monitor="val_accuracy",
                          patience=5, restore_best_weights=True)

# Learning rate scheduler
lr_schedule = keras.optimizers.schedules.ExponentialDecay(learningrate,
                          decay_steps=n_steps, decay_rate=0.1)
#  … and then use optimizer=optimizer(learning_rate=lr_schedule) in the compile call
# Note that n_steps is in terms of number of batches, not epochs

# Save the model weights etc (checkpoint callback) – good practice to do regularly during training
checkpoint = keras.callbacks.ModelCheckpoint("weights-{epoch:02d}.hdf5", verbose=1,
                          monitor="val_accuracy", save_best_only=True, mode="max")

# Load model weights (after defining model but before compiling)
model.load_weights("weights-10.hdf5")
```

Other Layers

```
# Convolutional Neural Network (CNN) layer
model.add(keras.layers.Conv2D(filters=n, kernel_size=3, strides=1, padding="same",
                              activation="relu"))

# Dropout layer
#  … in a standard dense layer (the usual case)
model.add(keras.layers.Dropout(rate = 0.2))
# … or for CNNs it is possible to dropout whole fiters (though this is less commonly used)
model.add(keras.layers.SpatialDropout2D(rate = 0.5))

# Batch normalisation layer (usually between dense layers or between conv and pooling layers)
model.add(keras.layers.BatchNormalization())
```

Keras – Functional Style

```
# Example in Sequential style
model = keras.models.Sequential()   # This is a style of building networks - the easiest option
model.add(keras.layers.Flatten(input_shape = [28 , 28]))  # inputs are 28 x 28 arrays - make 1D
model.add(keras.layers.Dense(300, activation = "relu"))    # first hidden layer
model.add(keras.layers.Dense(100, activation = "relu"))    # second hidden layer
model.add(keras.layers.Dense(10, activation = "softmax"))  # output layer

# Equivalent in Functional Style (note that inputs and outputs are explicitly assigned)
layer1 = keras.layers.Input(shape = [28 , 28])   # inputs are 28 x 28 arrays
hidden0 = keras.layers.Flatten(input_shape = [28, 28])(layer1)  # make 1D
hidden1 = keras.layers.Dense(300, activation = "relu")(hidden0)    # first hidden layer
hidden2 = keras.layers.Dense(100, activation = "relu")(hidden1)    # second hidden layer
output = keras.layers.Dense(10, activation = "softmax")(hidden2)  # output layer
model = keras.Model(inputs = layer1, outputs = output)

# Note that inputs can be combinations of several (e.g. for skip connections) – such as:
combo1 = keras.layers.concatenate([hidden1, hidden2])
output2 = keras.layers.Dense(10, activation = "softmax")(combo1)
```