# Bibliotheken

**Doctest**: Um Methoden zu testen.
*Nutzung*: Importieren von `doctest` - am besten erst in der `if __name__ == "__main__":`. Dann in die Doc-Strings einer Funktion einfach die Test-Aufrufe der jeweiligen Funktion schreiben wie man sie im REPL auch schreiben würde. Durch Aufrufen von `doctest.testmod()` werden die test dann ausgeführt.
*Beispiel*:

```
def test(a: int):
        """ Bla
        >>> test(1)
        1
        >>> test(5) # doctest: +SKIP
        6
        """
        return a

if __name__ == "__main__":
        import doctest
        doctest.testmod()
```

**Time**: Um die Laufzeit von Funktionen zu messen.
*Nutzung*: Importieren von `time`. `time.time()` gibt dann die Zeit als Float in Sekunden seit beginn einer Epoche (Unix 1. Januar 1970 00:00:00 UTC) an. Oder auch mit `time.process_time()` bzw. `time.process_time_ns()`.
*Probleme*: Die Scheduling-Strategien verhindern zuverlässige Voraussage wann ein Task dran kommt. Der Python Garbage Collector. Usw.
Deshalb: Immer mit angeben:

- Messmethode (Anzahl der Wiederholungen, wie gemessen, ...)
- verwendete Hardware
- Betriebssystem und Einstellungen
- verwendete Software

**Timeit**: Um die Laufzeit von Funktionen zu messen.
*Nutzung*: Importieren von `timeit`. Mit `timeit.timeit(stmt='pass', setup='pass', timer=<default>, number=1000000, globals=None)` kann dann die Ausführung von "stmt" genau "number" mal getimet werden. Wichtig: Timeit mittelt die Ergebnisse nicht!

**Tkinter**: Tkinter basiert darauf, dass jedes Fenster aus Widgets besteht die durch einen Layout-Manager auf dem Fenster platziert werden. Es gibt dabei die folgenden Widgets:

- `tk.Tk()` Erzeugt ein Root-Widget bzw. einfach ein Fenster
- `tk.Label(master, option, ...)` Erzeugt einen Text. Wobei master das Fenster und option z.B. `text="blabla"` sein kann.

- `tk.Frame(parent, option, ...)` Ist ein Container für andere Widgets und erlaubt Layout-Manager zu Schachteln. Optionen können sein `width, height, reflief='ridge'`
- `tk.Buttom(parent, option, ...)` Ist ein Knopf. Übliche options sind `text, command`
- `tk.Entry-Widget(parent)` Ermöglicht es Text einzugeben. Mittels `my_entry.get()` kann man den Text einlesen, mittels `my_entry.delete(0,tk.END)` kann es geleert werden und mit `my_entry.insert(0,"bla")` kann text eingefügt werden.
- `tk.Radiobutton(root, text, padx, variable, command, value)` Ist ein RadioButton wobei die variable eine Steuervariable des selben Datentyp wie value sein muss.
  Außerdem gibt es die folgenden Layout-Manager, wobei pro Fenster nur einer genutzt werden kann:
- `my_widget.pack()` bestimmt Position, Größe etc. automatisch.
- `my_widget.grid(row,column)` bestimmt die Position eines Widgets durch einen row und column Wert. Die Größe der Zellen Bestimmt der Algorithmus.
  Um nun die Eingabe noch rekativ zu machen, bietet Tkinter noch Steuervariablen an. Falls sich der Wert einer solchen Variable ändert, so ändert sich auch das verknüpfte Widget. Alle diese Variablen haben zwei Methoden `var.get()` um den Wert auszulesen und `var.set(value)` um den Wert zu setzen. Es gibt die folgenden Arten von Variablen
- `tk.DoubleVar()`
- `tk.IntVar()`
- `tk.StringVar()`
- `tk.BooleanVar()`

Hier etwas Beispiel Code:

```
import tkinter as tk # Import der Library

root = tk.Tk() # Erzeugen eines Root-Widgets. Es kann in jeder Anwendung nur ein root
geben

v = tk.IntVar() # Steuervariable für den RadioButton

root.title("Mein Titel") # Setzen des Fenster Namens

w = tk.Label(root, text='Hallo') # Ein Label Widget
w.pack() # gibt an welcher Layout-Manager verwendet werden soll

tk.Radiobutton(root, text="Python", vaiable=v, value=1).pack()
tk.Radiobutton(root, text="Haskell", vaiable=v, value=2).pack()

root.mainloop() # startet die Endlosschleife = Ereignisschleife.
```

**Numpy**: Numpy(kurz: np) bringt viele Operationen die mit (mehrdimensionalen) Arrays zu tun haben. Man unterscheidet dabei zwischen:

- Vektoren: Vektoren haben eine Achse und eine Dimension von $(n,)$
- Matrizen: Eine Matrix hat zwei Achsen und eine Dimension von $(m, n)$
- Tensor: Ein Tensor hat drei Achsen und eine Dimension von $(m, n, p)$
  Um einen solchen Numpy Array aus einer Python liste zu generieren gibt es die Funktion `np.array(my_list)`. Das Attribut `my_np_array.shape` gibt dann an welche Dimension der

Array von Außen nach Innen hat. (Also zuerst die Äußerste Liste, dann die eins inner bei usw.).
Man kann dabei Slicing genau gleich wie bei Listen machen. Besonders ist, das man dies auch für mehrer Dimensionen machen kann. Man trennt die Achsen dann durch ein Komma. Also `my_matrix[0:3,0:3]`. Weitere Relevante Funktionen sind:

- `my_array.reshape(x,y,...)`: Legt alle Elemente Reihe für Reihe in eine Liste und baut dann Reihe für Reihe wieder einen Array.
- `np.zeros(dim)` Füllt einen Array gegebener Dimension mit 0.
- `np.ones(dim)` Füllt einen Array gegebener Dim mit 1.
- `np.arange()` Wie `range()` aber gibt einen Array retour.
- `+,-,*,/,**` Alle Elementweise.

**Pandas**: Pandas(kurz: pd) bringt Funktionen zur Datenanalyse mit. Die Hauptkomponenten sind dabei Series und DataFrames. Eine Serie ist eine Spalte und ein DataFrame eine Tabelle. Einen DataFrame kann man in Jupyter Notebook einfach durch schreiben der Variable anzeigen lassen. Um ein Dict in einen DataFrame umzuwandeln nutzt man einfach den Konstruktor. `pd.DataFrame({'Äpfel':[2,3,2], 'Orangen': [4,1,0]})`. Dabei entspricht jeder Key einer Spalte und die Values die Einträge in den Zeilen. Der Index, also die Beschriftung der Spalten ist automatisch eine Zahl beginnend bei 0. Mann kann diesen durch Angabe einer `index=[...]` Liste ändern.
Um eine Spalte anzuzeigen nutzt man einfach Indexing also z.B.: `einkäufe['Orangen']`.
Um eine Zeile zu finden nutzt man die Methode `einkäufe.loc['Lily']`.
Will man csv Dateien Laden gibt es die Funktion `pd.read_csv(file, index_col=...)`. Wobei `index_col` einfach die Spalte des CSVs angibt, dessen Werte als Index genutzt werden sollen (default aufsteigende Zahlen). Mittels `my_dataframe.to_csv(filename)` kann man einen DataFrame als csv Speichern.
Weitere Operationen sind:

- `my_dataframe.head()`: Gibt die ersten 5 Zeilen aus.
- `my_dataframe.tail(n)`: Gibt die letzen $n$ Zeilen aus.
- `my_dataframe['bla']`: Um die Spalte 'bla' als Serie zu extrahieren.
- `my_dataframe[['bla']]`: Um die Spalte 'bla' als DataFrame zu extrahieren.
- `my_dataframe[['bla', 'blib']]`: Um zwei Spalten als DataFrame zu extrahieren.
- `my_dataframe_or_series.to_numpy()`: Um von Pandas zu Numpy zu konvertieren.
- `pd.DataFrame(data=my_array, columns=['Name'])`: Um von Numpy zu Pandas zu konvertieren.
- `my_dataframe.columns` gibt die Spalten des DataFrames aus.
- `.describe()` gibt Infos über die Verteilung an.
- `my_series.value_counts()` gibt die Häufigkeit aller Werte einer Spalte an.
- `my_dataframe[condition]` kann genutzt werden um einen DataFrame nach Bedingungen zu filtern. Also z.B.: `movies_df[(movies_df['director'] == 'Christopher Noaln') | True & False ...]

**MatPlotLib.Pyplot**: MatPlotLib(kurz: plt) dient dem Ploten von Daten. Man erzeugt dabei zuerst eine Figure mit: `fig = plt.figure(figsize=(4,3))`. Auf dieser kann man dann mittels

- `plt.plot(x, y, '-', label='')` eine Funktion zeichnen
- `plt.scatter(x, y, color='red')` einen Scatterplot zeichnen.
  Mit `plt.ylabel('')` und `plt.xlabel('')` und `plt.title('')` und `plt.legend()` kann man dann noch weiter Einstellungen treffen. Am ende kann man den Plot mit `plt.show()` zeigen. Figures können mit `fig.savefig(filename)` gespeichert werden.

**Seaborn**: Seaborn(kurz: sns) bietet eine Möglichkeit DataFrames zu visualisieren.

- `sns.load_dataset('name')` : Um einen Datensatz als DataFrame zu laden.
- `sns.scatterplot(x='x_axies', y='y_axies', hue='col', data=my_dataframe)` : Um ein Scatterplot zu zeichnen, welcher auf der X-Achse die Elemente aus der Spalte 'x_axies' und auf der Y-Achse die Elemente aus der Spalte 'y_axies' gefärbt nach der Spalte 'col' anzeigt.