Funktionen

→ basic func: def func_name(params):
indent → #statements
return expression #optional

Wichtig! Wie immer in Python zeigen Leerzeichen (spaces) an, in welchem code Block wir sind!

d.h. def bla(a):
print(a) geht nicht!

→ 4 (selten auch 2) spaces indent!

→ mehr im Detail:
(mehr als in Vorlesung)

sog. Type Hinting

```
def function_name(parameter: data_type) -> return_type:
    """Docstring"""
    # body of the function
    return expression
```

→ optional, ändert an sich nichts am Prog.

→ kann hilfreich sein um Bugs zu finden, in grossen Projekten etc

A Python docstring is a string literal used to document a Python module, class, function or method, so programmers can understand what it does without having to read the details of the implementation.

⇒ https://medium.com/vacatronics/why-you-should-consider-python-type-hints-770e5cb1570f

→ pass: Platzhalter für Code

e.g. def add(a, b):
""" adds two numbers """
pass

quasi #TODO

→ Python braucht pass, damit nachher noch Code kommen kann ohne indent Fehler.

numpy

zum Vgl.

→ numpy erleichtert operationen / Umgang mit Matrizen und allg. numerischen Aufgaben um Einiges

Python Listen

- Mehrdimensionale Listen

```
l = [[1, 2, 3], [4, 5], [6]]
```

- Matrizen (rechteckige Listen)

```
l = [[1, 2, 3], [4, 5, 6]]
```

1	2	3
[0][0]	[0][1]	[0][2]
4	5	6
[1][0]	[1][1]	[1][2]

- Zugriff auf Elemente

```
l[0][2] # = 3 (l[0, 2] geht nicht!)
```

l[0] = [1, 2, 3]

- Ausgabe

```
print(l) # gibt [[1, 2, 3], [4, 5, 6]] aus
```

- Elementweise Multiplikation?

```
[[2*x for x in r] for r in l]
```

51

→ einige Arten numpy arrays zu erstellen (→ Vorlesung)

- mit Sequenzen

```
l = [1, 2, 3, 4]
a = np.array(l)
b = np.array(range(2, 10, 3)) # = [2, 5, 8]
A = np.array([[1, 2], [3, 4]])
```

- mit zufälligen Zahlen aus [0, 1)

```
R = np.random.random(10) # zehn Zufallszahlen aus [0,1)
```

- mit zufälligen Zahlen in Intervall

```
R = np.random.uniform(-1, 1, 5) # fünf Zufallszahlen aus [-1, 1]
```

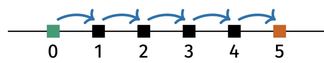
- mit zufälligen ganzen Zahlen in Intervall

```
R = np.random.randint(1, 7, 10) # zehn Würfelwürfe
```

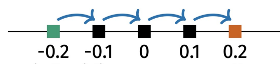
53

- mit Wertebereichen `np.arange`
  ähnlich wie `np.array(range(start, stop, step))`

```
np.arange(start, stop, step)
np.arange(start, stop) # step = 1
np.arange(stop) # start = 0, step = 1
a = np.arange(5) # = [0, 1, 2, 3, 4]
```




```
b = np.arange(-0.2, 0.2, 0.1) # = [-0.2, -0.1, 0, 0.1]
```

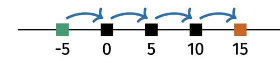


- mit äquidistanten Intervallen `np.linspace`
  definiert durch Anfang, Ende (**inklusiv!!**) und **Anzahl Schritte**

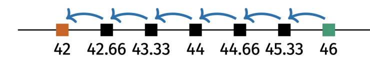
```
np.linspace(start, stop, num)
np.linspace(start, stop) # num = 50
```

Schrittgrösse: $\frac{\text{stop}-\text{start}}{\text{num}-1}$ stepsize

```
a = np.linspace(-5, 15, 5) # = [-5., 0., 5., 10., 15.]
```




```
b = np.linspace(46, 42, 7) # [46, 45.33, 44.66, 44.6, 42.66, 42.]
```



55

→ operationen + slicing auf numpy arrays

- Zugriff auf Element

```
a = np.linspace(2, 9, 8) # = [2, 3, 4, 5, 6, 7, 8, 9]
a[1]  # = 3
a[-1] # = a[-1 + len(a)] = a[-1 + 8] = a[7] = 9
```

2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

```
A = np.array([[1, 2, 3], [4, 5, 6]])
A[1, 2] # = A[1][2] = 6
```

geht mit normalen listen nicht

	0	1	2
0	1	2	3
1	4	5	6

57

- Slicing (Teilarrays)

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	3	1	2

```
A = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 3, 1, 2]])
A[1] # = A[1,:]  = [5, 6, 7, 8] (Zeile)
A[:, 2] # = [3, 7, 1] (Spalte)
A[1:2, 1:3] # = [[6, 7]]
A[1:2, 1:]  # = A[1:2, 1:4] = [[6, 7, 8]]
A[:, 1:3] # = A[0:3, 1:3] = [[2, 3], [6, 7], [3, 1]]
A[-2:-1, ::2] # = A[1:2, 0:4:2] = [[5, 7]]
```

A, B seien np.array(...)

→ element-wise addition A + 1 , A + B

〃 multipl. A * 2 , A * B

power A ** 2

→ matrix multipl. A @ B

⋮ etc

⇒ es gibt für fast alles schon eine (oder mehrere) Fkt
d. h. wenn ihr etwas benötigt googelt einfach

! Natürlich aufpassen bei den Aufgaben falls lib. Fkt nicht erlaubt sind.