Numerisches Python

Numerisches Python (engl. *Numeric Python*) soll eine Andeutung auf numerisches Programmieren und das python-spezifische Modul `NumPy` sein. Python ist in Wissenschaft und Forschung weit verbreitet, was nicht zuletzt an den hervorragenden Möglichkeiten liegt, mit Python große Datenmengen elegant und *effizient* zu verarbeiten.

Sobald es um die Lösung numerischer Probleme geht, ist die Leistungsfähigkeit von Algorithmen von höchster Wichtigkeit, sowohl was die Geschwindigkeit als auch den Speicherverbrauch betrifft. Reines Python - also wie bisher, ohne den Einsatz irgendwelcher Spezialmodule - würde sich nicht eignen für Aufgaben, für die z.B. MATLAB gemacht ist.

Nutzen Sie Python in Kombination mit den Modulen `NumPy`, `SciPy`, `Matplotlib` und `Pandas`, dann gehört die Programmiersprache zu den führenden numerischen Programmiersprachen und ist mindestens so effizient wie MATLAB.

- `NumPy` ist ein Modul, das grundlegende Datenstrukturen zur Verfügung stellt, die auch von `Matplotlib`, `SciPy` und `Pandas` benutzt werden. Es implementiert mehrdimensionale Arrays und Matrizen, und gibt wesentliche Funktionalitäten an die Hand, mit denen sich diese Datenstrukturen erzeugen und verändern lassen.

- `SciPy` baut auf `NumPy` auf, d.h. es baut auf den Datenstrukturen auf. Es erweitert die Leistungsfähigkeit mit weiteren Funktionalitäten wie beispielsweise Regression, Fourier-Transformation, Lösen von LGS und viele weitere.

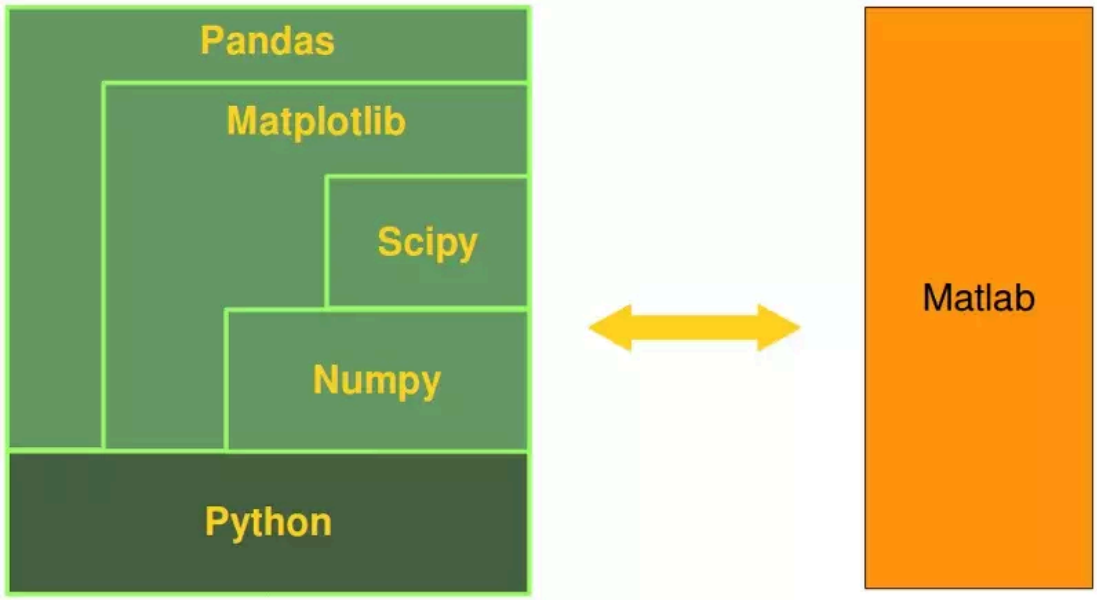
- `Pandas` ist das "jüngste Kind" in dieser Modulfamilie. Es benutzt alle bisher genannten Module und baut auf diesen auf. Der Fokus besteht darin, Datenstrukturen und Operationen zur Verarbeitung von Tabellen und Zeitreihen bereitzustellen. Der Name ist von "Panel data" abgeleitet. Es eignet sich daher z.B. bestens, um mit Tabellendaten, wie sie von Excel erzeugt werden, zu arbeiten.

- Menschen sind unheimlich gut darin, Bilder zu interpretieren, Computer haben ihre Stärke in der Auswertung von Zahlen. Um nun die Früchte der Datenverarbeitung mit Python zu ernten, will man oft die Daten als graphische Plots ausgeben. Am einfachsten lässt sich dies in Python mit dem Modul `Matplotlib` bzw. dessen Untermodul `PyPlot` realisieren.

Python als Alternative zu MATLAB

Ein wesentlicher Nachteil von MATLAB gegenüber Python sind die Kosten. Python mit all seinen Modulen ist kostenlos, wohingegen MATLAB recht teuer ist und je nach Toolbox noch viel teurer werden kann. Bei Python handelt es sich aber nicht nur um *kostenlose*,

sondern auch um *freie* Software, d.h. ihr Einsatz ist nicht durch irgendwelche Lizenzmodelle eingeschränkt.



Quelle: B. Klein, „Numerisches Python", Carl Hanser Verlag GmbH & Co. KG, 2023, S. 5

9. NumPy

9.1 Einführung

Zusammenfassung und Vergleich von NumPy-Datenstrukturen und Python

generelle Vorteile von Python-Datenstrukturen:

- `int` und `float` sind als mächtige Klassen implementiert. Daher können z.B. `int`-Zahlen beinnahe "unendlich" groß oder klein werden.
- Mit `list` hat man effiziente Methoden zum Einfügen, Anhängen und Löschen von Elementen
- `dict` bieten einen schnellen *Lookup*

Vorteile von NumPy-Datenstrukturen gegenüber Python:

- Array-basierte Berechnungen
- effizient implementierte mehrdimensionale Arrays
- gemacht für wissenschaftliche Berechnungen

Ein einfaches Beispiel

Zuerst muss das Modul `numpy` mit dem Befehl `import numpy as np` importiert werden. Dabei hat sich der Alias `np` als Konvention durchgesetzt. **Vermeiden** Sie `from numpy import *`.

In [4]:
```
# einfache Beispiele
# TODO Modul importieren
import numpy as np
```

```
# Temperaturwerte in Grad Celsius
cvalues = [20.8, 21.9, 22.5, 22.7, 22.3, 21.0, 21.2, 20.9, 20.1]

# TODO Erzeugen eines eindimensionalen NumPy-Arrays
c = np.array(cvalues)
print(c)

# TODO Temperaturwerte in Grad Fahrenheit umrechnen
f = c * 9/5 + 32
print(f)

# TODO Umrechnung mit reiner Python-Lösung
fvalues = [x*9/5+32 for x in cvalues]
print(fvalues)

# TODO Ausgabe des NumPy-Typs
print(type(c))
```

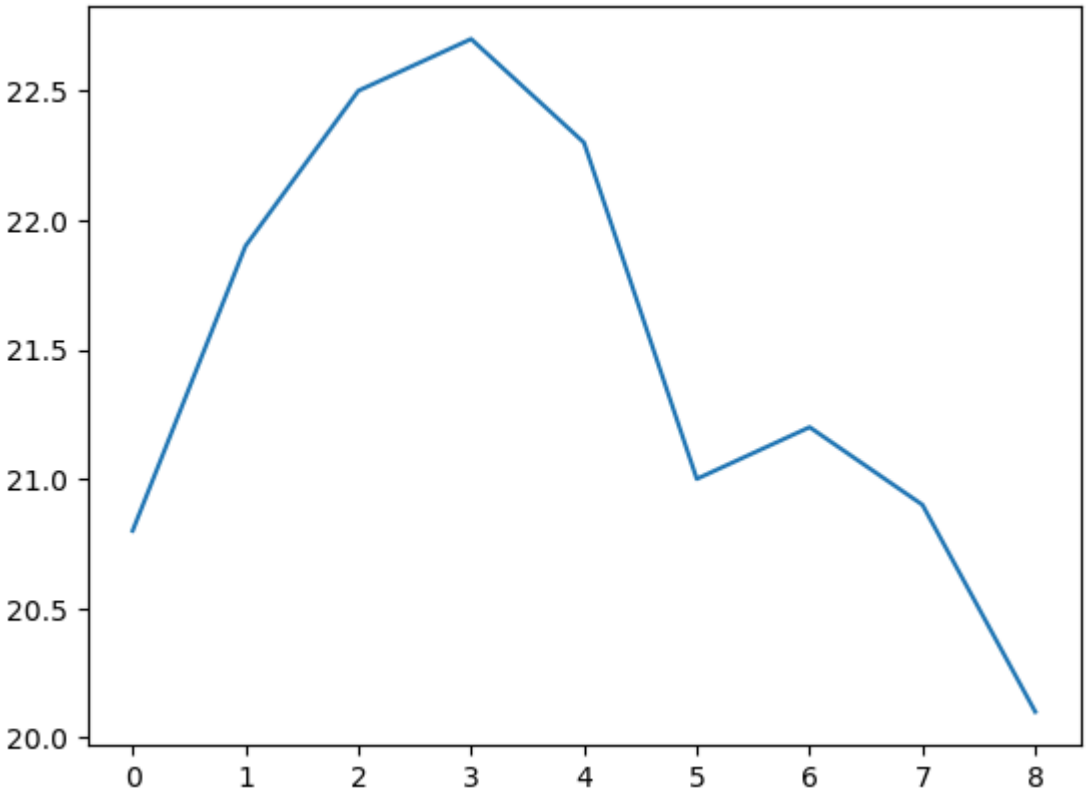
```
[20.8 21.9 22.5 22.7 22.3 21.  21.2 20.9 20.1]
[69.44 71.42 72.5  72.86 72.14 69.8  70.16 69.62 68.18]
[69.44, 71.42, 72.5, 72.86, 72.14, 69.8, 70.16, 69.62, 68.18]
<class 'numpy.ndarray'>
```

`ndarray` steht für *n-dimensional array* und wird synonym zur Bezeichnung *Array* verwendet.

Einfache Visualisierung von Werten

Auch wenn das Modul `Matplotlib` erst später im Detail besprochen wird, soll die einfachste Anwendung des Moduls - die zum Plotten der obigen Werte - an dieser Stelle bereits gezeigt werden. Dazu wir das Paket `pyplot` aus `matplotlib` benötigt. Speziell in Jupyter Notebook (oder in der ipyhton-Shell) sollte man zusätzlich `%matplotlib inline` schreiben. `inline` impliziert, dass der Plot im Notebook selber erscheint. Ein reines `%matplotlib` erzeugt ein neues Fenster mithilfe des jeweils konfigurierten GUI-Backend.

In [6]:
```
# TODO Temperaturwerte plotten
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(c);
```



Die Funktion `plot()` benutzt das Array `C` als Werte für die y-Achse. Als Werte für die x-Achse wurden die Indizes des Arrays `C` verwendet.

9.2 Arrays in NumPy

Wie in 9.1 gesehen, kann man mit der Funktion `array()` ein Array-Objekt aus einer beliebigen Zahlenfolge (z.B. Liste oder Tupel) erzeugen. Bei der Ausgabe mit der Standard-`print()`-Funktion liefert Python eine Darstellung, die der mathematischen Schreibweise für Matrizen sehr ähnelt. Beachten Sie: Zwischen den Zahlen sind keine Kommas.

Mehrdimensionale Arrays lassen sich beispielsweise mit verschachtelten Listen erzeugen:

In [12]:
```
# TODO mehrdimensionale Arrays
a = np.array([[1, 0], [2, 1]])
print(a)
# das zweite Argument gibt den Datentyp vor
a = np.array([[1, 0], [2, 1]], bool)
print(a)
print(a.ndim)
```

```
[[1 0]
 [2 1]]
[[ True False]
 [ True  True]]
2
```

Neben mehrdimensionalen Arrays lassen sich auch "0-dimensionale" erzeugen. Man spricht dann auch von Skalaren.

In [13]:
```
# TODO 0-dimensionales Array
x = np.array(42)
print(x)
np.ndim(x)
```

```
42
```

Out[13]:
```
0
```

Die Dimension lässt sich entweder über das Attribut `ndim` oder über die Methode `np.ndim()` bestimmen. Diese ist gleichbedeutend mit der Anzahl der Achsen.

Spezielle Arrays:

In [16]:
```
# Arrays aus Nullen oder Einsen
print(np.ones((2,3), int))
print(np.zeros((4,5) ))
```

```
[[1 1 1]
 [1 1 1]]
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

9.2.1 Arrays erzeugen mit `arange()`

Der Syntax von `arange()` lautet:

`arange([start=0,] stop[, step=1][, dtype = None])`
wobei:

- *gleichmäßig verteilte* Werte innerhalb der halb-offenen Intervalls `[start, stop)` generiert werden
- als Argumente sowohl `int` als auch `float` Werte übergeben werden können
- mit `dtype` der Type des Arrays bestimmt werden kann, wird nichts angegeben, wird der Type anhand der Eingabewerte ermittelt

Wird `dtype = int` gesetzt, ist die Funktion `arange()` beinahe äquivalent zur built-in Funktion `range()`. Der Unterschied liegt im Rückgabeobjekt. `arange()` liefert ein `ndarry` zurück, während `range()` ein `range`-Objekt, welches ein *Iterator* ist, zurückliefert.

In [19]:
```
# TODO Beispiel zu arange()
a = np.arange(1,7)
print(a)

# im Vergleich dazu range():
x = range(1,7)
print(x)
print(list(x))

# TODO weitere arange-Beispiele
a = np.arange(7.3)
print(a)
```

```
a = np.arange(0.5, 6.1, 0.8)
print(a)

# seltsame Beispiele
print("\nSeltsame Beispiel")
b = np.arange(12.04, 12.84, 0.08)
print("b, ",b)
c = np.arange(0.6, 10.4, 0.71, int)
print("c: ",c)
d = np.arange(0.28, 5, 0.71, int)
print("d: ",d)
help(np.arange)
```

```
[1 2 3 4 5 6]
range(1, 7)
[1, 2, 3, 4, 5, 6]
[0. 1. 2. 3. 4. 5. 6. 7.]
[0.5 1.3 2.1 2.9 3.7 4.5 5.3]

Seltsame Beispiel
b,  [12.04 12.12 12.2  12.28 12.36 12.44 12.52 12.6  12.68 12.76 12.84]
c:  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13]
d:  [0 0 0 0 0 0 0 0]
Help on built-in function arange in module numpy:

arange(...)
    arange([start,] stop[, step,], dtype=None, *, like=None)

    Return evenly spaced values within a given interval.

    ``arange`` can be called with a varying number of positional arguments:

    * ``arange(stop)``: Values are generated within the half-open interval
      ``[0, stop)`` (in other words, the interval including `start` but
      excluding `stop`).
    * ``arange(start, stop)``: Values are generated within the half-open
      interval ``[start, stop)``.
    * ``arange(start, stop, step)`` Values are generated within the half-open
      interval ``[start, stop)``, with spacing between values given by
      ``step``.

    For integer arguments the function is roughly equivalent to the Python
    built-in :py:class:`range`, but returns an ndarray rather than a ``range``
    instance.

    When using a non-integer step, such as 0.1, it is often better to use
    `numpy.linspace`.

    See the Warning sections below for more information.

    Parameters
    ----------
    start : integer or real, optional
        Start of interval.  The interval includes this value.  The default
        start value is 0.
    stop : integer or real
        End of interval.  The interval does not include this value, except
        in some cases where `step` is not an integer and floating point
        round-off affects the length of `out`.
    step : integer or real, optional
        Spacing between values.  For any output `out`, this is the distance
        between two adjacent values, ``out[i+1] - out[i]``.  The default
        step size is 1.  If `step` is specified as a position argument,
        `start` must also be given.
    dtype : dtype, optional
        The type of the output array.  If `dtype` is not given, infer the data
        type from the other input arguments.
    like : array_like, optional
        Reference object to allow the creation of arrays which are not
        NumPy arrays. If an array-like passed in as ``like`` supports
        the ``__array_function__`` protocol, the result will be defined
        by it. In this case, it ensures the creation of an array object
        compatible with that passed in via this argument.
```

```
    .. versionadded:: 1.20.0

Returns
-------
arange : ndarray
    Array of evenly spaced values.

    For floating point arguments, the length of the result is
    ``ceil((stop - start)/step)``.  Because of floating point overflow,
    this rule may result in the last element of `out` being greater
    than `stop`.

Warnings
--------
The length of the output might not be numerically stable.

Another stability issue is due to the internal implementation of
`numpy.arange`.
The actual step value used to populate the array is
``dtype(start + step) - dtype(start)`` and not `step`. Precision loss
can occur here, due to casting or due to using floating points when
`start` is much larger than `step`. This can lead to unexpected
behaviour. For example::

  >>> np.arange(0, 5, 0.5, dtype=int)
  array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
  >>> np.arange(-3, 3, 0.5, dtype=int)
  array([-3, -2, -1,  0,  1,  2,  3,  4,  5,  6,  7,  8])

In such cases, the use of `numpy.linspace` should be preferred.

The built-in :py:class:`range` generates :std:doc:`Python built-in integers
that have arbitrary size <python:c-api/long>`, while `numpy.arange`
produces `numpy.int32` or `numpy.int64` numbers. This may result in
incorrect results for large integer values::

  >>> power = 40
  >>> modulo = 10000
  >>> x1 = [(n ** power) % modulo for n in range(8)]
  >>> x2 = [(n ** power) % modulo for n in np.arange(8)]
  >>> print(x1)
  [0, 1, 7776, 8801, 6176, 625, 6576, 4001]  # correct
  >>> print(x2)
  [0, 1, 7776, 7185, 0, 5969, 4816, 3361]  # incorrect

See Also
--------
numpy.linspace : Evenly spaced numbers with careful handling of endpoints.
numpy.ogrid: Arrays of evenly spaced numbers in N-dimensions.
numpy.mgrid: Grid-shaped arrays of evenly spaced numbers in N-dimensions.
:ref:`how-to-partition`

Examples
--------
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
```

```
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

9.2.2 Arrays erzeugen mit `linspace()`

Der Syntax von `linspace()` lautet:

`linspace(start, stop[, num=50][, endpoint = True][, retstep=False])`
wobei:

- `endpoint` vorgibt, ob der Wert `stop` noch mitausgegeben wird oder nicht. Default = `True`
- `num` die Anzahl der *gleichmäßig verteilten* Werte aus dem Intervall `[start, stop]` vorgibt. Default = `50`
- für `retstep=True` noch zusätzlich der Abstand der zurückgegebenen Werte des Arrays mit zurückgegeben wird. Default = `False`

In [23]:
```
# TODO Beispiele zu linspace()
# optional zur "Verschönerung"
np.set_printoptions(linewidth=65, precision=3)

l = np.linspace(1, 10)
print(l)

l = np.linspace(1, 10, 7)
print(l)

l = np.linspace(1, 10, 7, endpoint = False)
print(l)

samples, spacing = (np.linspace(1, 10, 7, retstep=True))
print(samples, spacing)
```

```
[ 1.     1.184  1.367  1.551  1.735  1.918  2.102  2.286  2.469
  2.653  2.837  3.02   3.204  3.388  3.571  3.755  3.939  4.122
  4.306  4.49   4.673  4.857  5.041  5.224  5.408  5.592  5.776
  5.959  6.143  6.327  6.51   6.694  6.878  7.061  7.245  7.429
  7.612  7.796  7.98   8.163  8.347  8.531  8.714  8.898  9.082
  9.265  9.449  9.633  9.816 10.   ]
[ 1.   2.5  4.   5.5  7.   8.5 10. ]
[1.    2.286 3.571 4.857 6.143 7.429 8.714]
[ 1.   2.5  4.   5.5  7.   8.5 10. ] 1.5
```

Wofür braucht es den Parameter `retstep` überhaupt? Kann man nicht einfach die Differenz zweier benachbarten Werte berechnen?

In [24]:
```
# Beispiel, warum retstep Sinn macht
a, spacing = np.linspace(4, 23, endpoint=False, retstep=True)
print(a[:6])
# Abstände zwischen den ersten 6 Array-Elementen:
for i in range(6):
    print(a[i + 1] - a[i])
print(f"{spacing=}")
```

```
[4.   4.38 4.76 5.14 5.52 5.9 ]
0.3799999999999999
0.3799999999999999
0.3800000000000008
0.379999999999999
0.3800000000000008
0.3799999999999999
spacing=0.38
```

Vergleich von `arange()` und `linspace()`

Laufzeitvergleich:

In [25]:
```
#import laufzeitvergleich
%run laufzeitvergleich.py
```

```
Laufzeit für Python range()...: 0.12517261505126953
Laufzeit für NumPy  arange()..: 0.0029914379119873047
Laufzeit für NumPy  linspace(): 0.006981849670410156
arange()   ist   42 mal schneller als range()
linspace() ist   18 mal schneller als range()
<Figure size 640x480 with 0 Axes>
```

Attribute (*flags*) geben Informationen über das *Memory Layout* des Arrays:

In [26]:
```
# Attributvergleich
arr = np.arange(20, 30)
linsp = np.linspace(20, 30, 10, endpoint=False)

print(arr)
print(linsp)

print("Flags von arange():\n", arr.flags)
print("Flags von linspace():\n", linsp.flags)
```

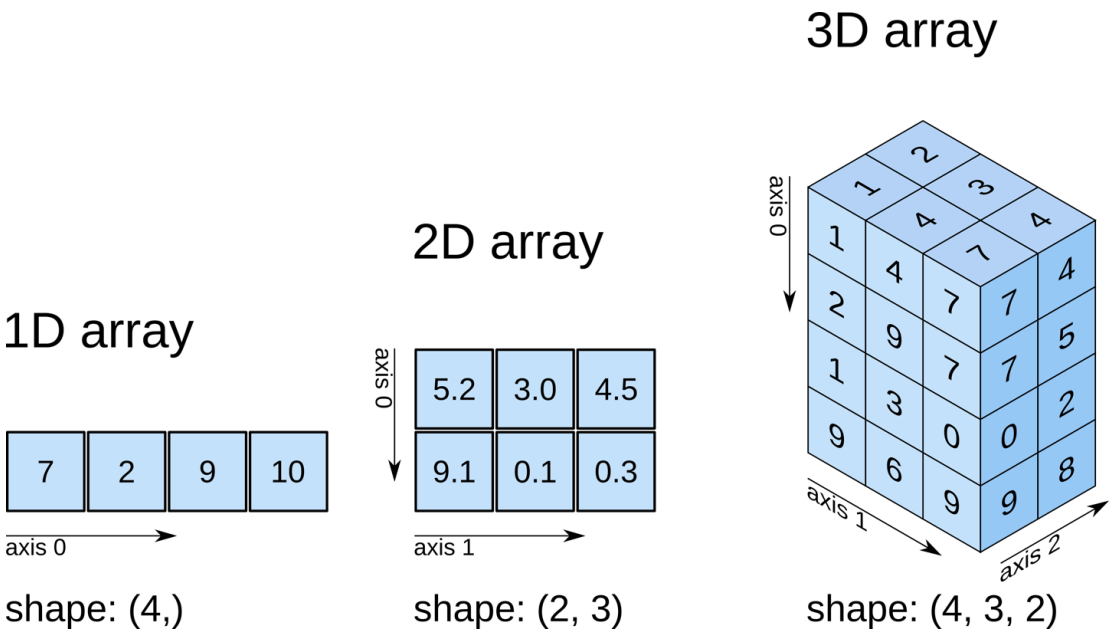
```
[20 21 22 23 24 25 26 27 28 29]
[20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]
Flags von arange():
   C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False

Flags von linspace():
   C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

Interessant ist hierbei das Attribut `OWNDATA` (s. später in 9.2.4):

> The array owns the memory it uses (True) or borrows it from another object (False).

9.2.3 Form eines Arrays ermitteln

Neben der Methode `np.ndim()` gibt es zur Bestimmung der Größe bzw. Gestalt eines Arrays die Methoden `shape()` und `size()`. Die erste Methode liefert ein Tupel mit der Anzahl der Elemente pro Achse (Dimension). Es gibt auch eine äquivalente Array-Property `shape`. Die zweite gibt die absolute Anzahl der Elemente eines Arrays zurück.



Quelle: F. Neumann. "Introduction to numpy and matplotlib". TU Berlin. fneum.github.io

In [35]:
```
# Beispiel zu shape()
x = np.array([[67, 63, 87],
              [77, 69, 59],
              [85, 87, 99],
              [79, 72, 71],
              [63, 89, 93],
              [68, 92, 78]])

y = np.array([[[111, 112], [121, 122]],
              [[211, 212], [221, 222]],
              [[311, 312], [321, 322]]])

z = np.array([1, 2, 3, 4, 5])

# TODO Form ermitteln
print(np.shape(x)) # Methodenaufruf
print(x.shape) # Property
print(np.size(x))

print(np.shape(y))
print(np.size(y))

print(np.shape(z))
```

```
(6, 3)
(6, 3)
18
(3, 2, 2)
12
(5,)
```

9.2.4 Form eines Arrays verändern

`numpy` Arrays besitzen Methoden, die eine *Sicht* (*view*) mit neuer Form kontruieren: `reshape()` und `ravel()`. Dabei ist die neue Form kein eigenes unabhängiges Objekt, sonder eine *Sicht*, die mit dem Original-Array verbunden bleibt. Wenn das Original-Array verändert wird, ändert sich auch die *Sicht*.

- Beim Aufruf von `reshape()` werden natürliche Zahlen übergeben, die die Form der gewünschten Sicht beschreiben

- `ravel()` liefert eine *verflachte* Sicht, ein eindimensionales Array, in dem alle Elemente des Original-Arrays hintereinander geschrieben sind.

In [45]:
```
# TODO Beispiel zu reshape() und ravel()
a = np.arange(12)
print(a)

b = a.reshape(3, 4)
print(b)

c = b.ravel()
print(c)

del a
print(c)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

In [ ]:
```
```

Mit der Methode `resize()` wird das Array (und dessen Größe) verändert. Die Argumente sind Zahlen, die die gewünschte Form beschreiben. Wenn die neue Form größer ist, werden Nullen ergänzt. Allerdings wird empfohlen, die Methode über `np.resize(array, (newsize))`, anstatt es über das `ndarray` direkt über `ndarray.resize((newsize))` aufzurufen. Vor allem bei über `linspace()` erzeugte Arrays kann es sonst zu Fehlermeldungen führen. Hierbei wird solange mit wiederholten Kopien des Arrays aufgefüllt, bis die gewünschte Größe erreicht ist (s. auch Dokumentation)

In [52]:
```
# TODO Array verändern
k = np.linspace(1, 20, 19, endpoint = False)
print(k)
#k.resize(10, 10)
k = np.resize(k, (10, 10))
print(k)

l = np.arange(1, 20)
```

```
print(l)
#l.resize(10,10)
l = np.resize(l, (10, 10))
print(l)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16.
 17. 18. 19.]
[[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
 [11. 12. 13. 14. 15. 16. 17. 18. 19.  1.]
 [ 2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
 [12. 13. 14. 15. 16. 17. 18. 19.  1.  2.]
 [ 3.  4.  5.  6.  7.  8.  9. 10. 11. 12.]
 [13. 14. 15. 16. 17. 18. 19.  1.  2.  3.]
 [ 4.  5.  6.  7.  8.  9. 10. 11. 12. 13.]
 [14. 15. 16. 17. 18. 19.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9. 10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.  1.  2.  3.  4.  5.]]
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[[ 1  2  3  4  5  6  7  8  9 10]
 [11 12 13 14 15 16 17 18 19  1]
 [ 2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19  1  2]
 [ 3  4  5  6  7  8  9 10 11 12]
 [13 14 15 16 17 18 19  1  2  3]
 [ 4  5  6  7  8  9 10 11 12 13]
 [14 15 16 17 18 19  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19  1  2  3  4  5]]
```

9.2.5 Indizierung und Slicing

Um auf einzelne Elemente zuzugreifen, verwendet man wie bei *Sequenzen* den `[]`-Operator:

In [53]:
```
# Zugriff auf Elemente eines Arrays
f = np.array([1, 7, 2, 3, 5, 8, 13, 21])
print(f[0]) # erstes Element
print(f[1]) # zweites Element
print(f[-1])# letztes Element
```

```
1
7
21
```

In [54]:
```
# Mehrdimensionale Arrays indizieren
y = np.array([[[111, 112], [121, 122]],
              [[211, 212], [221, 222]],
              [[311, 312], [321, 322]]])

print(y[0][1][0])   # wie bei verschachtelten Listen
print(y[0, 1, 0])   # funktioniert genauso
```

```
121
121
```

Auch das *Slicing* funktioniert ähnlich wie bei normalen Python-Sequenzen nach dem Syntax `[start:stop:step]`. **Achtung**: Während bei Listen und Tupel neue Objekte erzeugt werden, generiert der Teilsbereichtoperator bei NumPy nur eine *Sicht* auf das Original-Array. Um ein Array zu kopieren, benötigt man die Methode `copy()`.

In [58]:
```
# Einige Beispiele
a = np.arange(9)
print(a)
print(a[1:4])
print(a[:4])
print(a[4:])
print(a[::2])
b = a[:]    # Sicht auf a
print(b)
b[0] = 99
print(a)

# TODO (Teil-)Array kopieren
c = a.copy()[4:]
print(c)
c[0] = 49
print(c)
print(a)
```

```
[0 1 2 3 4 5 6 7 8]
[1 2 3]
[0 1 2 3]
[4 5 6 7 8]
[0 2 4 6 8]
[0 1 2 3 4 5 6 7 8]
[99  1  2  3  4  5  6  7  8]
[4 5 6 7 8]
[49  5  6  7  8]
[99  1  2  3  4  5  6  7  8]
```

In [56]:
```
# Gleiche Operationen bei einer Liste
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8]
print(lst)
print(lst[1:4])
print(lst[:4])
print(lst[4:])
print(lst[::2])
lst2 = lst[:]   # flache Kopie
lst2[0] = 99
print(lst)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3]
[0, 1, 2, 3]
[4, 5, 6, 7, 8]
[0, 2, 4, 6, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Im Zweifel kann man mithilfe der Methode `np.may_share_memory()` prüfen, ob zwei Arrays auf den gleichen Speicherbereich zugreifen.

In [59]:
```
# Prüfe Speicherbereich
print(np.may_share_memory(a, b))
```

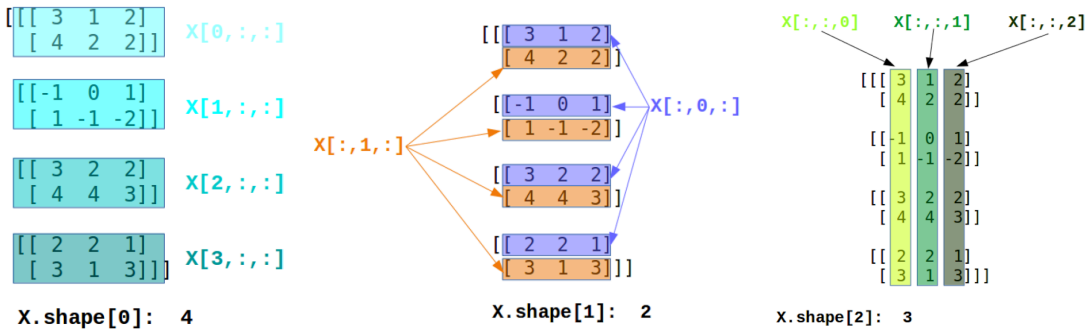
```
print(np.may_share_memory(a, c))
```

```
True
False
```

Bei dreidimensionalen Arrays ist der Zugriff etwas schwerer vorstellbar. Aus diesem Grund soll folgendes Beispiel genauer betrachtet werden

In [60]:
```
# dreidimensionales Array
X = np.array([[[3, 1, 2],
               [4, 2, 2]],

              [[-1, 0, 1],
               [1, -1, -2]],

               [[3, 2, 2],
                [4, 4, 3]],

                [[2, 2, 1],
                 [3, 1, 3]]])

# zuerst selber überlegen
print(X.shape)
```

```
(4, 2, 3)
```



Quelle: B. Klein, „Numerisches Python“, Carl Hanser Verlag GmbH & Co. KG, 2023, S. 30

9.2.6 Darstellung von Matrizen und Vektoren

Zur Darstellung von Matrizen und Vektoren werden zweidimensionale Arrays genutzt.

Beispiel zu Vektoren:

In [62]:
```
# TODO Vektoren mit numpy realisieren
v1 = np.array([[1, 2, 3]])    # Zeilenvektor n x 1
v2 = np.array([[1], [2], [3]])# Spaltenvektor 1 x n

print(v1, np.shape(v1))
print(v2, np.shape(v2))
```

```
[[1 2 3]] (1, 3)
[[1]
 [2]
 [3]] (3, 1)
```

Für Matrizen bietet sich die Kombination aus der Erstellung einer Zahlenfolge und der Methode `reshape()` an:

In [63]:
```
# TODO Möglichkeit, Matrix darzustellen
m = np.arange(9)
matrix = m.reshape(3, 3)
print(matrix)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

Zur Erzeugung einer Einheitsmatrix gibt es zwei Möglichkeiten:

1. `eye(n[, m[, k[, dtype]]])` : `n` : Anzahl der 1 er auf der Hauptdiagonalen; `m` : Optional. Anzahl der Spalten, falls Matrix nicht quadratisch; `k` : Optional. Anzahl der 0 er Spalten *vor* der Diagonalen aus 1.
2. `identity(n[, dtype])` : liefert quadratische Einheitsmatrix der Form n x n.

In [ ]:
```
# TODO Einheitsmatrix
i = np.identity(3, dtype = int)
print(i)

e =np.eye(3, 7, 1, dtype = int)
print(e)
```

Die Methode `transpose()` liefert eine transponierte Darstellung als *Sicht*. Arrays besitzen alternativ auch das Attribut `T`, das ebenfalls eine *Sicht* auf die transponierte Darstellung enthält.

In [ ]:
```
# TODO (nach den Pfingsferien) mit Beispielen von oben


# transponierter Vektor
```