Überblick u. Verständnis von Python

- functions aus einer Bibliothek laden:

  1) <u>import</u> math (lade math-Bibl.)

  ⇒ nutzbar ist: math.exp(1) $\hat{=}$ e^1

  2) <u>from</u> math <u>import</u> *

  ⇒ Aufruf: exp(1) $\hat{=}$ e^1

- Inhalt des geladenen Moduls ansehen:

  mlist = dir(math) ⇒ Typ <u>list</u>

- Was ist eine Anweisung:

  $$\boxed{\underbrace{\text{var}}_{\text{Ergebnis-Var.}} = \text{Ausdruck aus Konst., Funkt. u. Variablen}}$$

  <u>Wirkung</u>: 1) berechne Typ u. Wert des Ausdrucks rechts

  2) erzeuge od. überschreibe die Variable var links mit Wert u. Typ von rechts

  <u>Bsp</u>:

  ```
  i = 2
  i = i + 1    # i hat jetzt Wert 3
  ```

- Komplexere Anweisungen in Python haben die Struktur:

  Anweisungskopf :

  Anweisung 1

  Anw. 2

  ...

  } Anw'block

  Einrückung (Leerzeichen)

<u>Bsp</u> 1) <u>if</u> <log. Bedingung> :

  Anw. block

2) <u>for</u> var <u>in</u> <sequence>:

$\boxed{\text{Anw. block}}$

<sequence> kann sein:
- range (a, e) <u>oder</u> range (a, e, s)
- list, z.B. [1, 3, 7, 9]
- string, z.B. 'Hello'

- <u>Datentypen sind (wichtige):</u>
  - int — ganzzahlig, Bsp: $i = 3+5$
  - float — (floating point number) Bsp: $x = 32.25$
  - str — Zeichenkette (String) Bsp: s = 'Hello'
  - bool — logische Var., Bsp. $A = 3 < 1$
  - list — Bsp: L = [2, 3.5, 'otto']

    Zugriff: L[0] → 2

    L[2] → 'otto'
  - tuple — Tupel

    <u>Bsp</u>: $(x, y) = (2, 3) \Longrightarrow \begin{cases} x = 2 \\ y = 3 \end{cases}$

    P = (x, y) # P ist Tupel (entspricht Punkt)

print - Anweisung

- print (string1, var1, ...)

Bsp: print ("das Quadrat von x = ", x , "ist y=", x*x)

print ("\n") gibt Leerzeile aus

print ("===....=") gibt Trennlinie aus

- formatierte Ausgabe:

print ("<string1>%<format1> <string2>%<format 2>" % (var1, var2))

Bsp:

print ("Quadrat von x=%8.2f ␣␣ist y=%10.4f" % (x, x*x))

- Beispiele für Format-Beschreibungen

f - Format:

- allgemein: % m.s f

  m = Maskenbreite
  s = Stellenanzahl nach dem Punkt

es wird rechtsbündig in die Maske (aus m Zeichen) geschrieben

- Bsp: %6.2f , Var. x = - 2.7531

⇒ Ausgabe-Maske =

c_1	c_2	c_3	c_4	c_5	c_6
␣	-	2	.	7	5

← Zeichen der Maske

e - Format: (Exponenten-Format)

- allgemein: % m.s e

  m = Maskenbreite
  s = Stellenanzahl nach dem Punkt

- Bsp: %11.2e , Var. $x = -0.003627$ — gerundet auf: 63

⇒ Ausgabe-Maske = ␣␣␣-3.63e-03 ($c_1\ c_2\ c_3\ c_4\ c_5\ c_6\ c_7\ c_8\ c_9\ c_{10}\ c_{11}$)

i - Format

allgemein: %mi

m = Maskenbreite

- Bsp: %5i , Var. $k = -47$

⇒ Ausgabe-Maske = ␣␣-47 ($c_1\ c_2\ c_3\ c_4\ c_5$)

Anwendung: Tabelle der Qudrat- u. Kubik-Zahlen

Ziel: Tabelle:

```
k = 2    k^2 = 4      k^3 = 8
 :   ...
k = 10   k^2 = 100    k^3 = 1000
```

```
for k in range(2, 11):
    k2 = k*k
    k3 = k2*k
    print("k=%?  k^2=%?  k^3=%?" % (k,k2,k3))
```

input - Anweisung

<Eingabe-String> = input("<Eingabe-Aufforderung>")

Bsp:
```
xstring = input("Eingabe von x = ")
type(xstring)   ⇒ Typ ist str
```

```
# Umwandlung in Variable x (z.B. Typ float)
    x = float(xstring)

# verkürzte Variante
    x = float( input("Eingabe von x = "))
```

Programmierung rekursiv definierter Folgen

Bsp: Partialsumme $s_n := \sum_{k=0}^{n} a_k$

also: $s_0 = a_0$, $s_1 = s_0 + a_1$, $s_2 = s_1 + a_2$, ...

$\Rightarrow$ Rekursion: $\boxed{s_k = s_{k-1} + a_k}$ $\forall\, k = 1, 2, \ldots$

im Programm: $\underbrace{s}_{\text{neuer s-Wert}} = \underbrace{s}_{\text{alter s-Wert}} + \underbrace{a}_{\text{neuer a-Wert}}$

Programm-Code:

```
a = <Wert von a_0>          # Anfangswert für a
s = a                       # Anfangswert für s

for k in range(1, n+1):
    a = <Formel für neuen a-Wert a_k>
    s = s + a
    print("k = %3i  s_k = %16.12f" %(k,s))
```

Bsp: $\exp(x) \approx \sum_{k=0}^{n} \underbrace{\frac{1}{k!} x^k}_{= a_k}$, $a_0 = \frac{1}{0!} x^0 = 1$

Rekursion von a_k: $a_k = \frac{1}{k!} x^k = \frac{1}{(k-1)! \cdot k} x^{k-1} \cdot x$

$$\Rightarrow \quad a_k = a_{k-1} \cdot \frac{x}{k}$$

im Programm: $\underbrace{a}_{\text{neu}} = \underbrace{a}_{\text{alt}} * x / k$

weitere Bsp'le:

- Newton-Verfahren: $x_0 = \langle \text{geg. Startwert} \rangle$

$$\underbrace{x_{k+1}}_{\substack{\text{neuer} \\ x\text{-Wert}}} = \underbrace{x_k}_{\substack{\text{alter} \\ x\text{-Wert}}} - f(\underbrace{x_k}) / f'(\underbrace{x_k})$$

- Fixpunkt-Iteration:

$$x_{k+1} = \langle \text{Formel von } x_k \rangle$$

Das if-Kommando zur Ablaufsteuerung

Variante 1

if <logische Bedingung> :

$$\boxed{\text{Anweisungsblock}}$$

Variante 2

if <log. Bed.> :

Anw. block 1

else :

Anw. block 2

Variante 3

if <log. Bed. 1> :

Anw. block 1

elif <log. Bed. 2> :

Anw. block 2

elif <log. Bed. 3> :

Anw. block 3

...

else :

Anw. block

Bem: Var. 2 und 3 liefern eine vollständige Fallunterscheidung

Die while - Schleife

Syntax: **while** <logische Bedingung>:

Anweisungsblock

Wirkung: solange die <log. Bed.> wahr ist, wird der Anw.block ausgeführt

Definition eigener Funktionen

Syntax einer Funktion

def <Funktionsname>(<Liste der Eingabeparameter>):

"""

Dokumentationszeilen

"""

Anweisungsblock zur Bestimmung des Ergebnisses <fwert> der Funktion

return <fwert>

Bem: hat man mehrere Ergebniswerte, so ist <fwert> ein Tupel, z.B.

return (y, error)

Funktion mit Schlüsselwortparametern

Syntax :

```
def <fname> (<pos-par1>, ..., <key-par1>=<w1>, ...)
```

Positions-Parameter

Schlüsselwort-Parameter

Standartwert (default)

Bsp :

```
def f1 (x, y, z=0, p=2.5, out=False)
```

2 Pos. par.

3 Schlüsselwort-Param.

mögliche Aufrufe sind :

```
f1 (x, y)    # z=0, p=2.5, out=False

f1 (x1, y2, p=3.1)  # z=0, out=False
```

x=x1

y=y2

Das Paket Numpy ("Numerical Python")

$\boxed{\texttt{import numpy as np}}$ laden unter dem Namen np

Erzeugung von Vektoren (arrays)

1) aus einer Liste:

x = np.array([Liste der Elemente])

Bsp: x = np.array([2, 3, 7, 5])

2) über arange:

x = np.arange(start, stop, step = 1)

⇒ gleichmäßig verteilte Werte im Intervall [start, stop) mit Schrittweite step

3) über linspace:

x = np.linspace(start, stop, num = 50, endpoint = True, retstep = False)

⇒ liefert N = num gleichmäßig verteilte Werte im Intervall [start, stop] oder in [start, stop) wenn endpoint = False

wenn retstep = True, dann wird die Schrittweite h zusätzlich zurückgegeben:

x, h = np.linspace(..., retstep = True)

4.) über zeros oder ones

Bsp.: $z = $ np.zeros (4)

$\Rightarrow$ $z = [0., 0., 0., 0.]$

$v = $ np.ones (3)

$\Rightarrow$ $v = [1., 1., 1.]$

Bem.: $M = $ np.zeros $((2,3))$

$\Rightarrow$ $M = $ Nullmatrix mit 2 Zeilen, 3 Spalten

Indizierung

- $x[k]$ = k-tes Element von x, wobei $k = 0, 1, \dots, n-1$

mit $\boxed{n = \mathrm{len}(x)}$ Länge des Vektors x

- $M[i,j]$ = Element von Matrix M in Zeile i und Spalte j

- Teilvektor von x:

$v = x[1:3]$

liefert: $v = [\,x[1]\,,\ x[2]\,]$

alle Indices k mit $1 \leq k < 3$

Funktionen angewendet auf np-Vektoren

1) algebraische Operationen

geg.: np-Vektoren x, y gleicher Länge

- dann sind definiert: $x+y$, $x-y$, $a*x$ (a = Zahl)
- $x*y$ ist komponentenweise def.
- ebenso auch: $x*x$, $x**3$, x/y

$\Rightarrow$ eine Funktion wie z.B.

$$f(x) = x**3 - 2*x*x + 5*x - 3$$

ist komponentenweise definiert

2) math. Funktionen wie sin, cos, exp, log

Bsp.: $y = np.sin(x)$

ist komponentenweise definiert, d.h.

$$y[k] = \sin(x[k]), \quad 0 \le k < n$$

$$n = len(x)$$

Das Paket Matplotlib (plotten von Kurven)

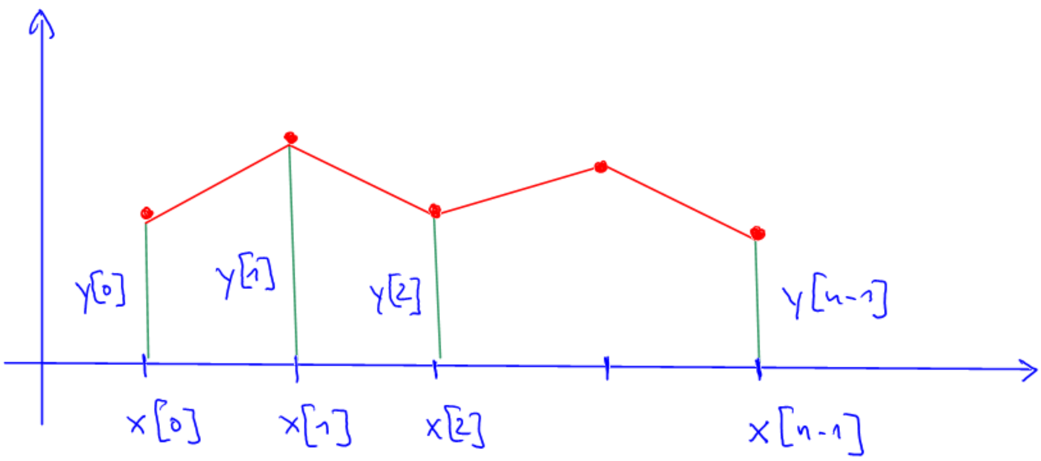
```
import matplotlib.pyplot as plt
```

laden unter dem Namen plt

das Prinzip: es werden Kurvenpunkte

$$(x[k], y[k]), \quad 0 \leq k < n,$$

gezeichnet und durch einen Polygonzug verbunden



plot-Befehl dazu:

$$\texttt{plt.plot}(x, y, \underbrace{\langle\text{format}\rangle}_{\text{codiert das Format der Kurve}})$$

$$\langle\text{format}\rangle = {}'\underbrace{[\text{marker}]}_{\text{Marker-symbol}}\underbrace{[\text{line}]}_{\text{Linien-art}}\underbrace{[\text{color}]}_{\text{Farbe}}{}'$$

Bsp: $\langle\text{format}\rangle = $ 'o-r'

Marker o dicker Punkt
Linie - durchgezogen
Farbe r rot

weitere plot-Befehle

plt.xlabel('<x_string>') Beschriftung der x-Achse mit <x_string>

plt.ylabel('<y-string>') Beschriftung der y-Achse mit <y-string>

plt.grid() schaltet die Gitterlinien an

plt.ylim([ymin, ymax]) setzt die Grenzen der y-Achse auf ymin und ymax

plt.xlim([xmin, xmax]) analog für x-Achse

plt.title('<string>') erzeugt die Bildüberschrift <string>

plt.show() erzeugt Bild mit allen plot-Befehlen

legend – Erklärungen zu Kurven

- geg.: ~ x = Stützstellenvektor (np-Vektor, Länge n)
  ~ y1 = np-Vektor der y-Koordinaten zu Kurve 1
  ~ y2 = — " — Kurve 2
  ...

- plot-Befehle:

plt.plot (x, y1 [, <format_1>], label = '<Text_1>')

plt.plot (x, y2 [, <format_2>], label = '<Text_2>')

... u.s.w.

wobei '<Text_k>' = Erklärtext zu Kurve k
k = 1, 2, ...

plt.legend () → erzeugt Subfenster mit der Zuordnung
<format Kurve k> zu '<Text_k>'

plt.show () → erzeugt Bild zu allen vorherigen plot-Befehlen

Logarithmische Achsen-Skalierung in plots

- sind die Werte x[k] und/oder y[k] der Koordinaten der plot-Punkte sehr klein oder sehr groß, dann ist eine logarithmische Achsen-Skalierung angebracht

- die plot-Befehle hierzu:

```
plt.loglog(x, y, ...)    # x- und y-Achse logarithm. skaliert

plt.semilogx(x, y, ...)  # nur x-Achse log. skaliert

plt.semilogy(x, y, ...)  # nur y-Achse log. skaliert
```

- Bsp: siehe Skript 'np_logplot.py'

Lineare Algebra

- seien x, y, a, b, ... np-Vektoren der Länge n und A, B, C ... np-Matrizen mit Format n×n
- Skalarprodukt von Vektoren :

  s = np.dot(x,y)

- Matrix mal Vektor : b = np.dot(A,x)

- für anspruchsvollere Aufgaben der linearen Algebra lädt man das Paket 'numpy.linalg' mittels :

```
import numpy.linalg as la
```

damit kann man realisieren :

1) Vektor-Norm zu Vektor x :

```
la.norm(x, ord=<p>)
```

Kurzform:

```
la.norm(x,<p>)
```

<p> codiert die Art der Norm, z.B.

$$\langle p\rangle = 1 \longrightarrow \|x\|_1 = \sum_{k=0}^{n-1} |x[k]|$$

$$\langle p\rangle = 2 \longrightarrow \|x\|_2 = \left(\sum_{k=0}^{n-1} |x[k]|^2\right)^{1/2}$$

$$\langle p\rangle = \text{np.inf} \longrightarrow \|x\|_\infty = \max_{0\le k<n} |x[k]|$$

2) Matrix-Norm zu A

analoger Aufruf: la.norm(A, <p>)

3) Konditionszahl einer Matrix A

c = la.cond(A, <p>)

berechnet die Zahl $c = \text{cond}(A) := \|A\| \cdot \|A^{-1}\|$

4) inverse Matrix $B = A^{-1}$

B = la.inv(A)

5) Determinante $d = \det(A)$:

$\boxed{d = \text{la.det}(A)}$

6) Lösung des Gleichungssystems $Ax = b$

x = la.solve(A, b)

7) Eigenwerte u. Eigenvektoren von $A \in \mathbb{R}^{n \times n}$

- sei λ_k EW zu EV $v_k \in \mathbb{R}^n$ mit $0 \le k < n$
- dann liefert der Befehl

$\boxed{(ew, EV) = \text{la.eig}(A)}$

$\lambda_k = ew[k]$, $v_k = EV[:, k] =$ k-te Spalte der Matrix EV