

## 2 Numerisches Python I

**Numerisches Programmieren** ist auch bekannt unter dem eher irreführenden Namen „wissenschaftliches Programmieren“. Unter numerischem Programmieren versteht man das Gebiet der Informatik und Mathematik, bei dem es um Approximationsalgorithmen geht, d.h. numerische Approximation von mathematischen Problemen oder numerischer Analysis.

Wir werden in dieser Einführung folgende Module kennenlernen:

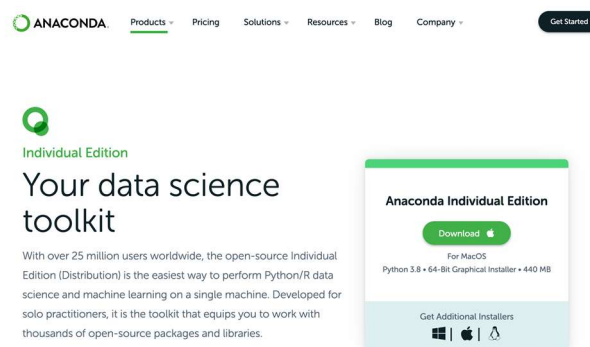
**Matplotlib** ist ein externes Python Modul, welches sich für das Plotten von Daten spezialisiert. NumPy lässt sich auch mit PyQt5 kombinieren und somit die Plots in einem GUI darstellen.

**NumPy** ist ein externes Open-Source Modul für Python, welches mathematische Grundfunktionalität zur Verfügung stellt. NumPy und SciPy beinhalten Funktionalität, welche mit kommerzieller Software wie Matlab vergleichbar ist. NumPy steht für „Numeric Python“ und enthält vor allem Algorithmen zur Manipulation von sehr grossen Arrays und Matrizen mit numerischen Werten.

### 2.1 Installation von Conda

Bei Anaconda sind zahlreiche Python Module bereits installiert. Leider fehlen die wichtigsten Geo-spezifischen Module, welche naträchlich installiert werden müssen. Der «geospatial stack» ist jedoch nicht ganz einfach zu installieren. In diesem Kapitel lernen wir wie wir die geo-Module installieren und im Jupyter Notebook verwenden können. Dabei lernen wir auch die «virtuellen Umgebungen» kennen, um Module einfach zu verwalten.

Anaconda kann hier heruntergeladen werden: <https://www.anaconda.com/download>



Unter Windows sollte für die Verwendung mit Visual Studio Code noch Powershell-Shortcut installiert werden (in Anaconda Prompt ausführen)

```
conda install powershell_shortcut
```

## 2.2 Einführung in Matplotlib

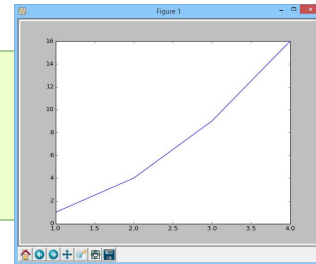
### 2.2.1 Grundlagen Plotten

Matplotlib ( <https://matplotlib.org/> ) ist eine sehr umfassende Bibliothek zum Plotten von Daten.

Wenn wir beispielsweise Punkte haben und diese plotten wollen, können wir das ganz einfach mit der `plot` Funktion tun:

```
import matplotlib.pyplot as plt

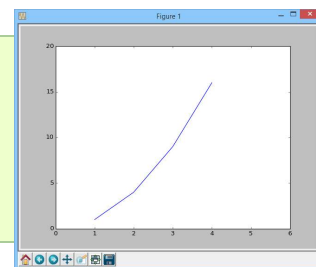
plt.plot([1,2,3,4], [1,4,9,16])
plt.show()
```



Wir können mit `axis()` noch den Bereich welcher dargestellt wird angeben:

```
import matplotlib.pyplot as plt

plt.plot([1,2,3,4], [1,4,9,16])
plt.axis([0,6,0,20])
plt.show()
```

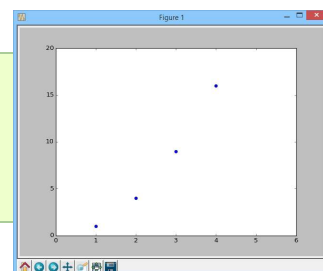


Ähnlich wie in Matlab kann auch noch angegeben werden wie das Resultat dargestellt wird. Standardmässig wird mit einer blauen Linie geplottet (Option "b-")

Sollen nun Punkte dargestellt werden so kann die Option "bo" für blaue Punkte oder "ro" für rote Punkte verwendet werden:

```
import matplotlib.pyplot as plt

plt.plot([1,2,3,4], [1,4,9,16], "bo")
plt.axis([0,6,0,20])
plt.show()
```



Die folgenden Farbabkürzungen sind möglich, alternativ kann auch der Parameter «color» verwendet werden, um eine Farbe zu definieren.

'b'	blau
'g'	grün
'r'	rot
'c'	cyan
'm'	magenta
'y'	gelb
'k'	schwarz
'w'	weiss

Der Linienstil resp. die Marker können folgendermassen gesteuert werden:

' - '	(Bindestrich) durchgezogene Linie
' - - '	(zwei Bindestriche) gestrichelte Linie
' - . '	Strichpunkt-Linie
' : '	punktierte Linie
' . '	Punkt-Marker
' , '	Pixel-Marker
' o '	Kreis-Marker
' v '	Dreiecks-Marker, Spitze nach unten
' ^ '	Dreiecks-Marker, Spitze nach oben
' < '	Dreiecks-Marker, Spitze nach links
' > '	Dreiecks-Marker, Spitze nach rechts
' 1 '	tri-runter-Marker
' 2 '	tri-hoch-Marker
' 3 '	tri-links Marker
' 4 '	tri-rechts Marker
' s '	quadratischer Marker
' p '	fünfeckiger Marker
' * '	Stern-Marker
' h '	Sechseck-Marker1
' H '	Sechseck-Marker2
' + '	Plus-Marker
' x '	x-Marker
' D '	rautenförmiger Marker
' d '	dünner rautenförmiger Marker
'   '	Marker in Form einer vertikalen Linie
' _ '	Marker in Form einer horizontalen Linie

Eine Beschriftung der Achse kann mit den Funktionen `xlabel("name")` und `ylabel("name")` eingefügt werden:

Mit `grid(True)` ist es zudem möglich ein Koordinatengitter darzustellen.

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], "b^")
plt.axis([0,6,0,20])
plt.xlabel("Äpfel")
plt.ylabel("Birnen")
plt.grid(True)

plt.show()
```

- Wir können übrigens auch direkt im Jupyter Lab plotten
- Visual Studio kann auch mit jupyter notebooks umgehen!

## 2.3 Einführung zu Numpy

Dies ist nur ein ganz kleiner Einstieg in NumPy, denn diese ist eine sehr grosse Bibliothek. Die Dokumentation von NumPy ist auf <http://docs.scipy.org/doc/> zu finden. Die Numpy Referenz (PDF) umfasst ca. 1500 Seiten. Ein guter Start in NumPy bietet sich auf: <http://docs.scipy.org/doc/numpy/user/basics.html> an.

Nach der Installation des NumPy Moduls können wir das Modul wie gewohnt importieren:

```
import numpy
```

Da numpy relativ oft geschrieben werden muss, bevorzugen viele Leute folgenden import:

```
import numpy as np
```

Damit können wir anstelle von „numpy“ ganz einfach „np“ schreiben. Wir verwenden auch diese Schreibweise.

Eine der wichtigsten Klassen von numpy ist die Array-Klasse. Diese ist sehr ähnlich wie eine Liste, jedoch darf darin nur genau ein Datentyp pro Array vorkommen. Es gibt in numpy mehr Datentypen als bei Standard-Python, jedoch sind diese Hardware-näher.

<b>np.bool_</b>	Boolean (True oder False)
<b>np.int_</b>	Standard integer Typ (32- oder 64-bit je nach Python)
<b>np.int8</b>	Byte (-128 to 127)
<b>np.int16</b>	Integer (-32768 to 32767)
<b>np.int32</b>	Integer (-2147483648 to 2147483647)
<b>np.int64</b>	Integer (-9223372036854775808 to 9223372036854775807)
<b>np.uint8</b>	Unsigned integer (0 to 255)
<b>np.uint16</b>	Unsigned integer (0 to 65535)
<b>np.uint32</b>	Unsigned integer (0 to 4294967295)
<b>np.uint64</b>	Unsigned integer (0 to 18446744073709551615)
<b>np.float_</b>	Abkürzung für float64.
<b>np.float16</b>	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
<b>np.float32</b>	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
<b>np.float64</b>	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
<b>np.complex_</b>	Abkürzung für complex128.
<b>np.complex64</b>	Komplexe zahl, bestehend aus zwei 32-bit floats
<b>np.complex128</b>	Komplexe zahl, bestehend aus zwei 64-bit floats

```
a = np.array([2,4,6,8], dtype=np.float64)
a = np.array([2,4,6,7])
```

Der Datentyp kann mit dtype abgefragt werden:

```
a.dtype
```

Ein Array kann auch in ein Array eines anderen Typs konvertiert werden:

```
b = a.astype(np.float64)
```

Auf die Elemente kann wie bei einer Liste zugegriffen werden, z.B.

```
a[0]  
a[2:3]
```

Es ist auch möglich mehrdimensionale Arrays zu erstellen, zum Beispiel ein zweidimensionales Array würde folgendermassen konstruiert:

```
a = np.array([[1,2,3], [4,5,6]], np.float64)
```

Bei mehrdimensionalen Arrays funktioniert das „Slicing“ sehr ähnlich, nur müssen wir die weiteren Dimensionen beachten:

Nehmen wir vorheriges Array, so gilt folgendes:

```
a[:]
```

hat den Wert

```
array([[ 1.,  2.,  3.],  
       [ 4.,  5.,  6.]])
```

also das gesamte Array

```
a[0,:]
```

hat den Wert `array([ 1., 2., 3.])`

```
a[1,:]
```

hat den Wert `array([ 4., 5., 6.])`

```
a[:,2]
```

hat den Wert `array([ 3., 6.])`

Wenn wir die Dimension abfragen wollen, können wir das mit dem Attribut „shape“. Dies liefert ein Tuple mit der Grösse des Arrays, also in unserem Beispiel wäre

```
a.shape
```

(2, 3) also ein 2x3 grosses Array. (2 Zeilen, 3 Spalten)

Wir können mit „in“ relativ einfach überprüfen, ob ein bestimmter Wert im Array vorkommt:

```
5 in a
```

liefert `True` und

```
20 in a
```

liefert `False`

Es gibt noch weitere Möglichkeiten ein Array zu erstellen:

`np.zeros(m,n)` erstellt ein  $m \times n$  grosses Array welches mit 0 gefüllt ist.

`np.ones(m,n)` erstellt ein  $m \times n$  grosses Array welches mit 1 gefüllt ist.

`np.arange(...)` erstellt ein Array mit fortlaufenden Werten. Es gibt dabei verschiedene Parameter:

`np.arange(n)`: Erstellt ein Array mit Werten im Bereich  $[0,n[$

`np.arange(a,b)`: Erstellt ein Array mit Werten im Bereich  $[a, b[$

`np.arange(a,b,s)`: Erstellt ein Array mit Werten im Bereich  $[a,b[$  mit Schrittweite  $s$ .

Mit `np.linspace(...)` können Arrays mit einem Bereich und einer bestimmten Anzahl Elemente erstellt werden.

`np.linspace(a,b,n)` erstellt ein Array im Bereich  $[a,b]$  mit  $n$  Elementen. zum Beispiel:

`np.linspace(0,1,11)` erstellt zum Beispiel

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
```

Der Parameter `dtype=Typ` kann bei diesen Funktionen angehängt werden, falls der Datentyp angegeben werden soll.

Falls ein Array mit zufälligen Werten erstellt werden soll, geht das mit

`np.random.random(n)`, welches ein Array mit  $n$  zufälligen Werten im Bereich  $[0, 1[$  erstellt.

## 2.4 Mathematische Funktionen und Konstanten in Numpy

Numpy hat zahlreiche Mathematische Funktionen, welche mit Arrays funktionieren.

### 2.4.1 Trigonometrische Funktionen

Die wichtigsten trigonometrische Funktionen von numpy sind in der folgenden Tabelle aufgeführt. Dabei steht  $x$  jeweils für ein Array. Der Rückgabewert ist ein Array mit den jeweiligen Funktionswerten (elementweise ausgeführt).

<code>np.sin(x)</code>	Sinus Funktion
<code>np.cos(x)</code>	Cosinus Funktion
<code>np.tan(x)</code>	Tangens Funktion
<code>np.arcsin(x)</code>	Arkussinus Funktion
<code>np.arccos(x)</code>	Arkuskosinus Funktion
<code>np.arctan(x)</code>	Arkustangens Funktion
<code>np.arctan2(x1, x2)</code>	Arkustanges Funktion mit korrektem Quadranten (Elementweise $x1/x2$ )
<code>np.rad2deg(x)</code>	Umwandung Radiant nach Grad
<code>np.deg2rad(x)</code>	Umwandlung Grad nach Radiant



## 2.4.2 Arithmetische Funktionen

Die wichtigsten arithmetischen Funktionen von numpy sind:

<code>np.add(x1, x2)</code>	Elementweises Addieren
<code>np.subtract(x1, x2)</code>	Elementweises Subtrahieren
<code>np.multiply(x1, x2)</code>	Elementweises Multiplizieren
<code>np.divide(x1, x2)</code>	Elementweises Dividieren

## 2.4.3 Weitere Mathematische Funktionen.

Es gibt zahlreiche weitere Mathematische Funktionen. Eine Liste sämtlicher Funktionen sind unter: <http://docs.scipy.org/doc/numpy/reference/routines.math.html> zu finden.

## 2.4.4 Mathematische Konstanten

In numpy sind einige Konstanten eingebaut, dazu gehören `np.pi` für  $\pi$  und `np.e` für  $e$ . Weitere Konstanten sind im Modul scipy zu finden:

<https://numpy.org/doc/stable/reference/constants.html>

## 2.5 NumPy und Matplotlib

### 2.5.1 Plotten einer Sinuskurve

NumPy und Matplotlib funktionieren sehr gut zusammen. Wollen wir beispielsweise eine Sinuskurve im Bereich  $[0, 2\pi]$  zeichnen so können wir mit Numpy das relativ einfach realisieren.

Zunächst importieren wir die erforderlichen Module:

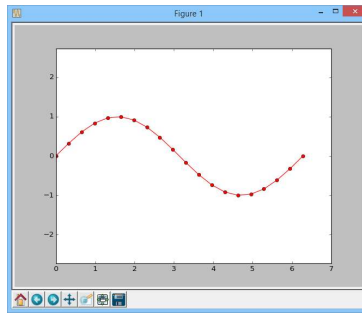
```
import numpy as np
import matplotlib.pyplot as plt
```

Dann erstellen wir ein Array x mit dem Wertebereich und ein Array y mit den Funktionswerten:

```
x = np.linspace(0, 2*np.pi, 20)
y = np.sin(x)
```

Danach können plotten:

```
plt.plot(x, y, "ro-")
plt.axis("equal")
plt.show()
```



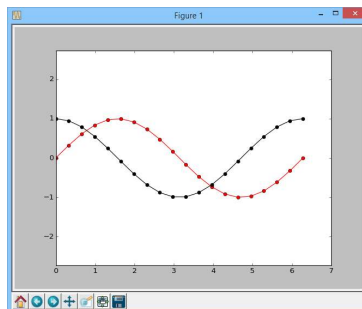
## 2.5.2 Plotten mehrerer Funktionen

Selbstverständlich können auch mehrere Funktionen geplottet werden. Wenn wir beispielsweise eine Sinuskurve und eine Cosinuskurve plotten wollen, so rufen wir einfach mehrmals die Funktion `plot` auf:

```
x = np.linspace(0, 2*np.pi, 20)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x,y1, "ro-")
plt.plot(x,y2, "ko-")

plt.axis("equal")
plt.show()
```

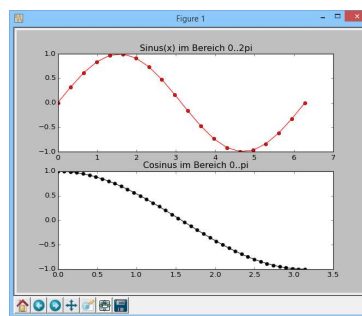


## 2.6 Subplots

Es können auch sogenannte „Subplots“ realisiert werden. Dies sind einfach mehrere Plots welche innerhalb eines Gitters angelegt werden. Zum Initialisieren wird die Funktion subplot verwendet:

```
plt.subplot(AnzahlZeilen, AnzahlSpalten, Nummer)
```

```
x1 = np.linspace(0, 2*np.pi, 20)
x2 = np.linspace(0, np.pi, 40)
y1 = np.sin(x1)
y2 = np.cos(x2)
plt.subplot(2,1,1)
plt.plot(x1,y1, "ro-")
plt.title("Sinus(x) im Bereich 0..2pi")
plt.subplot(2,1,2)
plt.plot(x2,y2, "ko-")
plt.title("Cosinus im Bereich 0..pi")
plt.show()
```



### 2.6.1 Plotten von Figuren

Mit Matplotlib lassen sich auch geometrische Formen relativ einfach zeichnen. Dazu benötigen wir das Axes Objekt, welches das Koordinatensystem definiert:

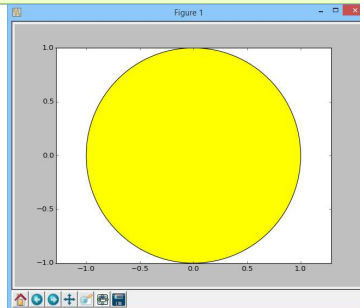
```
axes = plt.axes()
```

Auf diesem axes Objekt können wir unsere Figuren hinzufügen.  
Sehen wir uns dazu folgendes komplettes Beispiel an:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

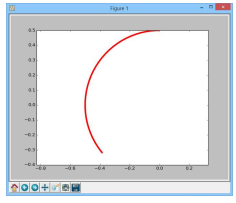
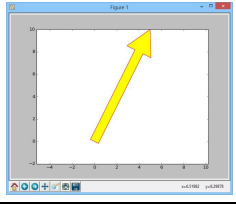
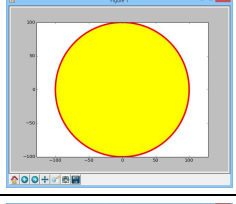
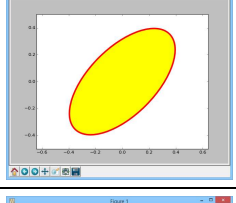
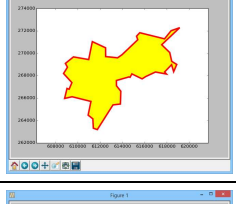
axes = plt.axes()

circle = patches.Circle((0, 0), radius=1.0, facecolor=(1,1,0))
axes.add_patch(circle)
plt.axis("equal")
plt.plot()
```



in dem Beispiel haben wir mit `plt.Circle()` einen Kreis erstellt. Dazu haben wir das Zentrum  $(0, 0)$  angegeben und ein Radius definiert. Zudem haben wir mit `facecolor` (oder `fc`) noch die Füllfarbe definiert. Die Farbe der Kante könnten wir mit `edgecolor` (oder `ec`) setzen. Die Liniendicke der Kante kann mit `linewidth` (oder `lw`) definiert werden.

Die wichtigsten Figuren(„Patches“) in sind:

Bogen	<p><code>Arc(xy,width,height,angle=0.0,theta1=0.0,theta2=360.0)</code></p> <p>xy: Zentrum als tuple z.B. (0,0) width: Länge der horizontalen Achse height: Länge der vertikalen Achse angle: Rotation in Grad (im Gegenuhrzeigersinn) theta1: Start-Winkel in Grad theta2: End-Winkel in Grad</p>	
Pfeil	<p><code>Arrow(x, y, dx, dy, width=1.0)</code></p> <p>x,y: Startposition dx, dy: Länge und Richtung width: Breite</p>	
Kreis	<p><code>Circle(xy, radius=5)</code></p> <p>xy: Zentrum als tuple radius: Radius des Kreises</p>	
Ellipse	<p><code>Ellipse(xy, width, height, angle=0.0)</code></p> <p>xy: Zentrum als tuple width: horizontale Achse height: vertikale Achse angle: Winkel (in Grad, Gegenuhrzeigersinn)</p>	
Polygon	<p><code>Polygon(xy, closed=True)</code></p> <p>xy: Liste von x,y-Koordinaten, z.B. [[x0,y0],[x1,y1],[x2,y2],...,[xn,yn]] closed: True, wenn Endpunkt mit Startpunkt verbunden werden soll</p>	
Rechteck	<p><code>Rectangle(xy, width, height, angle=0.0)</code></p> <p>xy: Koordinate unten links als tuple width: Breite height: Höhe angle: Winkel (in Grad, Gegenuhrzeigersinn)</p>	