

Einführung in die Programmierung mit Python

Lucerne University of
Applied Sciences and Arts

**HOCHSCHULE
LUZERN**

Syllabus

1. Python installieren; Jupyter Notebooks
2. Grundlagen: Arithmethik, Variablen und Datentypen
3. Methoden; Listen; Bedingte Anweisungen
4. Schleifen
5. Benutzerdefinierte Funktionen
6. **Datenaufbereitung und Grafische Darstellungen**

Python-Pakete

- Der Funktionsumfang von Python lässt sich mit sogenannten *Paketen* erweitern.
- Viele davon sind in Anaconda bereits vorinstalliert. Wir müssen sie nur laden (*importieren*):

```
In [1]: import numpy # Paket für numerische Berechnungen. Siehe optionales Notebook
```

Nun kann man Funktionen aus dem Paket wie folgt verwenden:

```
In [2]: numpy.mean([1, 2, 3])
```

```
Out[2]: np.float64(2.0)
```

Um sich Tipparbeit zu ersparen, kann man beim Import einen Kurznamen angeben. Dieser ist willkürlich, aber für häufig genutzte Pakete haben sich Konventionen ergeben, z.B. np für numpy:

```
In [3]: import numpy as np  
np.mean([1, 2, 3])
```

```
Out[3]: np.float64(2.0)
```

Es ist auch möglich, bestimmte Funktionen aus einem Paket zu importieren, so dass man sie direkt verwenden kann:

```
In [4]: from numpy import mean  
mean([1, 2, 3])
```

```
Out[4]: np.float64(2.0)
```

Pandas Dataframes

Einführung in Pandas

- pandas (von *panel data*) ist eines der wichtigsten Pakete ([Benutzerhandbuch](#)).
- Es bietet eine Reihe von Datenstrukturen (*series*, *dataframes* und *panels*), die für die Speicherung von Beobachtungsdaten entwickelt wurden, sowie leistungsstarke Methoden zur Manipulation (*munging* oder *wrangling*) dieser Daten.
- Es wird üblicherweise als `pd` importiert:

```
In [5]: import pandas as pd
```

- Pandas ist unglaublich leistungsfähig; wir werden hier nur die Oberfläche ankratzen.
In der Praxis sind LLMs nützlich, um jeweils die richtige Syntax zu finden.

Series

- Eine Pandas Series ist im Wesentlichen ähnlich einer Liste (genauer einem NumPy-Array), jedoch nicht unbedingt mit Ganzzahlen indiziert.

```
In [6]: pop = pd.Series([5.7, 82.7, 17.0], name='Bevölkerung'); pop # der beschreibt
```

```
Out[6]: 0      5.7
         1     82.7
         2    17.0
Name: Bevölkerung, dtype: float64
```

- Der Unterschied ist, dass der Index beliebig sein kann, nicht nur eine Liste von Ganzzahlen:

```
In [7]: pop.index=['DK', 'DE', 'NL']
```

- Der Index kann für die Indizierung verwendet werden (offensichtlich...):

```
In [8]: pop['NL']
```

```
Out[8]: np.float64(17.0)
```

- Der Index bleibt bei Operationen an einer Series erhalten:

```
In [9]: gdp = pd.Series([3494.898, 769.930], name='Nominales BIP in Milliarden USD', index=[DE, NL])
```

```
Out[9]: DE    3494.898  
NL    769.930  
Name: Nominales BIP in Milliarden USD, dtype: float64
```

```
In [10]: gdp / pop
```

```
Out[10]: DE    42.259952  
DK      NaN  
NL    45.290000  
dtype: float64
```

- Ein Vorteil von `Series` im Vergleich zu Listen (bzw. NumPy-Arrays) ist, dass sie mit fehlenden Daten umgehen können, die als `NaN` (Not A Number) dargestellt werden.

Dataframes

- Ein DataFrame ist eine Sammlung von Series mit einem gemeinsamen Index (der die Zeilen beschriftet).

```
In [11]: data = pd.concat([gdp, pop], axis=1, sort=False); data # zwei Series zu einem
```

```
Out[11]:      Nominales BIP in Milliarden USD  Bevölkerung
```

	Nominales BIP in Milliarden USD	Bevölkerung
DE	3494.898	82.7
NL	769.930	17.0
DK	NaN	5.7

- Spalten werden durch Spaltennamen indiziert:

```
In [12]: data.columns
```

```
Out[12]: Index(['Nominales BIP in Milliarden USD', 'Bevölkerung'], dtype='object')
```

```
In [13]: data['Bevölkerung'] # data.Bevölkerung funktioniert auch
```

```
Out[13]: DE    82.7
          NL    17.0
          DK     5.7
          Name: Bevölkerung, dtype: float64
```

- Zeilen werden mit der Methode `loc` indiziert:

```
In [14]: data.loc['NL']
```

```
Out[14]: Nominales BIP in Milliarden USD      769.93
          Bevölkerung                  17.00
          Name: NL, dtype: float64
```

- Im Gegensatz zu Arrays können Dataframes Spalten mit unterschiedlichen Datentypen haben.
- Es gibt verschiedene Möglichkeiten, Spalten hinzuzufügen. Eine davon ist, einfach einer neuen Spalte zuzuweisen:

```
In [15]: data['Sprache'] = ['Deutsch', 'Niederländisch', 'Dänisch'] # eine neue Spalte
```

- Um Zeilen hinzuzufügen, verwenden Sie loc oder concat:

```
In [16]: print(data.loc["DE"])
data.loc['AT'] = [386.4, 8.7, 'Deutsch'] # eine Zeile mit dem Index 'AT' hinzugefügt
s = pd.DataFrame([[511.0, 9.9, 'Schwedisch']], index=['SE'], columns=data.columns)
data = pd.concat([data, s]) # eine Zeile durch Anhängen eines weiteren Dataframes hinzugefügt
data
```

```
Nominales BIP in Milliarden USD      3494.898
Bevölkerung                          82.7
Sprache                             Deutsch
Name: DE, dtype: object
```

```
Out[16]:
```

	Nominales BIP in Milliarden USD	Bevölkerung	Sprache
DE	3494.898	82.7	Deutsch
NL	769.930	17.0	Niederländisch
DK	NaN	5.7	Dänisch
AT	386.400	8.7	Deutsch
SE	511.000	9.9	Schwedisch

- Die Methode `dropna` kann verwendet werden, um Zeilen mit fehlenden Werten zu löschen:

```
In [17]: data = data.dropna(); data
```

```
Out[17]:
```

	Nominales BIP in Milliarden USD	Bevölkerung	Sprache
DE	3494.898	82.7	Deutsch
NL	769.930	17.0	Niederländisch
AT	386.400	8.7	Deutsch
SE	511.000	9.9	Schwedisch

- Nützliche Methoden zur Gewinnung von Zusammenfassungsinformationen über einen Dataframe sind `mean`, `std`, `info`, `describe`, `head` und `tail`.

```
In [18]: data.describe()
```

```
Out[18]:
```

	Nominales BIP in Milliarden USD	Bevölkerung
count	4.000000	4.000000
mean	1290.557000	29.575000
std	1478.217475	35.605559
min	386.400000	8.700000
25%	479.850000	9.600000
50%	640.465000	13.450000
75%	1451.172000	33.425000
max	3494.898000	82.700000

```
In [19]: data.head() # zeigt die ersten paar Zeilen; data.tail zeigt die letzten paar
```

```
Out[19]:
```

	Nominales BIP in Milliarden USD	Bevölkerung	Sprache
DE	3494.898	82.7	Deutsch
NL	769.930	17.0	Niederländisch
AT	386.400	8.7	Deutsch
SE	511.000	9.9	Schwedisch

- Um einen Dataframe als CSV-Datei auf der Festplatte zu speichern, verwenden Sie

```
In [20]: data.to_csv('myfile.csv') # to_excel existiert ebenfalls.
```

- Um Daten in einen Dataframe zu laden, verwenden Sie pd.read_csv:

```
In [21]: pd.read_csv('myfile.csv', index_col=0)
```

```
Out[21]:
```

	Nominales BIP in Milliarden USD	Bevölkerung	Sprache
DE	3494.898	82.7	Deutsch
NL	769.930	17.0	Niederländisch
AT	386.400	8.7	Deutsch
SE	511.000	9.9	Schwedisch

```
In [22]: import os  
os.remove('myfile.csv') # aufräumen
```

Normalerweise erstellen Sie Dataframes nicht von Grund auf; sie entstehen eher durch das Abrufen von Daten aus einer Quelle. Zum Beispiel kann das Paket yfinance Daten direkt von Yahoo Finance herunterladen.

```
In [23]:
```

```
# !pip install yfinance
import yfinance as yf
from datetime import datetime, timedelta
end_date = datetime.now()
start_date = end_date - timedelta(days=5*365) # siehe unten
nvda_data = yf.download('NVDA', start=start_date, end=end_date)
nvda_data.head()
```

```
C:\Users\JumpStart\AppData\Local\Temp\ipykernel_14492\612714658.py:6:
FutureWarning: YF.download() has changed argument auto_adjust default
to True
    nvda_data = yf.download('NVDA', start=start_date, end=end_date)
[*****100%*****] 1 of 1 completed
```

```
Out[23]:
```

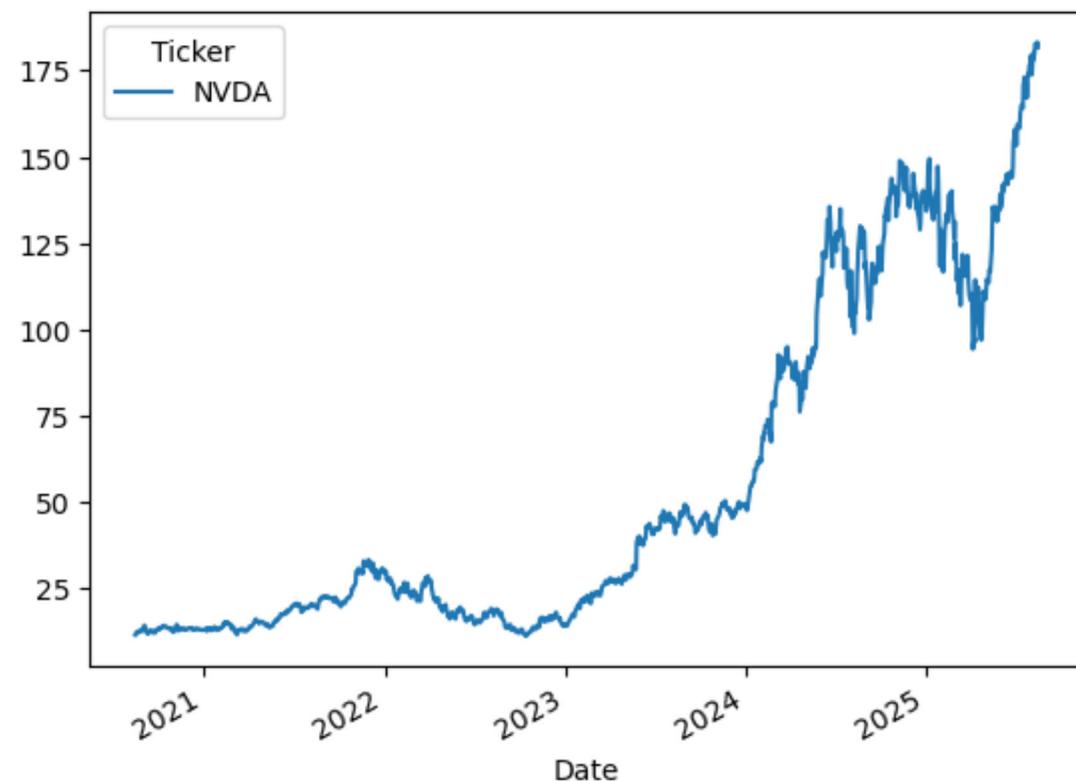
Price	Close	High	Low	Open	Volume
Ticker	NVDA	NVDA	NVDA	NVDA	NVDA
Date					
2020-08-14	11.526009	11.666296	11.402914	11.492120	366436000
2020-08-17	12.296466	12.368977	11.778424	11.812312	621300000
2020-08-18	12.220470	12.454948	12.046544	12.409099	503448000
2020-08-19	12.098622	12.274542	12.058255	12.256103	622624000
2020-08-20	12.101114	12.334346	11.839726	11.935660	921388000

Plotten

Pandas kann direkt zum Plotten verwendet werden. Erweiterte Funktionen erfordern `matplotlib` (mehr dazu unten).

```
In [24]: nvda_data.Close.plot()
```

```
Out[24]: <Axes: xlabel='Date'>
```

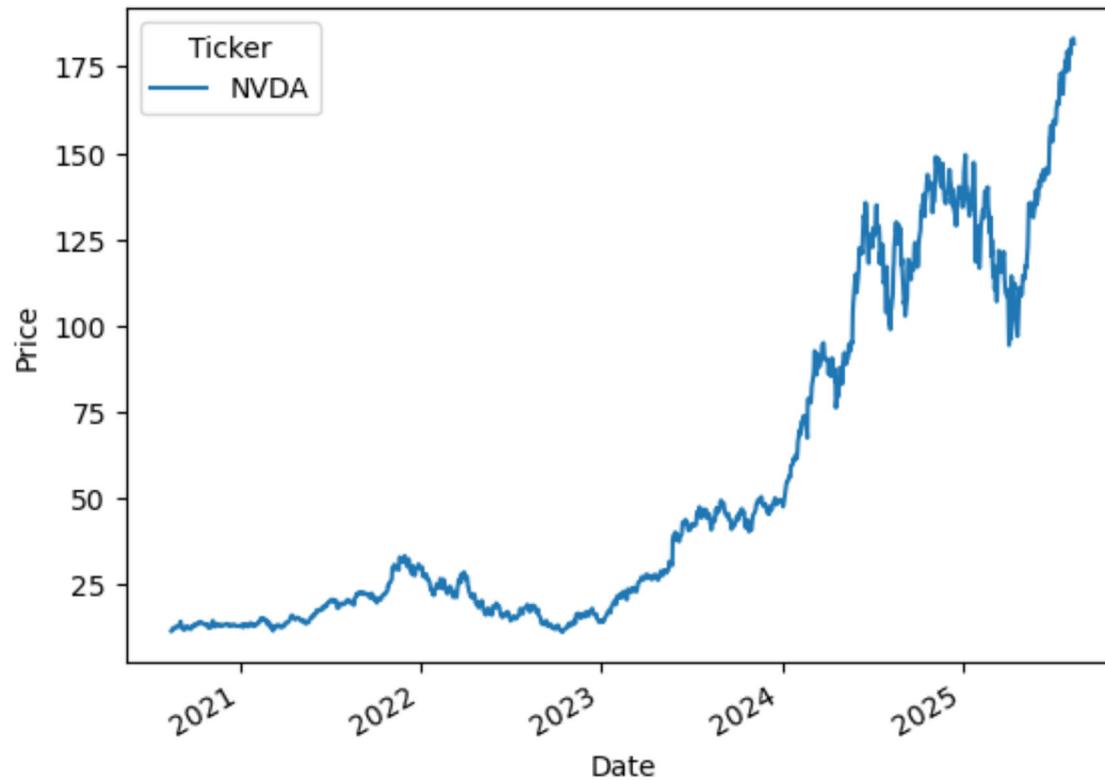


- Erweitertes Plotten erfordert die Bibliothek `matplotlib` ([Benutzerhandbuch](#)), die von den Plotting-Funktionen von Matlab® inspiriert ist.
- Die Haupt-Plotting-Funktionen befinden sich im Modul `pyplot`. Es wird üblicherweise importiert als

```
In [25]: import matplotlib.pyplot as plt
```

`matplotlib` ermöglicht es uns, den obigen Plot zu customizen:

```
In [26]: nvda_data.Close.plot()  
plt.xlabel("Date")  
plt.ylabel("Price")  
plt.gcf().autofmt_xdate()  
plt.show()
```



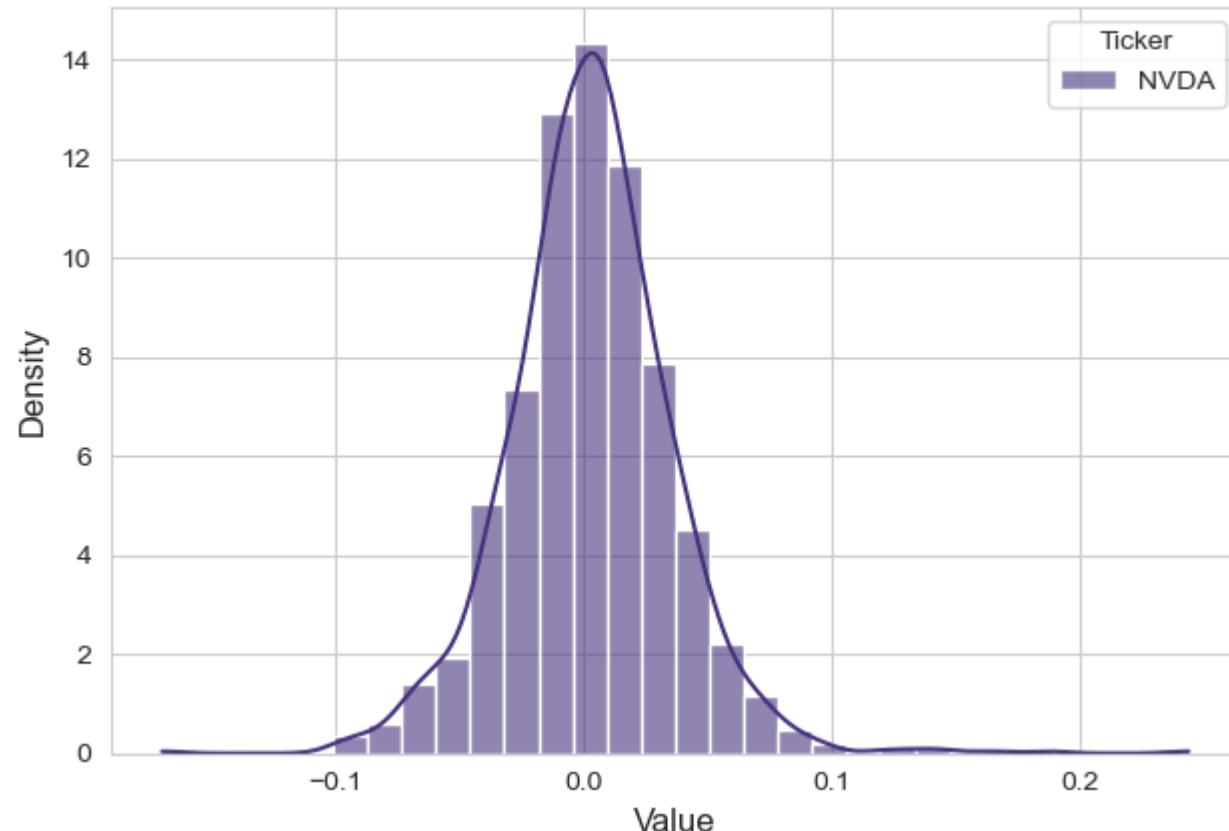
- Die Bibliothek `seaborn` ([Benutzerhandbuch](#)) bietet statistische Visualisierungen auf höherem Niveau:

```
In [27]: # !pip install seaborn
import seaborn as sns
sns.set_style("whitegrid") # Kosmetik
sns.set_palette("viridis") # Kosmetik

returns = nvda_data.Close.pct_change() # Tagesrenditen

sns.histplot(data=returns, bins=30, kde=True, stat="density", alpha=0.6)
plt.title("Distribution of Returns", fontsize=14, pad=15)
plt.xlabel("Value", fontsize=12)
plt.ylabel("Density", fontsize=12)
plt.tight_layout()
plt.show()
```

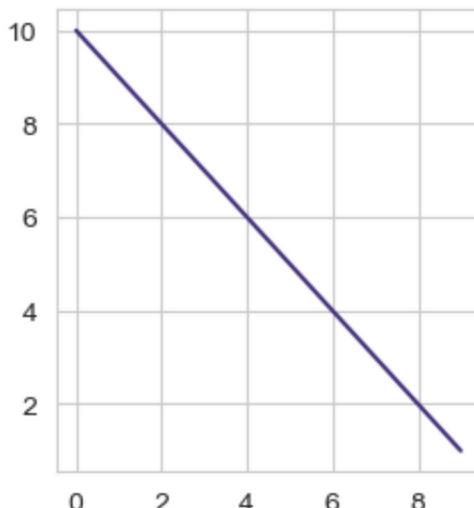
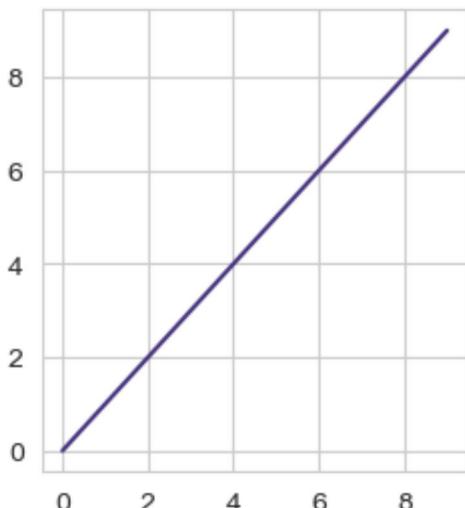
Distribution of Returns



- Ich werde hier nur eine kurze Einführung in matplotlib geben. Das grundlegende Objekt in matplotlib ist eine `Figure`, in der sich `Subplots` (oder `Axes`) befinden.
- Um eine neue `Figure` zu erstellen, einen `Axis` hinzuzufügen und darauf zu plotten:

In [28]:

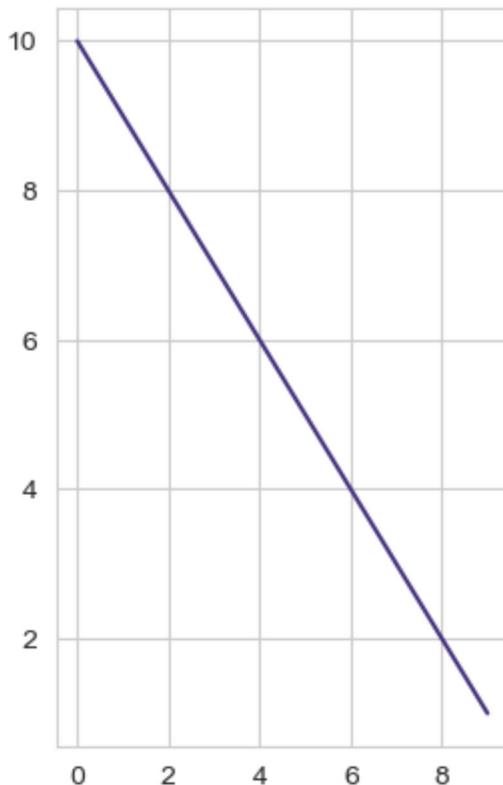
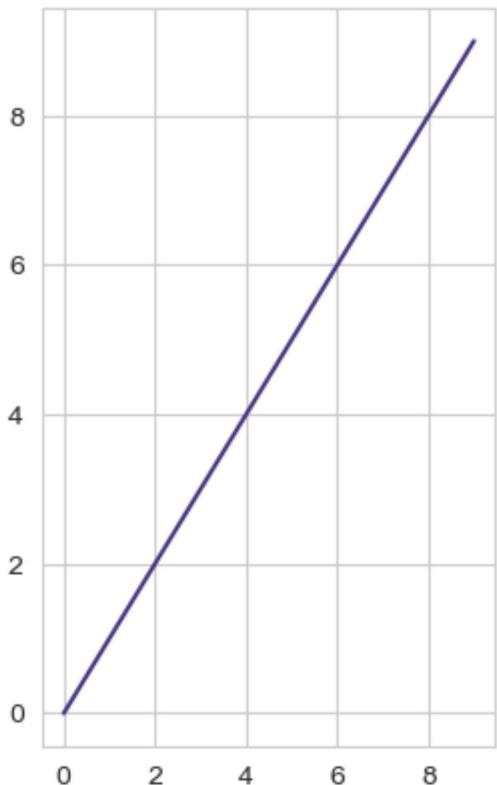
```
# mit dem Inline-Backend müssen diese im selben Cell sein.  
fig = plt.figure(figsize=(6,3)) # eine neue leere Figure erstellen. Größe ist 6x3  
ax1 = fig.add_subplot(121) # Layout: (1x2). ax1 ist der obere linke  
ax2 = fig.add_subplot(122)  
ax1.plot(range(10))  
ax2.plot(range(10, 0, -1));
```



- Standardmäßig plottet matplotlib in den aktuellen Axis und erstellt einen (sowie eine Figure), falls nötig. Mit der Bequemlichkeitsmethode `subplot` können wir dasselbe erreichen, ohne explizit auf Figures und Axes zu verweisen:

In [29]:

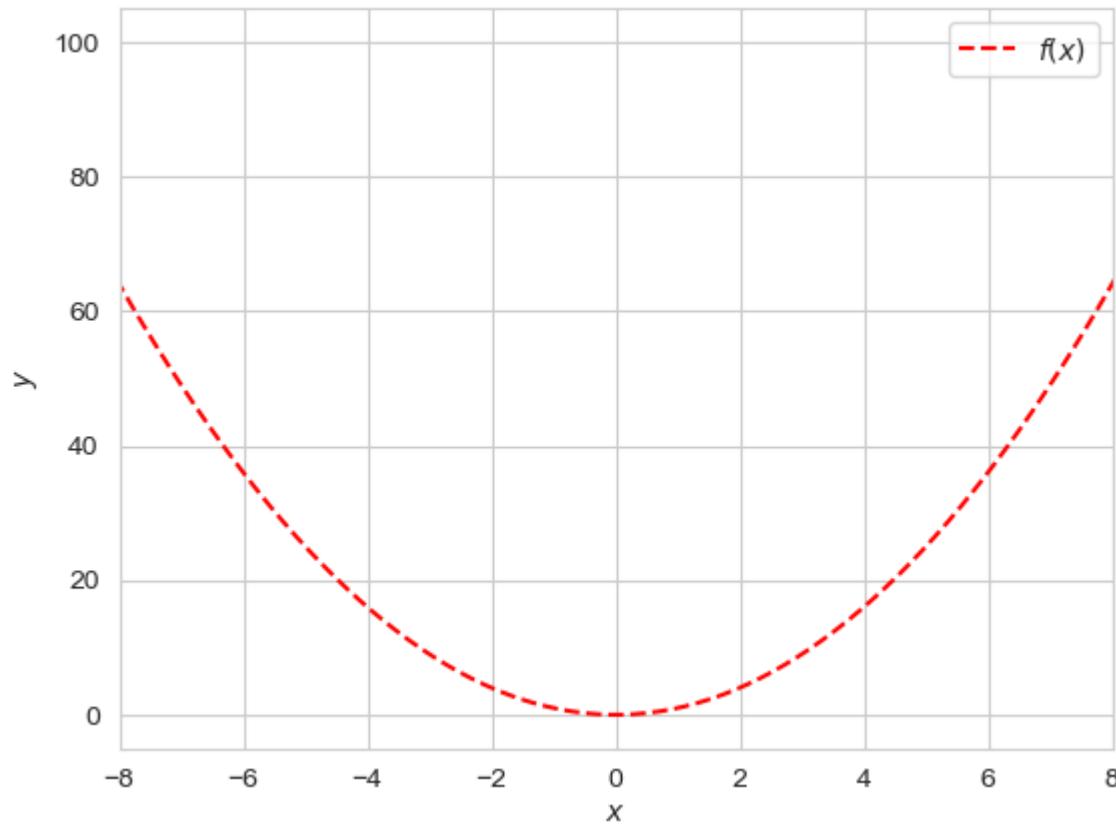
```
plt.subplot(121)
plt.plot(range(10))
plt.subplot(122)
plt.plot(range(10, 0, -1));
```



- Um zwei Vektoren x und y gegeneinander zu plotten:

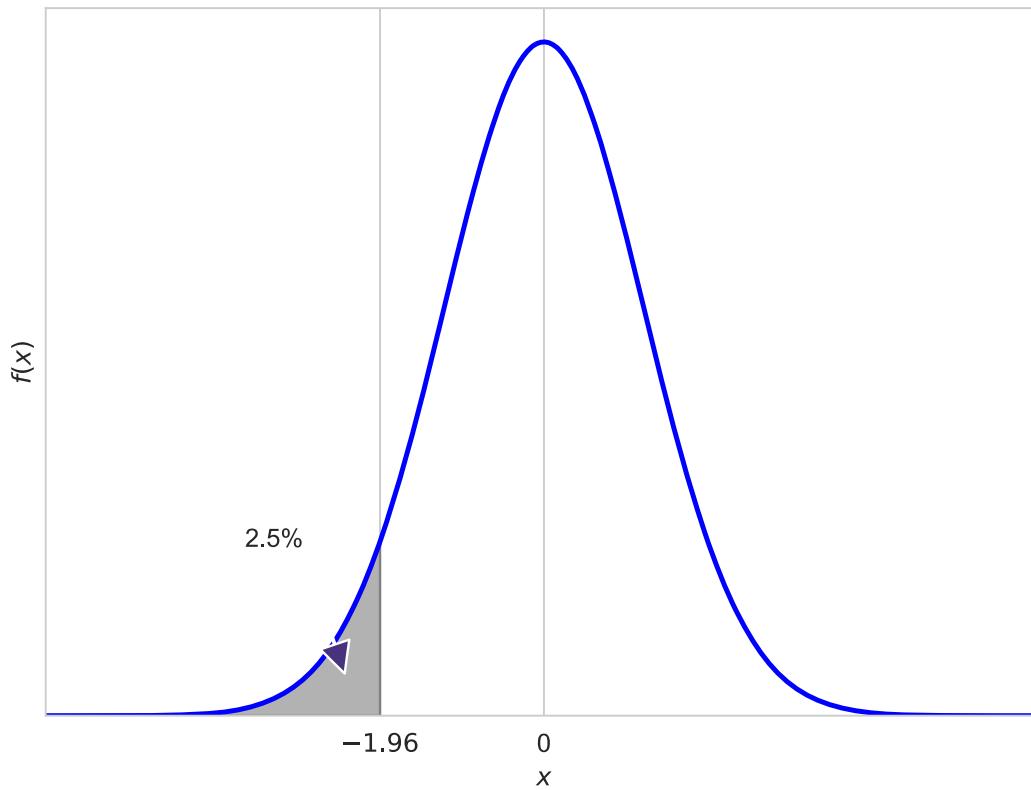
```
In [30]: import numpy as np
x = np.linspace(-10, 10, 100)
y = x**2
plt.plot(x,y,'r--') # gestrichelte rote Linie
plt.xlabel('$x$') # Mathematische (LaTeX) Gleichungen können durch Einschließen
plt.ylabel('$y$')
plt.title('Eine Parabel')
plt.legend(['$f(x)$']) # erwartet eine Liste von Strings
plt.xlim(xmin=-8, xmax=8); # Achsengrenzen
# plt.savefig('filename.pdf') # den Plot auf der Festplatte speichern
```

Eine Parabel



Ein fortgeschrittenes Beispiel: Quantile der Normalverteilung

```
In [31]: from matplotlib.patches import Polygon
import scipy.stats as stats
a, b, c = -5, 5, stats.norm.ppf(0.05)
x = np.linspace(a, b, 500)
y = stats.norm.pdf(x)
fig = plt.figure(figsize=(7, 5))
ax = fig.add_subplot(111)
plt.plot(x, y, 'b', linewidth=2)
plt.ylim(ymin=0)
plt.xlim(xmin=a, xmax=b)
Ix = np.linspace(a, c)
Iy = stats.norm.pdf(Ix)
verts = [(a, 0)] + list(zip(Ix, Iy)) + [(c, 0)]
poly = Polygon(verts, facecolor='0.7', edgecolor='0.5')
ax.add_patch(poly)
ax.annotate(
    '2.5%', xy=(-2, 0.025), xytext=(-3, 0.1),
    arrowprops=dict(width=.5),
)
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
ax.set_xticks([c, 0])
ax.set_xticklabels(['-$1.96$', '0'])
ax.set_yticks([])
plt.savefig('img/var.svg')
plt.close()
```



Arbeiten mit Zeitreihen

Datentypen

- Es gibt verschiedene Datentypen in Python zur Darstellung von Zeiten und Daten.
- Der grundlegendste ist `datetime` aus dem gleichnamigen Paket:

```
In [32]: from datetime import datetime  
datetime.today()
```

```
Out[32]: datetime.datetime(2025, 8, 13, 16, 57, 5, 769563)
```

- `datetime`-Objekte können aus Strings mit `strptime` und einem Format-Spezifizierer erstellt werden:

```
In [33]: datetime.strptime('2017-03-31', '%Y-%m-%d')
```

```
Out[33]: datetime.datetime(2017, 3, 31, 0, 0)
```

- Pandas verwendet `Timestamps` anstelle von `datetime`-Objekten. Im Gegensatz zu `Timestamps` speichern sie Frequenz- und Zeitzoneneinformationen. Die beiden können größtenteils austauschbar verwendet werden.

```
In [34]: pd.Timestamp('2017-03-31')
```

```
Out[34]: Timestamp('2017-03-31 00:00:00')
```

- Eine Zeitreihe ist eine `Series` mit einem speziellen Index, genannt `DatetimeIndex`; im Wesentlichen ein Array von `Timestamp`s.
- Sie kann mit der Funktion `date_range` erstellt werden.

```
In [35]: import numpy as np
myindex = pd.date_range(end=pd.Timestamp.today(), normalize=True, periods=100,
P = 20 + np.random.randn(100).cumsum() # einige Aktienkurse erfinden.
aapl = pd.Series(P, name="AAPL", index=myindex)
aapl.tail()
```

```
Out[35]: 2025-08-07    18.202414
          2025-08-08    18.639847
          2025-08-11    18.719556
          2025-08-12    19.122900
          2025-08-13    19.122571
Freq: B, Name: AAPL, dtype: float64
```

- Zur Vereinfachung erlaubt Pandas die Indizierung von Zeitreihen mit Datumsstrings:

```
In [36]: aapl['4/11/2025']
```

```
Out[36]: np.float64(23.100221382689444)
```

Empfohlene Lektüre

- <https://python-course.eu/numerical-programming/> 23-28, 32-34

Hausaufgaben

[https://github.com/guipsamora/pandas_exercises/tree/
master/01_Getting_%26_Knowing_Your_Data/Chipotle](https://github.com/guipsamora/pandas_exercises/tree/master/01_Getting_%26_Knowing_Your_Data/Chipotle) (Nur bis Aufgabe 12. Zusätzlich:

Visualisieren Sie die Verteilung der Artikelpreise mit Seaborn und/oder Matplotlib)

In []: