8 Python

8.1 Einführung

Ein hervorragende Quelle für eine Einführung in Python ist der ehemalige Kurs von *Dr. Jochen Schulz*.

Mathematisch orientiertes Programmieren, Ein Kurs Python im wissenschaftlich/mathematischen Kontext zu erlernen.

Im Absprache mit Dr. Jochen Schulz werden in diesem Kapitel Texte und Programmcode aus diesem Kurs (teilweise unverändert) übernommen, ohne expliziet als Zitat gekennzeichnet zu sein.

An dieser Stelle vielen Dank an Dr. Jochen Schulz für seine freundliche Genehmigung.

Die vollständige *Python documentation* inklusive

- Tutorial
- Bibliotheks- und Sprachreferenz
- HOWTOs und FAQs
- etc.

finden Sie unter `https://docs.python.org/3/` .

In dieser Vorlesung wird **Python 3** verwendet.

Der Vorgänger Python 2 ist auch noch produktiv im Einsatz, aber leider sind Python 3 und Python 2 **nicht kompatibel**.

für neue Projekte sollte man darauf achten Python 3 zu verwenden.

Unter Linux ist das Programm `python3` sowohl ein **Python-Interpeter**, der **Python-Skripte** Zeile für Zeile ausführt, als auch eine interaktiven **Python-Shell**, die zeilenweise Code entgegen nimmt.

Python-Interpreter

Ein Python-Skript ist eine Textdatei, die Python-Programmcode enthält und, nach Konvention, mit `.py` endet.

Ein Python-Skript kann zur Abarbeitung an den Python-Interpeter übergeben werden.

Fügt man einem Python-Skript als erste Zeile den passenden *shebang*, z.B. **`#!/usr/bin/python3`**, hinzu und ist die Datei ausführbar kann man das Python-Skript direkt auf der Kommandozeile starten.

Beispiel

Datei **`hello.py`**, Version 1

```
1 print('hello world')
```

```
> python3 hello.py
hello world
```

Datei **`hello.py`**, Version 2

```
1 #!/usr/bin/python3
2 print('hello world')
```

```
> chmod u+x hello.py
> ./hello.py
hello world
```

Python-Shell

Startet man **`python3`** erhält man folgende Ausgabe.

```
> python3
Python 3.x.x (default, <date>, <time>)
[GCC 9.x.x] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Das Prompt **`>>>`** gibt an, dass man sich in der interaktiven Python-Shell befindet und Code zeilenweise eingeben kann.

Beendet wird die Python-Shell mit *Ctrl-D* (*Strg-D*).

```
>>> 42
42
>>> 2+3
5
```

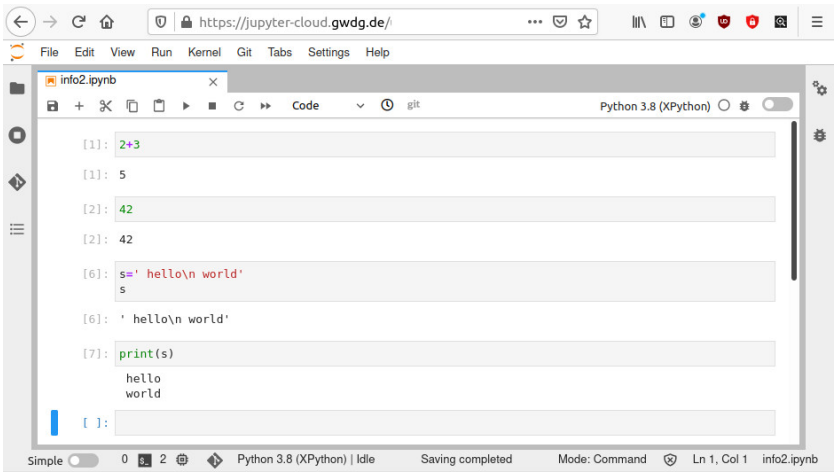
Blöcke (Prompt `...`) werden mit einer Leerzeile abgeschlossen, eingerückt werden kann mit *Tab* (*Tabulator*) oder *Spaces* (*Leerzeichen*).

```
>>> if True:
...     'hello world'
...
'hello world'
```

Jupyter

Jupyter ist eine Web-Anwendung, mit der (verteilte) Dokumente erstellt und bearbeitet werden können, die Code (inklusive Auswertung), Visualisierung und erläuternden Text enthalten.

GWDG Jupyter `https://jupyter-cloud.gwdg.de/` stellt u.a. einen Python3-Kernel bereit, der für die Beispiele dieser Veranstaltung und Experimente genutzt werden.



In Jupyter können Zellen mehrere Zeilen enthalten.

Die Tastenkombination *Shift-Return* (*Umschalt-Eingabe*) veranlasst das Auswerten einer Zelle.

Python-Skripte

Python-Skripte sind eine Liste von **Befehlen** (*statements*), die sequentiell ausgeführt werden.

Mögliche Befehle lassen sich (unvollständig) unterteilen in **einfache** und **zusammengesetzte Befehle**.

Einfache Befehle werden üblicherweise in einer einzelnen Zeile geschrieben.

- Ausdrücke
- Zuweisungen
- Kontroll-Fluss-Befehle (`return`, `continue`, ...)
- ...

Zusammengesetzte Befehle

- Funktionsdefinitionen (`def`)
- Konditionale (`if`)
- Schleifen (`for`, `while`)
- Objekt-Klassen-Definitionen (`class`)
- ...

Ausdrücke

Ausdrücke (*expressions*) bestehen aus Werten, Variablen und aus Operatoren, die diese kombinieren.

Beispiele

```
>>> 2+3
5
```

```
>>> 42
42
```

Ausdrücke haben selbst einen Wert; Ausdrücke, die als Befehl oder Teil von Befehlen auftreten, werden ausgewertet, d.h. so lange gemäß einer Reihe von Regeln reduziert, bis ein Wert übrig bleibt.

Hinweis

Die interaktive Shell zeigt den Wert jedes Ausdrucks an. Benutzt man ein Jupyter-Notebook und ist der letzte Befehl ein Ausdruck, so wird der Wert

diese Ausdrucks angezeigt. Ansonsten muss man für Ausgaben die `print` Funktion verwenden (siehe *Funktionen*).

Kommentare

Jede Eingabe wird generell als Code interpretiert und ausgeführt.

Will man **Kommentare** hinzufügen, die nicht interpretiert werden sollen, geht das mit dem Sonderzeichen `#`.

Der Text, der in einer Zeile auf das `#`-Zeichen folgt, wird nicht interpretiert und ausgeführt.

```
>>> 2+3   # an expression
5
```

Funktionen

Neben einfachen arithmetischen Operatoren können Ausdrücke Aufrufe von **Funktionen** enthalten.

Python-Funktionen sind ähnlich wie Funktionen (Abbildungen) im mathematischen Sinn.

Die Syntax für Funktionsaufrufe ist wie folgt.

```
function_name(argument1, argument2, ...)
```

Ein Funktionsaufruf wird zu einem Wert, dem **Rückgabewert**, ausgewertet und eine Funktion kann zusätzlich einen **Effekt** haben.

Die **Argument** von Funktionen sind Ausdrücke.

Bemerkungen

- In funktionalen Programmiersprachen (z.B. Haskell) haben Funktionen, wie in der Mathematik, keinen Effekt.
- Anders als z.B in Haskell muss die Argumentliste beim Funktionsaufruf in runde Klammern eingeschlossen und durch Kommata getrennt werden.

Beispiel

Zur Betragsfunktion

$$|x| = \begin{cases} x & \text{für } x \geq 0 \\ -x & \text{sonst} \end{cases}$$

gibt es eine Entsprechung in Python, die Funktion **abs**.

```
>>> abs(-1)
1
```

Der Funktionsaufruf wird zum Rückgabewert ausgewertet, mit dem dann wieder Ausdrücke und Funktionsaufrufe gebildet werden können.

```
>>> abs(abs(-1)-2)
1
```

Beispiel

Die Funktion **print** wird zum speziellen Wert **None** ausgewertet, d.h. praktisch hat **print** keinen Rückgabewert, und hat den Effekt, dass sein Argument auf die Standardausgabe geschrieben wird.

```
>>> print(10)
10
```

Bemerkung

Der Effekt von **print** unterscheidet sich von der implizieten Ausgaben des Wertes eines Ausdrucks, bei der das Objekt, das den Wert enthält, implizit ausgegeben wird.

```
>>> print(10)
10
>>> 10
10
>>> print('hello')
hello
>>> 'hello'
'hello'
>>> print(' hello\n world')
 hello
 world
>>> ' hello\n world'
' hello\n world'
```

Variablen und Zuweisungen

Alle Werte in Python sind Objekte (siehe *Objekte*), die in irgendeiner geeigneten Form im Arbeitsspeicher liegen.

Variablen haben **Bezeichner** (*identifier*) und speichern Referenzen auf Objekte, d.h. mit Variablen können Objekte referenziert werden.

Eine **Zuweisung** hat folgende Form.

```
identifier = expression
```

Der Ausdruck **`expression`** wird ausgewertet, und das Resultat wird der Variable mit dem Bezeichner **`identifier`** zugewiesen.

Tritt in folgenden Ausdrücken derselbe Bezeichner auf, wird er durch den der Variable zugewiesenen Wert ersetzt. Dabei wird der Ausdruck nicht noch einmal ausgewertet.

Hat der Ausdruck einen Effekt, tritt dieser nur einmal auf, nämlich zum Zeitpunkt der Zuweisung.

Beispiel

```
>>> a=3+5
>>> print(a)
8
>>> b=a
>>> print(b)
8
```

Die Variablen **`a`** und **`b`** referenzieren dasselbe Objekt mit dem Wert 8, welches Ergebnis des Ausdrucks **`3+5`** ist.

Die linke Seite einer Zuweisung muss eine Variable sein, die das Objekt, welches das Ergebnis der rechten Seite repräsentiert, referenzieren kann.

```
>>> 2*a=4
  File "<stdin>", line 1
SyntaxError: cannot assign to operator
```

Auf diese Weise z.B. Gleichungen zu lösen ist nicht möglich.

Bezeichner (*identifier*)

Bezeichner bestehen aus

- Groß- und Kleinbuchstaben `A-Z` und `a-z`,
- Ziffern `0-9`,
- dem Unterstrichen _.

Bezeichner

- dürfen nicht mit einer Ziffer beginnen,
- unterscheiden zwischen Groß- und Kleinschreibung,
- sollten keine Umlaute oder sonstige Sonderzeichen enthalten.

Bemerkung

Die Beschränkung auf die Buchstaben und Ziffern des ASCII (*American Standard Code for Information Interchange*) ist eine Konvention. Der Interpreter akzeptiert auch Unicode Buchstaben und Ziffern, z.B. `ä` oder α (*code point* `U+03B1`).

In dieser Veranstaltung halten wir uns an die Konvention.

Schlüsselwörter (*keywords*)

Es gibt eine Reihe von Schlüsselwörtern (*keywords*), die reserviert sind und nicht als Bezeichner verwendet werden dürfen.

Die Schlüsselwörter können in verschiedenen Python-Versionen unterschiedlich sein, d.h. einige könnten hinzugefügt oder einige entfernt werden.

Eine Liste der Schlüsselwörter der eingesetzten Version bekommt man mit Hilfe des Moduls `keyword` (siehe *Module*).

```
>>> import keyword
>>> keyword.kwlist
'False', 'None', 'True', 'and', 'as', 'assert', 'async',
'await', 'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Klassen und Objekte

Eine **Klasse** beschreibt die gemeinsamen Attribute (Variablen) und Methoden (Funktionen) einer Menge von gleichartigen Instanzen dieser Klasse.

Instanzen einer Klasse werden **Objekte** (dieser Klasse) genannt.

Die Klasse eines Objekts bestimmt, wie das Objekt im Speicher abgelegt wird und welche Operationen mit dem Objekt durchgeführt werden können.

Alle Werte in Python sind Objekte.

Die Klasse eines Objekts wird auch als **Typ** des Objekts bezeichnet.

Beispiel

Mit der Funktion `type` kann der Typ/die Klasse eines Objekts ermittelt werden.

```
>>> type(10)
<class 'int'>
>>> type(None)
<class 'NoneType'>
```

In Python haben Variablen keinen Typ, sondern nur Objekte (Werte).

Der Typ einen Ausdrucks, dazu gehört auch ein Variablenname, ist der Typ der Wertes, zu dem der Ausdruck ausgewertet wird.

```
>>> a=10
>>> type(a)
<class 'int'>
>>> a=True
>>> type(a)
<class 'bool'>
>>> a=None
>>> type(a)
<class 'NoneType'>
```

Ein Objekt kann selbst wieder Variablen (**Objekt-Attribute**) und Funktionen (**Objekt-Methoden**) besitzen.

Es gilt i.A. die Regel das eine Klasse alle Funktionen definiert, die auf Objekte dieser Klasse angewendet werden können.

Python hat eingebaute Methoden (*built-in functions*), z.B. **`print`** oder **`type`**, die global verfügbar sind und auf (beliebige) Objekte angewendet werden können.

Objekt-Methoden (Funktionen) werden durch das Objekt, gefolgt von einem Punkt, gefolgt von dem Namen der Funktion aufgerufen.

```
object.function()
```

Was der Methodenaufruf tut, hängt nicht nur von den Argumenten, sondern auch vom Objekt selbst ab. Insbesondere können Methoden den Zustand, d.h. die Attribute des Objekt verändern.

Objekt-Attribute (Variablen) werden durch das Objekt, gefolgt von einem Punkt, gefolgt von dem Variablennamen aufgerufen.

```
objekt.variable
```

Beispiel

Die *built-in function* **`bin`** konvertiert eine Integer-Zahl (Objekt der Klasse **`int`**) in eine binäre Zeichenkette mit dem Präfix **`0b`**.

Die Klasse **`int`** definiert eine Objekt-Methode **`int.bit_length()`** die jedes Objekt der Klasse **`int`** (Integer-Zahl) besitzt.

`int.bit_length()` gibt die Anzahl der Bits zurück, die notwendig sind, um eine Ganzzahl als binäre Zeichenkette darzustellen, ohne Vorzeichen und führende Nullen.

```
>>> a=550
>>> bin(a)
'0b1000100110'
>>> a.bit_length()
10
```

Elementare Datentypen (*built-in types*)

Elementare Datentypen (*built-in types*) werden durch den Interpreter global bereitgestellt.

Numerische Typen (*numeric types*)

- Wahrheitswert (`bool`)
- Ganze Zahl (`int`)
- Gleitkommazahl (`float`)
- Komplexe Zahl (`complex`)

Sequenz-Typen *sequence types*

- String/Zeichenkette (`str`)
- Liste (`list`)
- Tupel (`tuple`)
- Bereich (`range`)

Zuordnungs-Typen *mapping types*

- Dictionary/Wörterbücher (`dict`)

Bemerkung. Komplexe Zahlen werden wir nicht behandeln.

Die Objekte der meisten elementaren Datentypen (z.B. `bool`, `int`, `float`, `string`, `tuple`, `range`) sind *immutable* (unveränderlich).

Nachdem ein *immutable* Objekt erstellt und ihm ein Wert zugewiesen wurde, kann man diesen Wert und damit auch den Zustand des Objekts nicht mehr ändern.

Einige elementaren Datentypen (z.B. `list`, `dict`) sind *mutable*, der Zustand des Objektes, kann mit geeigneten Objekt-Methoden verändert werden.

8.2 Numerische Typen

8.2.1 Wahrheitswerte (`bool`)

Wahrheitswerte (*booleans*) repräsentieren Wahr/Falsch-Werte und haben den Typ `bool`.

```
True
False
```

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Es gibt drei **Operatoren auf Wahrheitswerten**.

Operator	Bedeutung
`x or y`	wenn `x` *wahr* ist, dann liefere `x`, sonst `y`
`x and y`	wenn `x` *falsch* ist, dann liefere `x`, sonst `y`
`not x`	wenn `x` *falsch* ist, dann liefere `True`, sonst `False`

Die Operatoren `and` und `or` garantieren die **bedingte Auswertung** (*short-circuit evaluation*), d.h.

- `or` wertet den zweiten Ausdruck nur aus, wenn der erste *falsch* ist,
- `and` wertet den zweiten Ausdruck nur aus, wenn der erste *wahr* ist,

weil im jeweils anderen Fall das Ergebnis der Auswertung bereits fest steht.

Es gibt acht **Vergleichsoperatoren** die einen Wahrheitswert zurückliefern.

Operator	Bedeutung
`<`	echt kleiner
`<=`	kleiner oder gleich
`>`	echt größer
`>=`	größer oder gleich
`==`	gleich
`!=`	ungleich
`is`	gleiches Objekt
`is not`	unterschiedliches Objekt

8.2.2 Ganze Zahlen (`int`)

Ganze Zahlen haben den Typ `int`.

```
>>> 1 + 3
4
>>> -1 * 9
-9
>>> 2 ** 3    # 2 to the power 3
8
```

Ganze Zahlen können beliebig groß sein, solange genug Arbeitsspeicher vorhanden ist.

```
>>> 2 ** 1000
10715086071862673209484250490600018105614048117055336074437503883703510511
```

Division ergibt (seit Python 3) immer eine Gleitkommazahl (siehe *Gleitkommazahlen*).

```
>>> 9/3
3.0
```

Ganzzahlige Divison ist die Division mit Rest (*remainder*), das Ergebnis wird ganzzahligen durch Abrunden.

```
>>> 9//3      # remainder 0
3
>>> 10//3     # remainder 1
3
>>> -10//3    # remainder 2
-4
>>> 10//-3    # remainder -2
-4
```

Modulo berechnet den Rest der ganzzahligen Division, es gilt **`n==(n//m)*m+n%m`** .

```
>>> 9%3
0
>>> 10%3
1
>>> -10%3
2
>>> 10%-3
-2
```

8.2.3 Gleitkommazahlen (`float`)

Gleitkommazahlen sind Zahlen mit begrenzter Genauigkeit; nur eine beschränkte Anzahl an signifikanten Stellen kann gespeichert werden (meist 53 Bits). Diese haben den Typ **`float`**.

Die Eingabe von **`float`** geschieht implizit durch die Angabe eines Punktes für die Nachkommastellen.

```
>>> type(3.0)
<class 'float'>
>>> type(3.)
<class 'float'>
```

Durch die begrenzte Genauigkeit können Rundungsfehler auftreten und Stellen wegfallen.

```
>>> 3 * 0.1
0.30000000000000004
>>> 0.1 == 0.10000000000000001
True
```

8.2.4 Operationen

Die numerischen Typen unterstützen u.a. die folgenden **arithmetischen Operationen**.

Operation	Bedeutung
`x+y`	Summe von `x` und `y`
`x-y`	Differenz von x und `y`
`x*y`	Produkt on `x` und `y`
`x/y`	Quotient von `x` und `y`
`x//y`	abgerundeter Quotient von `x` und `y`
`x%y`	Rest von `x//y`
`-x`	`x` negiert
`+x`	`x`
`abs(x)`	Betrag von `x`
`int(x)`	`x` umgewandet in eine ganze Zahl
`float(x)`	`x` umgewandet in eine Gleitkommazahl
`pow(x,y)`	`x` hoch `y`
`x**y`	`x` hoch `y`

Die Typen `bool` und `int` unterstützt die folgenden **Bit-Operatoren**.

Operator	Bedeutung
`x|y`	bitweise *or* von `x` und `y`
`x^y`	bitweise *exclusive or* von `x` und `y`
`x&y`	bitweise *and* von `x` und `y`
`x<<n`	`x` verschoben nach links um `n` Bits
`x>>n`	`x` verschoben nach rechts um `n` Bits
`~x`	bitweise Invertierung von `x`

Bemerkung

für alle Operatoren ist einen Priorität und Assoziativität festgelegt, die Reihenfolge und Auswertungsrichtung eines Ausdrucks mit mehreren Operatoren bestimmt. Darauf wird hier nicht weiter eingegangen.

Ist die Auswertungsreihenfolge wichtig, muss passende geklammert (mit runden Klammern) werden.

8.3 Sequenz-Typen

8.3.1 Strings (`str`)

Strings (Zeichenketten) haben den Typ **`str`** und werden mit einfachen `'` oder doppelten `''` Anführungszeichen gekennzeichnet.

```
>>> s='hello world'
>>> s
'hello world'
>>> type(s)
<class 'str'>
```

Strings können beliebige Sonderzeichen enthalten.

```
>>> print('öäü')
oau
```

Mehrere Strings direkt hintereinander werden zusammengefügt, was manchmal bei sehr langen Zeilen nützlich sein kann.

```
>>> print('Lorem ipsum dolor sit amet, consetetur sadipscing '
... 'elitr, sed diam nonumy eirmod tempor invidunt ut labore '
... 'et dolore magna aliquyam erat, sed diam voluptua.')
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
```

Mehrzeilige Strings können auch mit drei `'` oder `''` geschrieben werden.

```
>>> print('''Ein String
... der ueber mehrere Zeilen
... geht und Zeilenumbrueche
... direkt enthaelt.''')
Ein String
der ueber mehrere Zeilen
geht und Zeilenumbrueche
direkt enthaelt.
```

Bemerkung

Die Python-Shell bemerkt die fehlende schließende Klammer/die fehlenden drei ’ und beginnt einen neue Zeile mit ... für den Rest der Parameterliste/des mehrzeiligen Strings.

In eine Python-Skript können Sie die Zeilen einfach untereinander schreiben.

Der Backslash \ wird verwendet, um Zeichen zu codieren, die selbst eine besondere Bedeutung haben, wie z.B. die Anführungszeichen \’ und \’’ oder den Backslash selbst \\.

Weiterhin kann man so auch Zeichen codieren, die man nicht ohne weiteres eingegeben kann z.B. den Zeilenumbruch **n**.

```
>>> print('String with quote \', double quote ", '
... 'newline\nand backslash \\.')
String with quote ', double quote ", newline
and backslash \.
```

Bemerkungen

- Die mit einem Backslash beginnenden Zeichenfolgen nennt man *escape sequences*.
- Mit **u** kann man einen Unicode *code point* angegeben, z.B. **"\u03B1"** für α.
- Innerhalb von einfachen Anführungszeichen müssen doppelte Anführungszeichen nicht als *escape sequences* dargestellt werden und umgekehrt.

Die Klasse **str** definiert die Objekt-Methode **format**, die auf jedem Objekt von **str** aufgerufen werden kann und es erlaubt in Strings Platzhalter durch die Stringrepräsentationen beliebiger Werte zu ersetzen.

Eine **kurze** Einführung.

```
>>> s = 'Pi is approximately {}. Another approximation is {}/{}.'
>>> a = 3.1415
>>> b = 22
>>> c = 7
>>> print(s.format(a, b, c))
Pi is approximately 3.1415. Another approximation is 22/7.
```

Dabei wird der n-te Platzhalter {} durch das n-te Argument der `format`-Funktion ersetzt.

Dasselbe kann man durch sogenannte **f-strings** (*formatted string literals*) erreichen. Ein `f` direkt vor dem String sorgt dafür, dass man direkt Variablen und Ausdrücke in die Platzhalter schreiben kann.

```
>>> print(f'Pi is approximately {a}.')
Pi is approximately 3.1415.
>>> print(f'Another approximation is {b}/{c}.')
Another approximation is 22/7.
```

8.3.2 Listen (`list`)

Listen (`list`) sind **veränderbare** geordnete Sammlungen von beliebigen Objekten.

```
[a, b, c, ...]
```

- Eine Liste ist eine Aneinanderreihung beliebiger Objekte, die durch Kommata getrennt sind.
- Eine Liste ist in eckige Klammern `[.., ..]` eingeschlossen.
- Elemente sind geordnet (daher indizierbar)
- Listen sind *mutable*.

Bemerkung

Obwohl Listen beliebige Objekte aufnehmen können, ist die Hauptanwendung von Listen die Verwaltung einer **variablen Anzahl** von Objekten **eines einzelnen Typs** oder eng verwandter Typen (z.B. `int` und `float`).

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> a
[3, False, 1.4, 'text']
>>> type(a)
<class 'list'>
>>> len(a)       # length of list
4
```

Zugriff auf Listenelemente geht über die Angabe des Index in eckigen Klammern.

```
list[index]
```

list und **index** sind Ausdrücke, die zu einer Liste bzw. einem Integer ausgewertet werden.

- Die Indizierung beginnt bei 0.
- Negative Indizes zählen von rechts.

Bemerkung

Eckige Klammern, die sich direkt an einen Ausdruck anschließen, sind im Allgemeinen ein Zugriff per Index auf einen *sequence type*.

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> a[0]
3
>>> a[2]
1.4
>>> a[-1]
'text'
```

Teillisten (*slices*) können extrahiert werden, indem Indizes für Start und Ende der Teilliste angegeben werden.

```
liste[start:stop]
```

Die Teilliste enthält das Element mit Index **`start`**, aber **nicht** das Element mit Index **`stop`**.

Wird einer der Indizes weggelassen, starten Teillisten am Anfang oder gehen bis zum Ende.

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> a[2:4]      # a[4] is not valid
[1.4, 'text']
>>> a[0:-2]
[3, False]
>>> a[:2]
[3, False]
>>> a[2:]
[1.4, 'text']
>>> a[:]
[3, False, 1.4, 'text']
>>> a[1:1]
[]
```

Mit dem Operator **`+`** können Listen verkettet werden.

```
list1 + list2
```

Der Operator **`in`** testet, ob ein Element in einer Liste gespeichert ist.

```
value in list
```

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> b=a+[True, 10]
>>> b
[3, False, 1.4, 'text', True, 10]
>>> True in a
False
>>> True in b
True
```

Listen sind *mutable*, d.h. sie können nach ihrer Konstruktion im Speicher bearbeitet werden, z.B. über einen Zuweisungen an einen Index einer Liste.

```
list[index] = value
```

Die Zuweisungen an den Index **index** der Liste **list** weist nicht dem Namen **list[index]** den Wert **value** zu (vergleiche *Zuweisungen*), sondern ändert in der Variablen **list** den Eintrag am Index **index** auf den Wert **value**.

Erinnerung. Strings sind *immutable*.

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> a[1] = -1
>>> a
[3, -1, 1.4, 'text']
```

```
>>> s = 'abcdefghi'
>>> s[3] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Die Objekt-Methode **pop** entfernt ein Element vom Ende der Liste und liefert dieses zurück. Das geht auch mit einem Index als optionales Argument.

```
value = list.pop()
value = list.pop(index)
```

Man kann auch das erste Auftreten eines Element aus einer Liste entfernen.

```
liste.remove(element)
```

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> v = a.pop()
>>> v
'text'
>>> a
[3, -1, 1.4]
>>> a.pop(1)
-1
>>> a
[3, 1.4]
>>> a.remove(3)
>>> a
[1.4]
```

Manche Operationen gibt es sowohl als *in-place* (verändern das Objekt) als auch als *out-of-place* (liefern ein neues Objekt zurück) Version. Ein Beispiel dafür ist das Sortieren von Listen.

Die *built-in functions* **sorted** erzeugt eine neue Liste, die die Elemente der als Argument übergebenen Liste in sortierter Reihenfolge enthält. Die ursprüngliche Liste bleibt unverändert.

```
sorted(list)
```

Die Objekt-Methode **sort** sortiert die Elemente der Liste, auf der sie aufgerufen wird.

```
list.sort
```

Beispiel

```
>>> a = [1, 3, 2, -1]
>>> sorted(a)
[-1, 1, 2, 3]
>>> a
[1, 3, 2, -1]
>>> a.sort()
>>> a
[-1, 1, 2, 3]
```

Der Zuweisungsoperator weist Variablen Referenzen auf Objekte zu.

Wenn ein *mutable* Objekt verändert wird hat das einen Effekt für alle Variablen, die dieses Objekt referenzieren.

Beispiel

```
>>> a = [1, 3, 2, -1]
>>> b = a
>>> a
[1, 3, 2, -1]
>>> b
[1, 3, 2, -1]
>>> a.sort()
>>> a
[-1, 1, 2, 3]
>>> b
[-1, 1, 2, 3]
```

8.3.3 Tupel (`tupel`)

Tupel (`tupel`) sind **unveränderbare** geordnete Sammlungen von beliebigen Objekten.

```
(a, b, c, ...)
```

- Ein Tupel ist eine Aneinanderreihung beliebiger Objekte, die durch Kommata getrennt sind.
- Ein Tupel ist in runde Klammern `(.., ..)` eingeschlossen.
- Ein Tupel mit einem einzelnen Element benötigt ein zusätzliches Komma, um es von einem eingeklammerten Wert zu unterscheiden.
- Tupel sind ähnlich wie Listen, allerdings *immutable*.

Bemerkung

Die Hauptanwendung von Tupeln ist die Verwaltung einer **festen Anzahl** von Objekten **beliebigen Typs**.

Beispiel

```
>>> a = (1, 2, -1)
>>> a[1]
2
>>> sorted(a)
[-1, 1, 2]
>>> (10)
10
>>> (10,)
(10,)
>>> a[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

8.3.4 Dictionary (`dict`)

Ein **Dictionary** ordnet hashbare Schlüssel beliebigen Objekten zu.

```
{key1: value1, key2: value2, ...}
```

- Ein Dictionary enthält Zuordnungen der Form **`key: value`**.
- **`key`** können alle *hashable objects* sein, insbesondere die Objekte der *immutable built-in types*.
- **`value`** kann beliebigen Typ haben, also auch *mutable* sein.
- Dictionaries sind *mutable*.

Auf ein Dictionary zugreifen (lesen und schreiben) kann man u.a. mit **`dict[key]`**.

Entnehmen (Löschen und Zurückgeben) von Einträgen erfolgt mit der Objekt-Methode **`dict.pop()`**.

Dictionaries können mittels der Objekt-Methode **`dict.update()`** aneinander gehängt werden. **Vorsicht** bei bereits vorhandenen Schlüsseln werden die Objekte überschrieben.

Beispiel

```
>>> d = {1: 10, 'key-text': [True, False],
... (1, 2, 3): 'value-text'}
>>> d[1]
10
>>> d['key-text']
[True, False]
>>> d[(1, 2, 3)]
'value-text'
>>> d[2] = 2
>>> d
{1: 10, 'key-text': [True, False], (1, 2, 3): 'value-text', 2: 2}
>>> d.pop('key-text')
[True, False]
>>> d
{1: 10, (1, 2, 3): 'value-text', 2: 2}
>>> e = {2:5, 'x':44}
>>> d.update(e)
>>> d
{1: 10, (1, 2, 3): 'value-text', 2: 5, 'x': 44}
```

Operationen

Die folgenden Operationen werden von den elementare Sequenz-Typen (*built-in sequence types*), wie z.B. **str** und **list**, unterstützt.

Operation	Bedeutung
x in s	**True** wenn ein Element von **s** gleich **x** ist, sonst **False**
x not in s	**False**, wenn ein Element von **s** gleich **x** ist, sonst **True**
s+t	die Verkettung von **s** und **t**
s*n	**n**-malige Verkettung von **s** (mit sich selbst)
n*s	siehe **s*n**
s[i]	Element mit Index **i** in **s**, Zählung beginnt bei 0
s[i:j]	Abschnitt aus **s** von Index **i** bis **j-1**
s[i:j:k]	Abschnitt aus **s** von **i** bis **j-1** mit Schrittlänge **k**
len(s)	Länge von **s**
min(s)	kleinstes Element von **s**
max(s)	größtes Element von **s**
s.count(x)	Anzahl der Vorkommen von **x** in **s**

8.4 Funktionen

Die meisten nicht-trivialen Programme haben Code-Teile, die mehrfach ge-

nutzt werden sollen. Es bietet sich an, diese in selbst-definierte Funktionen auszulagern.

Funktionen mit klar definiertem Verhalten, Argumenten und Rückgabewerten erhöhen die Verständlichkeit von Programmen.

Funktionen werden in Python mit dem Schlüsselwort **def** definiert.

```
1 def function_name(paramter1, parameter2, ...):
2     statement1
3     statement2
4     ...
```

Über den Bezeichner **function_name** kann die Funktion aufgerufen werden.

Die Elemente der **Parameterliste** sind ebenfalls Bezeichner, über die innerhalb der Funktion auf die übergebenen Argumente zugegriffen werden kann.

In der Funktionsdefinition folgt nach **def** der Bezeichner der Funktion, die Parameterliste in runden Klammern und ein Doppelpunkt.

In der nächsten Zeile beginnt der Code-Block. Python entscheidet durch Einrückung, welche Befehle zum Block gehören, deshalb muss jede Zeile des Blocks in gleicher Weise, d.h. auch mit denselben Zeichen, eingerückt sein.

Bemerkung. In der Python-Shell kann mit *Tab* (*Tabulator*) eingerückt werden, in Skripten ist eine Einrückung mit vier *Spaces* (*Leerzeichen*) üblich.

Bei jedem Aufruf der Funktion werden die Befehle des Code-Blocks der Reihe nach abgearbeitet. Innerhalb der Funktion kann der Befehl

```
return expression
```

verwendet werden, um die Funktion zu beenden und den Wert des Ausdrucks **expression** an den Aufrufer zurückzuliefern.

Bemerkung

Ein **return** ohne Argument liefert **None**. Eine Funktion ohne explizites **return** wird implizit um ein **return** (ohne Argument) ergänzt.

Beispiel

Datei **function-example.py**

```
#!/usr/bin/python3

def print_and_add(x, y):      # function definition
    print('x =', x)
    print('y =', y)
    return x+y
    print('line will not reached')

print(print_and_add(2, 3))    # function call
```

```
> ./function_example.py
x = 2
y = 3
5
```

Erinnerung

Eine Datei muss ausführbar sein, um sie auf der Kommandozeile zu starten.

```
> chmod u+x function-example.py
```

Alternativ kann man die Datei auch direkt an den Python-Interpreter übergeben.

```
> python3 function-example.py
```

8.5 Kontrollstrukturen

8.5.1 Konditionale (`if`)

Konditionale erlauben es, Code-Teile nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt/nicht erfüllt ist.

```
if expression:
    statements
    ...
elif expression:
    statements
    ...
elif expression:
    statements
```

```
    ...
else:
    statements
    ...
```

- Die **if**-/**elif**-/**else**-Blöcke werden von oben nach unten abgearbeitet.
- für **if**/**elif** wird der jeweilige Ausdruck **expression** ausgewertet.
  - Hat der Ausdruck **expression** den Wahrheitswert **False**, wird zum nächsten **elif**- bzw. **else**-Block gesprungen.
  - Hat der Ausdruck **expression** den Wahrheitswert **True**, werden die Befehle **statements** des Blocks ausgeführt. Danach werden keine weiteren **elif**- oder **else**-Blöcke mehr abgearbeitet.
  - Ist keine Ausdruck von **if**/**elif True**, werden die Befehle **statements** des **else**-Blocks ausgeführt.
- Der **if**-Teil ist obligatorisch, **elif** und **else** sind optional.

Beispiel

```
>>> def absolute_value(x):
...     if x > 0:
...         return x
...     else:
...         return -x
...
>>> print(absolute_value(1))
1
>>> print(absolute_value(-2))
2
```

Bemerkung

In diesem Kontext haben nicht nur Ausdrücke vom Typ **bool** einen Wahrheitswert.

Z.B. haben `0`, die leere Liste `[]` und der leere String `''` den Wahrheitswert **False**.

Als Faustregel gilt, dass *leere* Werte (oft) den Wahrheitswert **False** haben.

8.5.2 Schleifen (`for`, `while`))

Schleifen dienen dazu, eine Reihe von Befehlen mehrfach auszuführen, wobei die Anzahl der Wiederholungen selbst programmatisch geregelt werden kann.

Eine **`for`**-Schleife durchläuft ein iterierbares Objekte (*iterable*) mit einer Schleifevariable, die das aktuelle Element in der Schleife verfügbar macht.

```
for identifier in iterable:
    statements
    ...
```

- **`identifier`** ist der Bezeichner der Schleifevariable.
- **`iterable`** ist ein iterierbares Objekte.

Bemerkung

String, Listen, Tupel und Dictionaries sind *iterable*.

Beispiel

Bei Strings, Listen und Tupel erfolgt die Iteration über alle Elemente der Liste, von links nach rechts.

```
>>> s=''
>>> for c in 'short text':
...     s=s+c
...
>>> s
'short text'
>>> for val in [1, 'zwei', (3,4)]:
...     print(val)
...
1
zwei
(3, 4)
>>> sum=0
>>> for x in (1, 5, 7, 22, 4):
...     sum+=x
...
>>> sum
39
```

Beispiel

Bei Dictionaries erfolgt die Iteration über die Schlüssel, in unbestimmter Reihenfolge.

```
>>> d = {1: 'one', 2: 'two'}
>>> for key in d:
...     print('{0} => {1}'.format(key, d[key]))
...
1 => one
2 => two
```

Ein Dictionary liefert mit der Objekt-Methoden

- **keys** eine Liste aller Schlüssel,
- **values** eine Liste aller Werte,
- **items** eine Liste aller **(key, value)**-Tupel,

über die dann interiert werden kann.

Bemerkung

Listen und Tupel unterstützen Mehrfachbelegung (*multiple assignment*).

```
>>> x, y, z  = (1,2,3)
```

Beispiel

```
>>> d={1: 'one', 2: 'two'}
>>> for key in d.keys():
...     print(key)
...
1
2
>>> for value in d.values():
...     print(value)
...
one
two
>>> for key, val in d.items():
...     print('{0} => {1}'.format(key, val))
...
1 => one
2 => two
```

Ranges (Zahlenreihen) werden, ähnlich wie die Indizes bei *slices*, mit Zahlenwerte von **start** bis **stop-1** und Schrittweite **step** erzeugt.

```
range(start, stop, step)
```

- **start** und **step** sind optional.
- Ranges sind *immutable* und *iterable*.

Bemerkung

Der Vorteil von Ranges gegenüber Listen oder Tupel ist, dass ein Range-Objekt immer die gleiche (kleine) Menge an Speicher benötigt, unabhängig von der Größe des Bereichs, den es repräsentiert, da es nur die Start-, Stop- und Step-Werte speichert und die einzelnen Elemente und Teilbereiche nach Bedarf berechnet.

Beispiel

```
>>> for val in range(5):
...     print(val)
...
0
1
2
3
4
>>> for val in range(-3, 0):
...     print(val)
...
-3
-2
-1
>>> for val in range(3, 10, 2):
...     print(val)
...
3
5
7
9
```

Die **while**-Schleife wiederholt einen Block solange wie eine bestimmte Bedingung erfüllt ist.

```
while expression:
    statements
    ...
```

- Der Ausdruck **expression** wird ausgewertet.
  - Ist der Ausdruck **True** wird der Code-Block ausgeführt. Nach dem Ende des Blocks geht die Ausführung wieder in die Zeile **while ...** .
  - Ist der Ausdruck **False** geht die Ausführung hinter dem Code-Block weiter.

Bemerkung

Es kann sein, dass der Block überhaupt nicht oder fortlaufenden ausgeführt wird.

Beispiel

Euklidischer Algorithmus zur Bestimmung des *greatest common divisor* (*gcd*) zweier Zahlen n und m

Solange $n \neq m$

- Ersetze die größere der beiden Zahlen n und m durch die Differenz zwischen größerer und kleinerer Zahl.

Gilt $n = m$ ist dies der *gcd*.

Datei **gcd.py**

```
#!/usr/bin/python3
# greatest common divisor
def gcd(n, m):
    print('start')
    while n != m:
        print(f'loop: n = {n} m = {m}')
        if n > m:
            print('  n is greater')
            n = n - m
        else:
            print('  m is greater')
            m = m - n
    print(f'stop: n = m = {n}')
    return n
```

```
gcd(24, 42)
```

```
> ./gcd.py
start
loop: n = 24 m = 42
  m is greater
loop: n = 24 m = 18
  n is greater
loop: n = 6 m = 18
  m is greater
loop: n = 6 m = 12
  m is greater
stop: n = m = 6
```

8.6 Klassen und Objekte

- Klassen geben die Möglichkeit Daten mit Funktionalitäten zentral zu vereinigen.
- Eine neue Klasse ist ein neuer Typ Objekt von dem Instanzen erschaffen werden können.
- Jede Instanz (Objekt) einer Klasse ist definiert durch Attribute (Variablen), welche den internen Status verwalten.
- Jede Klasse definiert Methoden (Funktionen) mit denen Instanzen dieser Klasse ihren internen Status verändern oder in geeigneter Form nach außen geben können.

```
class ClassName:
    <Statement-1>
    .
    .
    .
    <Statement-N>
```

- Die meisten Statements sind Funktionsdefinitionen.
- Klassendefinitionen müssen ausgeführt werden bevor sie benutzt werden können.
- Das Ergebnis der Ausführung ist ein Klassenobjekt.

Klassenobjekte unterstützen zwei Arten von Operationen: Attributreferenzierung und Instanziierung.

```
class MyClass:
    i = 42

    def f(self):
        return 'hello world'
```

- Attributreferenzierung benutzt die Syntax `obj.attrName`.
- `MyClass.i` und `MyClass.f` sind valide Attributreferenzierungen.
- `MyClass.i` gibt einen Interger zurück.
- `MyClass.f` gibt eine Funktion zurück.
- Die Referenzen können benutzt werden um neue Werte zuzuweisen.
- Klasseninstanziierung benutzt die Funktions-Syntax.
- Klassenname als parameterlose Funktion gibt eine leere Instanz dieser Klasse zurück.

Im folgenden Beispiel wir eine neue Instanz der Klasse erstellt und von der variable `x` referenziert.

```
x = MyClass()
```

Diese Instanziierung erstellt ein neues leeres Objekt dieser Klasse.

- Die spezielle Funktion `__init__()` kann benutzt werden um die Instanziierung zu steuern.
- Argumente werden dabei bei der Instanziierung an `__init__()` weitergegeben.

```
>>>class Complex:
...
...   def __init__(self, realpart, imagpart):
...     self.r = realpart
...     self.i = imagpart
...
>>>x = Complex(3.0, -4.5)
x.r, x.i
(3.0, -4.5)
```

- Ein Objekt besitzt nach der Erstellung eigene Variablen (**Objekt-Attribute**) und Funktionen (**Objekt-Methoden**).

- Die Klasse definiert alle Funktionen, welche auf Objekten dieser Klasse angewendet werden können.
- Objekt-Methoden (Funktionen) werden durch das Objekt, gefolgt von einem Punkt, gefolgt von dem Namen der Funktion aufgerufen (`object.function()`).
- Objekt-Attribute (Variablen) werden durch das Objekt, gefolgt von einem Punkt, gefolgt von dem Variablennamen aufgerufen (`objekt.variable`).
- Es gibt eingebaute Methoden (*built-in functions*), z.B. **`print`** oder **`type`**, die global verfügbar sind und auf (beliebige) Objekte angewendet werden können.

Die Werte von Instanzvariabeln sind einzigartig mit jeder einzelnen Instanz (je nach Initialisierung). Klassenvariabeln sind Attribute und Methoden, die von allen Instanzen dieser Klasse geteilt werden.

```
>>>class Komplex:
...   kind = "Dies hier ist eine Komplexe Zahl."
...
...   def __init__(self, realpart, imagpart):
...     self.r = realpart         # Instanzvariable
...     self.i = imagpart         # Instanzvariable
...
>>>x = Komplex(3.0, -4.5)
>>>y = Komplex(5.0, 2.5)
>>>print(x.r)
>>>print(y.r)
>>>print(x.kind)
>>>print(y.kind)
3.0
5.0
Dies hier ist eine Komplexe Zahl.
Dies hier ist eine Komplexe Zahl.
```

Polymorphimus beschreibt in den meisten Programmiersprachen Methoden und Operatoren, welche mit dem gleichen Bezeichner auf unterschiedlichen Objekten ausgeführt werden können. In Python sind zum Beispiel die Funktionen `print()` und `len()` auf unterschiedlichen Objekten ausführbar.

```
>>>len(["Hallo", "Welt"])
2
>>>len("Hallo Welt")
10
```

Für ein Objekt einer eigenen Klasse `myobj` kann man z.B. in der Klasse mit `__str__()` eine Stringrepräsentation definieren, welche beim Aufruf von `print(myobj)` aufgerufen wird.

```
>>>class Komplex:
...   kind = "Dies hier ist eine Komplexe Zahl."
...
...   def __init__(self, realpart, imagpart):
...     self.r = realpart         # Instanzvariable
...     self.i = imagpart         # Instanzvariable
...   def __str__(self):      # Methodenname für Stringrepräsentation str(
        return "Real: " + str(self.r) + " Imag: " + str(self.i)
>>>x = Komplex(3.0, -4.5)
>>>print(x)
Real: 3.0 Imag: -4.5
```

In einer Funktionsdefinition können Default-Werte mittels `=defWert` angegeben werden.

```
>>>class Komplex:
...   kind = "Dies hier ist eine Komplexe Zahl."
...
...   def __init__(self, realpart, imagpart=0.0):
...     self.r = realpart         # Instanzvariable
...     self.i = imagpart         # Instanzvariable
...   def __str__(self):      # Methodenname für Stringrepräsentation str(
        return "Real: " + str(self.r) + " Imag: " + str(self.i)
>>>x = Komplex(3.0)
>>>print(x)
Real: 3.0 Imag: 0.0
```

Default-Werte werden meistens in Dokumentationen der Funktionen erklärt z.B. in `sort(*, key=None, reverse=False)` beschreibt `key` welcher Key zum Sortieren benutzt werden soll und `reverse` ob der Vergleichsoperator umgedreht werden soll.

- Vererbung erlaubt es einer Klasse Methoden und Attribute von einer anderen Klasse vordefiniert zu bekommen.
- Die Eltern-Klasse ist die Klasse von der geerbt wird (Base-Klasse).
- Die Kind-Klasse ist die Klasse, die von einer anderen Klasse erbt (Derived-Klasse).

Die Syntax ist dabei wie bei der Erstellung einer normalen Klasse mit dem Zusatz der Base-Klasse in runden Klammern.

```
class DerivedClassName(BaseClassName):
```

```
    <Statement-1>
    .
    .
    .
    <Statement-N>
```

Methoden und Attribute, welche in der Derived-Klasse nicht gefunden werden, werden in der Base-Klasse gesucht. Mit der Funktion `super()` kann man expliziet die Base-Klasse ansteuern, z.B. bei der Erstellung eines Objektes mit `__init__()`.

```
class Person:
  def __init__(self, firstname, lastname, age): #Konstruktor
    self.firstname = firstname #self ist Selbstreferenz
    self.lastname = lastname
    self.age = age

  def __str__(self): #Stringrepräsentation bei str(obj)
    return f"({self.firstname}, {self.lastname}, {self.age})"

  def __repr__(self): #Stringrepräsentation z.B. in Listen
    return str(self)

  def __eq__(self, other): #Gleichheits-Operator
    return self.lastname == other.lastname and \
        self.firstname == other.firstname

  def __hash__(self): #Hashfunktion z.B. für Dict-Keys
    return hash((self.firstname, self.lastname))
```

```
class Employe(Person): #Vererbung
  def __init__(self, firstname, lastname, age, id, salary):
    super().__init__(firstname, lastname, age) #Base-Klasse
    self.id = id
    self.salary = salary

  def __gt__(self, other): #Vergleichsoperator z.B. für sort()
    return self.salary < other.salary
```

8.7 Pandas

pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

Sie finden die Informationen des nächsten Kapitels und sehr viel extra Wissenswertes zum nachlesen unter `https://pandas.pydata.org/`.

Das Pandas Modul wird wie folgt in das eigene Skript geladen.

```
import pandas
```

Wir können mit dem Schlüsselwort `as` einen Alias definieren.

```
import pandas as pd
```

8.7.1 Datenstrukturen

- Eine `Series` ist ein eindimensionales gelabeltes Array, welches jeden möglichen Datentypen enthalten kann (z.B. Int, String, Float, Python-Objekte).
- Die Achsen werden als `index` bezeichnet.

Wir initialisieren eine `Series` Datenstruktur wie folgt:

```
s = pd.Series(data, index=index)
```

- `index` ist eine Liste von Labeln und muss die gleiche Länge wie `data` haben.
- `data` kann z.B. eine Liste sein, ein Skalarer Wert, ein Python-Dict oder vieles mehr.

`data` als Liste.

```
>>>pd.Series([1,2,3], index=["a", "b", "c"])
a    1
b    2
c    3
dtype: int64
```

`data` als Skalar.

```
>>>pd.Series(5, index=["a", "b", "c"])
a    5
b    5
c    5
dtype: int64
```

`data` als Python-Dict. Die Indices wählen die Werte aus. Existiert ein Wert nicht bekommen wir `nan` – not a number.

```
>>>d = {"a": 0.0, "b": 1.0, "c": 2.0}
>>>pd.Series(d, index=["b", "c", "d", "a"])
b    1.0
```

```
c    2.0
d    NaN
a    0.0
dtype: float64
```

Wir demonstrieren Zugriffe auf die Daten an folgenden Beispielen:

```
>>>s = pd.Series([1,2,3,4,5,6],
                 index=["a", "b", "c", "d", "e", "f"])
>>>s.iloc[2]    #Zugriff auf Integfer Location.
3
>>>s.iloc[2:]   #Slice
c    3
d    4
e    5
f    6
dtype: int64
>>>s[s>3]       #Boolscher Filter
d    4
e    5
f    6
dtype: int64
```

Wir demonstrieren Zugriffe auf die Daten an folgenden Beispielen:

```
>>>s = pd.Series([1,2,3,4,5,6],
                 index=["a", "b", "c", "d", "e", "f"])
>>>s.iloc[[5,3,2]]      #Liste von Zeilen-Indices
f    6
d    4
c    3
dtype: int64
>>>s["c"]           #Zugriff via Index
3
>>>s["e"] = 12.0    #Neuen Wert zuweisen.
>>>s
a     1
b     2
c     3
d     4
e    12
f     6
dtype: int64
```

Überlicherweise braucht man keine Schleifen um die Gesamtheit der Werte in einer `Series` zu ändern.

```
>>>s = pd.Series([1,2,3,4,5,6],
```

```
                  index=["a", "b", "c", "d", "e", "f"])
>>>s*2          #Jeden Wert mit 2 multiplizieren.
a     2
b     4
c     6
d     8
e    10
f    12
dtype: int64
>>>s.mean()    #Built-In Funktionen für den Mittelwert.
3.5
```

- Ein `DataFrame` bringt das Konzept einer `Series` auf ein 2-dimensionales Level.
- Wir bekommen benannte Spalten und Zeilen mit potenziell unterschiedlichen Datentypen in jeder Spalte.
- Vergleichbar zu z.B. Excel-Tabellen, SQL-Datenbanken, Dict von Series (alles was auf relationaler Algebra basiert).

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>pd.DataFrame(d, index=["d", "b", "a"])
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0
>>>pd.DataFrame(d, index=["d", "b", "a"],
                  columns=["two", "three"])
```

```
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

Wir betrachten den `DataFrame` semantisch und syntaktisch als Dict von `Series`-Objekten. So ergeben sich folgende Operationen auf `DataFrames`. Das Zugreifen und Hinzufügen von Spalten.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
        "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df["one"]    #Auswahl einer Spalte
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
>>>df["three"] = df["one"] * df["two"] #Neue Spalte
>>>df["flag"] = df["one"] > 2          #Neue Spalte
>>>df
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Das Löschen von Spalten.

```
>>>del df["two"]           #Spalte Löschen
>>>three = df.pop("three") #Spalte Löschen und Zurückgeben
>>>df
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

Das Einfügen eines Skalars.

```
>>>df["foo"] = "bar"
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
```

```
c  3.0   True  bar
d  NaN  False  bar
```

Das Einfügen einer `Series` die nicht über alle Indices wie der `DataFrame` verfügt.

```
>>>df["one_trunc"] = df["one"][:2]
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

- Spalte auswählen `df[col]` gibt eine `Series` zurück.
- Zeile durch Label auswählen `df.loc[label]` gibt eine `Series` zurück.
- Zeile durch Integer-Location auswählen `df.iloc[loc]` gibt eine `Series` zurück.
- Zeilen Slicen `df[5:10]` gibt einen DataFrame zurück.
- Zeilen durch boolschen Vektor selektieren `df[bool_vec]` gibt einen DataFrame zurück.

Lambda Funktionen sind kleine anonyme Funktionen, welche beliebig viele Argumente übergeben bekommen, aber nur einen einzigen Ausdruck auswerten. Die Syntax ist dabei: `lambda arg1, arg2, arg3 : expression`. Lambdas benutzt man meistens, wenn Funktionen als Argumente übergeben werden.

```
>>>l = lambda x1, x2: x1 + x2 + 5
>>>print(l(3,5))
13
```

Mit der Funktion `apply` können wir entlang der Spalten (Default) oder Zeilen (Parameter `axis=1` übergeben) eines DataFrames Funktionen anwenden. Die Methode wird auf einem DataFrame ausgeführt und bekommt eine Funktion (z.B. ein lambda) übergeben.

```
>>>df = pd.DataFrame([[1,2], [3,4], [5,6]], columns=['A', 'B'])
>>>df
   A  B
0  1  2
1  3  4
2  5  6
>>>df.apply(lambda x : x*x)
```

```
   A  B
0  1  4
1  9  16
2  25  36
```

Mit der Funktion `assign` können wir neue Spalten erstellen, welche potenziell aus bestehenden Spalten abgeleitet werden können. Die Funktion bekommt als Argumente Spalten-keys und deren Werte übergeben. Dabei können die Werte z.B. `Series` sein oder auch Funktionen, die neue Werte aus übergebenen Tupeln erstellen. Weiter rechts stehende Argumente können bereits auf Ergebnisse aus vorherigen Argumenten zugreifen.

```
>>>dfa = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]}
>>>dfa.assign(C=lambda x: x["A"] + x["B"],
              D=lambda x: x["A"] + x["C"])
   A  B  C   D
0  1  4  5   6
1  2  5  7   9
2  3  6  9  12
```

8.7.2 Analyse Funktionen

Mit `shape` bekommen wir ein 2-Tuple mit den Dimensionen unseres DataFrames zurück. Dabei ist `shape[0]` die Anzahl der Zeilen und `shape[1]` die Anzahl der Spalten.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
        "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.shape
(4, 2)
```

Mit den Funktionen `head()` (und `tail()`) bekommen wir einen kleinen Teil des DataFrames. Der übergebene Wert gibt dabei an wie viele Zeilen wir berücksichtigen wollen (Default-Wert 5). Mit `head(n)` (`tail(n)`) bekommen wir die ersten (letzten) n Zeilen.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
        "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
```

```
                        index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.head(2)
        one     two
a       1.0     1.0
b       2.0     2.0
```

Mit der Funktion `count()` kann die Anzahl der nicht-NA Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.count()
one    3
two    4
dtype: int64
```

Mit der Funktion `sum()` kann die Summe aller Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.sum()
one     6.0
two    10.0
dtype: float64
```

Mit der Funktion `mean()` kann der Durchschnitt aller Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.mean()
```

```
one    2.0
two    2.5
dtype: float64
```

Mit der Funktion `median()` kann der Median aller Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.median()
one    2.0
two    2.5
dtype: float64
```

Mit den Funktion `min()` (und `max()`) kann das Minimum (Maximum) aller Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.min()
one    1.0
two    1.0
dtype: float64
```

Mit der Funktion `std()` kann die Standardabweichung der Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.std()
one    1.000000
two    1.290994
dtype: float64
```

Mit der Funktion `unique()` bekommen wir aus einer Series ein Array aller einzigartigen Werte zurück. Wir werden nicht weiter auf Arrays eingehen. Für uns reicht aus, dass wir über diese wie Listen iterieren können.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 3.0, 4.0],
                      index=["a", "b", "c", "d", "e"])}
>>>df = pd.DataFrame(d)
>>>df["two"].unique()
array([1., 2., 3., 4.])
```

Mit `values` bekommen wir aus einer Series ein Array aller Werte zurück. Wir werden nicht weiter auf Arrays eingehen. Für uns reicht aus, dass wir über diese wie Listen iterieren können.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df["one"].values
array([ 1.,  2.,  3., nan])
```

Mit der Funktion `describe()` bekommt man eine statistische Übersicht (z.B. der Werte in den Spalten). Die Prozentzahlen stehen dabei für entsprechende Perzentile.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.describe()
count   3.0     4.000000
mean    2.0     2.500000
std     1.0     1.290994
min     1.0     1.000000
25%     1.5     1.750000
50%     2.0     2.500000
75%     2.5     3.250000
max     3.0     4.000000
```

8.7.3 Einlesen und Auffüllen

Eine CSV-Datei (comma-separated values) ist ein Format um Daten tabellarisch abzuspeichern.

- Eine Zeile wird mit `\n` abgeschlossen.
- Werte innerhalb der Zeile werden mit einem Delimiter/Separator (Default Wert `','`) getrennt.
- Die erste Zeile gibt meistens die Namen der Spalten an.

```
Index,one,two
a,1,1
b,2,2
c,3,3
d,,4
```

Mit der Pandas Funktion `read_csv()` können wir eine csv-Datei direkt in einen DataFrame laden.

- Das erste Argument ist der Pfad zur Datei.
- Mit `sep=...` kann man einen Separator/Delimiter spezifizieren.
- Mit `index_col="Index"` kann man einen Spalte zum Index promoten.

```
>>>df = pd.read_csv("Test.csv", sep=',', index_col="Index")
>>>df
        one     two
Index
a       1.0     1
b       2.0     2
c       3.0     3
d       NaN     4
```

Mit den Funktion `fillna()`,`ffill()` und `bfill()` können wir fehlenden Werten neue Werte zuweisen.

- Wir können z.B. mit `fillna(n)` alle fehlenden Werte durch einen Default-Wert n ersetzen.
- Den letzten/nächsten validen Werte benutzen (`ffill()` und `bfill()`).

```
>>>df = pd.read_csv("Test.csv", sep=',', index_col="Index")
>>>df.fillna(0)
        one     two
Index
a       1.0     1
b       2.0     2
```

```
c       3.0     3
d       0.0     4
```

8.7.4 Kombinieren von Daten

Die Funktion `concat()` konkateniert Series und DataFrame Objekte entlang der angegebenen Achse und kann dabei die Menge der jeweiligen Indices schneiden oder vereinigen. Die Funktion bekommt eine Liste oder ein Dictionary mit kompatiblen Typen (Series, DataFrame) übergeben.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>df2 = pd.DataFrame({"A": ["A2", "A3"],
                        "B": ["B2", "B3"],
                        "C": ["C2", "C3"],
                        "D": ["D2", "D3"],}, index=[2, 3])

>>>pd.concat([df1, df2])
   A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3
```

Mit dem Keyword `join` wird spezifiziert was mit jenen Achsen-Werten gemacht werden soll, welche nicht im ersten DataFrame vorkommen. Mit `join="outer"` wird die Vereinigung aller Werte gebildet.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>df3 = pd.DataFrame({"A": ["A1", "A3"],
                        "B": ["B1", "B3"],
                        "C": ["C1", "C3"],
                        "D": ["D1", "D3"],}, index=[1, 3])

>>>pd.concat([df1,df3], axis=1, join="outer")
   A   B   C   D   A   B   C   D
0  A0  B0  C0  D0  NaN NaN NaN NaN
1  A1  B1  C1  D1  A1  B1  C1  D1
3  NaN NaN NaN NaN A3  B3  C3  D3
```

Mit dem Keyword `join` wird spezifiziert was mit jenen Achsen-Werten gemacht werden soll, welche nicht im ersten DataFrame vorkommen. Mit `join="inner"` wird der Schnitt aller Werte gebildet.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>df3 = pd.DataFrame({"A": ["A1", "A3"],
                        "B": ["B1", "B3"],
                        "C": ["C1", "C3"],
                        "D": ["D1", "D3"],}, index=[1, 3])

>>>pd.concat([df1,df3], axis=1, join="inner")
  A  B  C  D  A  B  C  D
1 A1 B1 C1 D1 A1 B1 C1 D1
```

Es kann eine Series-Objekt als Spalte an einen DataFrame angefügt werden.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>column = pd.Series(["E0", "E1"], index=[0,1], name="E")
>>>pd.concat([df1,column], axis=1)
   A   B   C   D   E
0  A0  B0  C0  D0  E0
1  A1  B1  C1  D1  E1
```

Es kann eine Series-Objekt als Zeile an einen DataFrame angefügt werden. Dazu wird aus dem Series-Objekt mit `to_frame()` ein DataFrame-Objekt und dieses dann mit `.T` transponiert.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>row = pd.Series(["A2", "B2", "C3", "D3"],
                    index=["A","B","C","D"], name="3")
>>>pd.concat([df1,row.to_frame().T], axis=0)
         A       B       C       D
0       A0      B0      C0      D0
1       A1      B1      C1      D1
3       A2      B2      C3      D3
```

Mit `merge(left, right)` kombiniert man zwei DataFrame-Objekte wie in relationalen Datenbanken z.B. SQL.

- Spalten auswählen, welche als Keys zum Zusammenfügen benutzt werden.
- Key-Kombinationen, welche mehrfach in den Tabellen vorkommen erzeugen ein kartesisches Produkt.
- Mit `how=` wird angegeben, welche Keys in der erzeugten Tabelle vorkommen.
- Wenn ein Key in einer der Tabellen nicht vorkommt, werden entsprechende `NA` hinzugefügt.

Methoden für `how=<methode>` umfassen:

- Mit `left` werden Keys aus dem linken DataFrame benutzt.
- Mit `right` werden Keys aus dem rechten DataFrame benutzt.
- Mit `outer` werden Keys aus der Vereinigung beider benutzt.
- Mit `inner` werden Keys aus dem Schnitt beider benutzt.
- Mit `cross` wir das kartesische Produkt aller Einträge gebildet.

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                         "key2": ["K0", "K0", "K0", "K0"],
                         "C": ["C0", "C1", "C2", "C3"],
                         "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="left", on=["key1", "key2"])
        key1    key2    A       B       C       D
0       K0      K0      A0      B0      C0      D0
1       K0      K1      A1      B1      NaN     NaN
2       K1      K0      A2      B2      C1      D1
3       K1      K0      A2      B2      C2      D2
4       K2      K1      A3      B3      NaN     NaN
```

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
```

```
                          "key2": ["K0", "K0", "K0", "K0"],
                          "C": ["C0", "C1", "C2", "C3"],
                          "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="right", on=["key1", "key2"])
        key1    key2    A       B       C       D
0       K0      K0      A0      B0      C0      D0
1       K1      K0      A2      B2      C1      D1
2       K1      K0      A2      B2      C2      D2
3       K2      K0      NaN     NaN     C3      D3
```

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                         "key2": ["K0", "K0", "K0", "K0"],
                         "C": ["C0", "C1", "C2", "C3"],
                         "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="outer", on=["key1", "key2"])
        key1    key2    A       B       C       D
0       K0      K0      A0      B0      C0      D0
1       K0      K1      A1      B1      NaN     NaN
2       K1      K0      A2      B2      C1      D1
3       K1      K0      A2      B2      C2      D2
4       K2      K0      NaN     NaN     C3      D3
5       K2      K1      A3      B3      NaN     NaN
```

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                         "key2": ["K0", "K0", "K0", "K0"],
                         "C": ["C0", "C1", "C2", "C3"],
                         "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="inner", on=["key1", "key2"])
        key1    key2    A       B       C       D
0       K0      K0      A0      B0      C0      D0
1       K1      K0      A2      B2      C1      D1
2       K1      K0      A2      B2      C2      D2
```

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})
```

```
>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                         "key2": ["K0", "K0", "K0", "K0"],
                         "C": ["C0", "C1", "C2", "C3"],
                         "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="cross")
  key1_x key2_x A   B   key1_y  key2_y  C   D
0 K0     K0     A0  B0  K0      K0      C0  D0
1 K0     K0     A0  B0  K1      K0      C1  D1
2 K0     K0     A0  B0  K1      K0      C2  D2
3 K0     K0     A0  B0  K2      K0      C3  D3
...
```

8.7.5 Group by: split-apply-combine

Mit Group by wird generell ein Prozess bezeichnet, welcher ein oder mehrere der folgenden Schritte beinhaltet:

- **Splitting**: Teilen der Daten nach gegebenen Kriterien in Gruppen.
- **Applying**: Funktionen auf einzelne Gruppen anwenden.
- **Combining**: Ergebnisse in geeigneter Weise abspeichern.

Im Schritt Applying können z.B. folgende Dinge gemacht werden:

- **Aggregation**: Zusammenfassungen von den Daten der Gesamten Gruppe (z.B. Anzahl Einträge, Summe, Median).
- **Transformation**: Gruppenspezifische Berechnungen/Anpassungen (z.B. NA's gruppenweise ersetzen).
- **Filtration**: Gruppen oder Daten ausfiltern, auf Basis von Gruppeneigenschaften.

Es soll für jedes Label festgelegt werden zu welchem Gruppennamen es zugeordnet wird:

- Durch eine Funktion.
- Ein Dictionary, welches zu jedem Label einen Gruppennamen ausgibt.
- Einträge von Spalten/Zeilen, welche gleichzeitig den Gruppennamen definieren.
- Listen von oben genannten Dingen.

```
>>>speeds = pd.DataFrame([("bird", "Falconiformes", 389.0),
                          ("bird", "Psittaciformes", 24.0),
                          ("mammal", "Carnivora", 80.2),
```

```
                        ("mammal", "Primates"),
                        ("mammal", "Carnivora", 58)],
      index=["falcon", "parrot", "lion", "monkey", "leopard"],
      columns=("class", "order", "max_speed"))

>>>speeds.groupby("class")
<pandas.core.groupby.generic.DataFrameGroupBy object at ...>
>>>speeds.groupby("class").max()
        order   max_speed
class
bird    Psittaciformes  389.0
mammal  Primates        80.2
```

Iteration durch Gruppennamen und entsprechende DataFrames. Dazu das Beispiel von den vorherigen Folien:

```
>>>groups = speeds.groupby("class")
>>>for c,g in groups:
>>>... print(c)
>>>... print(g)
bird
       class           order  max_speed
falcon  bird   Falconiformes      389.0
parrot  bird  Psittaciformes       24.0
mammal
         class      order  max_speed
lion     mammal  Carnivora       80.2
monkey   mammal   Primates        NaN
leopard  mammal  Carnivora       58.0
```

Zugriff auf eine einzelne Gruppe. Dazu das Beispiel von den vorherigen Folien:

```
>>>groups = speeds.groupby("class")
>>>groups.get_group("mammal")
        class   order   max_speed
lion    mammal  Carnivora       80.2
monkey  mammal  Primates        NaN
leopard         mammal  Carnivora       58.0
```

Eine Aggregation ist eine Operation, welche die Dimension des Gruppen-Objekts reduziert. Das Ergebnis einer Aggregation ist meistens ein Wert für die gesamte Gruppe (z.B. Summe, Maximum, Minimum).

```
>>>groups = speeds.groupby("class")
>>>groups.max()
Selection deleted
groups.max()
        order   max_speed
```

```
class
bird    Psittaciformes  389.0
mammal  Primates        80.2
```

Bekannte Built-In Aggregation-Funktionen:

- `count()` berechnet die Anzahl nicht-NA Einträge jeder Gruppe.
- `max()` berechnet das Maximum in jeder Gruppe (analog Minimum).
- `mean()` berechnet den Durchschnitt der Werte in jeder Gruppe.
- `median()` berechnet den Median der Werte in jeder Gruppe.
- `nunique()` berechnet die Anzahl einzigartiger Werte in jeder Gruppe.
- für viele weitere Built-In Funktionen kann man in die Dokumentation schauen.

Transformationen sind Group by Operationen, bei denen die ursprüngliche Indexierung beibehalten wird und nicht die neue Indexierung der Gruppennamen benutzt wird.

```
>>>groups = speeds.groupby("class")["max_speed"]
>>>groups.bfill()
falcon     389.0
parrot      24.0
lion        80.2
monkey      58.0
leopard     58.0
Name: max_speed, dtype: float64
```

Bekannte Built-In Transformationen-Funktionen:

- `bfill()` berechnet Werte für NA Werte innerhalb der Gruppe (analog `ffill()`).
- `cummax()` berechnet das kummulative Maximum in jeder Gruppe (analog Minimum).
- `cumsum()` berechnet die kummulative Summe in jeder Gruppe.
- `diff()` berechnet die Differenz benachbarter Werte.
- für viele weitere Built-In Funktionen kann man in die Dokumentation schauen.

Filtrationen sind Group by Operationen, bei denen ganze Gruppen, Teile von Gruppen oder beides ausgefiltert werden. Eine Filtration gibt eine gefilterte Version des aufrufenden Objektes zurück.

Bekannnte Built-In Funktionen sind:

- `head()` wählt die oberste(n) Zeile(n) jeder Gruppe aus.
- `tail()` wählt die unterste(n) Zeile(n) jeder Gruppe aus.
- `nth()` wählt die n-te Zeile jeder Gruppe aus.

```
>>>speeds.groupby("class").nth(1)
         class           order  max_speed
parrot    bird  Psittaciformes       24.0
monkey  mammal        Primates        NaN
```

Mit der `filter()` Funktion kann eine Nutzerdefinierte Funktion bekommen und ganze Gruppen auf Grundlage dieser Funktion zu `True` oder `False` evaluieren. Alle zu `True` evaluierten Gruppen werden zurückgegeben.

```
>>>sf = pd.Series([1, 1, 2, 3, 3, 3])
>>>sf.groupby(sf).filter(lambda x: x.sum() > 2)
3    3
4    3
5    3
dtype: int64
```