Die Einführung in Python an der Uni Tübingen

von

Nico Steinle

Dieses Script ist für den persönlichen gebrauch gedacht

und dokumentiert nur den Inhalt des Kurses

2020 - 2021
V0.2

Contents

1 Video 01

Das erste Video ist über das installieren von Python. Deshalb ignoriert.

2 Video 02 Einführung und Werkzeuge

Inhalte des Kurses:

1. Was ist Python?
2. Grundlagen
3. Variablen und Datentypen
4. Datenstrukturen
5. Kontrollstrukturen
6. Funktionen
7. Objektorientierte Programmierung
8. Datenanalyse und -visualisierung
9. Interaktive Dokumente mit Jupyter Notebook

2.1 Was ist Python?

Von dem niederländischen Programmierer Guido van Rossum 1989 entwickelt.
Die Geschíchte:
Python 1.0 (Erste Vollversion):Januar 1994
Python 2.0:Oktober 2000
Python 3.0:Dezember 2008
Python 2.7.18 (Letzte 2.x Version, keine weitere Unterstützung):20.April 2020
Python 3.9.0 (neuster Version):5.Oktober 2020

Warum Python?
Es ist eine sehr übersichtliche Sprache mit klarer Syntax was es gut für Programmieranfänger macht.
Python kann durch Pakete erweitert werden was flexibel macht. Es gibt ein große Nutzergemeinde und es ist eine der wichtigsten Sprachen in Forschung und Wirtschaft.
In Ranglisten für Programmiersprachen ist es immer sehr weit oben.

2.2 Grundlagen

1 + 1
2
2 + 2
4
Man kann durch ein \die Eingabe über mehrere Zeilen aufteilen.
1 + \
... Python wartet nun auf die erneute Eingabe des Benutzers
... 1
2

Zum erstellen von Python Files reicht ein normaler Texteditor. Es wird aber ein Code editing Programm empfohlen das die Syntax hervorhebt und mit Code completion arbeitet. Was das arbeiten mit Python deutlich einfacher gestaltet.

Warum ist das Einrückungen wichtig sind:
Viele Programmiersprachen verwenden etwa Klammern, um Code in Blöcken zu organisieren. Solche Blöcke können beispielsweise Schleifen sein, in denen man die gleiche Operation für jeden Bestandteil einer Sammlung von Werten ausführen lässt. Hier ist ein Beispiel in R: Es gibt die Zahlen von eins bis zehn nacheinander aus:

```
for(i in 1:10){
    print(i)
}
```

Die R Klammern verwendet um den Code zu strukturieren, könnte man das ganze Stück auch linksbündig (und/oder in einer einzigen Zeile) schreiben und es würde immer noch funktionieren:

```
for(i in 1:10) {print(i)}
```

Für Python sind solche Einrückungen nötig, denn sonst kann es die Rang- und Reihenfolge der verschiedenen Codesesegmente nicht korrekt bewerten. Auch muss die Anzahl der Einrückungen pro Code-Level konstant bleiben. Normalerweise verwendet man vier Leerstellen; Sie sollten den Tabulator hier vermeiden:

```
for i in range (1, 11):
    print(i)
```

Dadurch, dass Blöcke eingerückt und weniger Klammern verwendet werden, wird der Code aber auch leichter lesbar.

3 Video 03

3.1 Arithmetische Operationen: Python als Taschenrechner

Addieren	+
Subtrahieren	-
Multiplizieren	*
Dividieren	/
Potenzieren	**
Teilungsrest (Modulus)	%
Ganzzahldivision	//

3.2 Hello World!

```
print('Hello World!')
```

Was passiert hier? In dieser kurzen Zeile sehen wir bereits einige wichtige Aspekte von Python und auch anderen Programmiersprachen sowie einen der augenfälligsten Unterschiede zwischen Python2 und Python3:
Es gibt Funktionen. Funktionen erlauben es Ihnen, Code für eine bestimmte Aufgabe zu schreiben und diesen dann immer wieder zu verwenden. Viele Funktionen haben Parameter, d.h., Sie können Objekte wie etwa die Phrase "Hello World!" oder eine Zahl als Argumente an die Funktion weitergeben und von der bearbeiten lassen. Man kann die bereits in Python integrierten Funktionen nutzen (wie hier print()) oder eigene Funktionen schreiben.

Es gibt verschiedene Datentypen. "Hello World!" ist eine Zeichenkette (character string), also eine Abfolge von Buchstaben, Leerzeichen, Satzzeichen, oder auch Ziffern. Es steht deshalb in Anführungszeichen, während etwa eine Zahl wie 12345 ohne Anführungszeichen auskommt.
Beachten Sie folgenden Unterschied zwischen Python2 und 3: In Python3 ist print eine Funktion und der ausgebende Text steht in Klammern: print("Hello World!") In Python2 war print eine Anweisung und verwendete deshalb keine Klammern:

```
print 'Hello World!'
```

3.3 Variablen

Variablen sind ein wichtiges Werkzeug, wenn man mit Python (und anderen Sprachen arbeitet):
$a = 1$
In diesem Fall haben wir ein sog. Objekt(object) erzeugt (dazu später mehr) und einer Variable(variable)zugewiesen.
Eine Variable ein Verweis auf einen gespeicherten Wert. Das kann ein einzelnes Zeichen oder ein ganzer Datensatz sein. Sie können sich den Wert der Variable anzeigen lassen, indem Sie einfach deren Namen in die Konsole eingeben.
Der Wert einer Variablen kann jederzeit geändert werden. Man kann auch der gleichen Variable einen zweiten Namen zuweisen.
Hier wird kein zweites Objekt erzeugt, sondern nur ein zweiter Verweis. Sie können das sehen, wenn Sie sich die ID des Objektes mit der *id*() Funktion anzeigen lassen. Diese IDs werden sich von Computer zu Computer und Sitzung zu Sitzung unterscheiden.
Wenn sich der Wert einer Variable ändert, ändert sich auch die ID. Es wird quasi ein neu erzeugt.

```
a=300
b=300
c=30
d=30
```

Danach noch die id() a-d anzeigen lassen.

```
>>> id(a)
2405776730096
>>> id(b)
2405776730320
>>> id(c)
2405775600848
>>> id(d)
2405775600848
```

Als Sie Python gestartet haben, hat Python für einen kleinen Zahlenbereich bereits Objekte erstellt (ganze Zahlen von -5 bis 256) und verwendet sie dann bei der Ausführung Ihres Codes. Python kann dadurch etwas effizienter arbeiten.

3.3.1 Variablennamen: Regeln

Variablennamen können folgende Zeichen beinhalten:

1. Buchstaben
2. Zahlen
3. Unterstrich

Es bestehen zwei wichtige Einschränkungen: Variablennamen dürfen nur mit einem Unterstrich oder einem Buchstaben beginnen.
Ein weiterer wichtiger Punkt: Python achtet bei Variablennamen auf Groß- und Kleinschreibung.

3.3.2 Variablennamen: Format

Aus technischer Sicht steht es ihnen frei, wie Sie Variblennamen gestalten, solange Sie Namen, die aus mehreren Teilen(Wörtern, Zahlen) bestehen, zusammenschreiben. So können sie etwa eine Variable, die sich auf die Durchschittstemperatur für Mai 2020 bezieht, folgendermaßen schreiben:
DURCHSCHNITTSTEMPERATURMAI2020

Das ist allerdings schwer zu lesen. Bessere Alternativen sind:
Der erste Buchstaben in jedem Wort wird groß-, der Rest kleingeschrieben (Pascal Case):
DurchschittTemperaturOktober2019

Wie bei Pascal Case, nur wird der erste Buchstabe des Namens kleingeschrieben (Camel Case):
durchschnittTemperaturOktober2019

Oder das Variablennamen immer klein geschrieben und Wörter durch einen Unterstrich getrennt werden (Snake Case):

```
durchschnitt_temperatur_oktober_2019
```

Die Wahl bleibt letzendlich Ihnen überlassen, solange Sie konstant sind. Wenn Sie mit anderen an einem Projekt arbeiten, fragen Sie, ob es interne Richtlinien gibt.

3.3.3 Schlüsselwörter (keywords)

Diese Liste ändert sich mit den Versionen. Man kann die aktuelle Liste mit help(”keywords”) anzeigen lassen.

3.3.4 Variablennamen: Was noch zu beachten ist

Verwenden Sie klare und eindeutige Namen:
DurchschittTemperaturOktober2019 statt xy

Sie können zwar die Namen von existierenden Funktionen als Variablennamen benutzen, sollen es aber nicht tun, um Verwirrungen zu vermeiden.

3.4 Datentypen

Geben Sie den folgenden Code in die Konsole ein:

```
1 + 1
1 + "a"
"a" + "b"
3 * "abc"
```

```
>>> 1 + 1
2
>>> 1 + "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> "a" + "b"
'ab'
>>> 3* "abc"
'abcabcabc'
```

Der Grund für den Fehler ist, dass es in Python verschiedene Datentypen gibt, mit denen man unterschiedliche Sachen machen kann und die etwa von arithmetischen Operatoren unterschiedlich behandelt werden.
In diesem Fall haben Sie versucht, den Operator + mit zwei verschiedenen Datentypen (int[integer] und str[string]) zu verwenden, was dieser nicht erlaubt.

3.4.1 Boolesche Werte

Ein bool (boolean) kann nur einen von zwei Werten haben: True (Wahr) oder False (Falsch). True wird mit 1 gleichgesetzt und False mit 0. Dieser Typ kommt etwa zum Einsatz, wenn Sie testen, ob der Wert einer Variablen eine bestimmte Eigenschaft hat.
In [2]:

```
a = "Hallo"
a.isalpha()   #Beinhaltet a nur Buchstaben?
```

Out[2]: True
In [3]:

```
a.isanum()    #Beinhaltet a nur alphanumerische Zahlen?
```

Out[3]: True
In [4]:

```
a.isdigit()   #Beinhaltet a nur Zahlen?
```

Out[4]: False

3.4.2 Zeichenketten

Zeichenketten, oder character strings/strings, sind Aneinanderreihungen von Zeichen (Buchstababen, Zahlen, Satzzeichen, Leerstellen, usw.) von beliebiger Länge. Sie werden bei der Definition immer in Anführungszeichen (normalerweise doppelt) gesetzt:

```
>>> a = "Hello World!"
```

Wenn eine Zeichenkette ein Anführungszeichen enthalten soll, haben Sie zwei Möglichkeiten:
Sie setzen vor das Anführungszeichen einen backslash. Dieser funktioniert als sog. Escape-Zeichen (escape character):

```
>>> a = "Sie sagt \"Hallo\""
```

Wenn Sie ein doppeltes Anführungszeichen in die Zeichenketten einfügen wollen, umschließen Sie die ganze Kette mit einfachen Anführungszeichen und umgekehrt:

```
>>> a = 'Sie sagt "Hallo"'
```

3.4.3 Integer

Ein Iteger ist eine ganze Zahl, also eine Zahl ohne Nachkommastellen. Geben Sie die folgende Zeile ein, um der Variable a den Wert 1 zuzuweisen und lassen Sie sich dann mit type() den Datentyp anzeigen.

```
>>> a = 1
>>> type(a)
<class 'int'>
```

3.4.4 Gleitkommazahl

1.5 ist eine sogenannte Gleitkommazahl (float. oder floating-point number), d.h., eine Zahl, die auch Nachkommastellen hat. Bitte beachten Sie, dass Python nicht – wie im Deutschen üblich – ein Komma verwendet, um Nachkommastellen zu signalisieren, sondern einen Punkt.

3.4.5 Komplexe Zahl

Komplexe Zahlen seien hier nur der Vollständigkeit halber erwähnt.

```
>>> a = 2 + 2j
>>> type (a)
<class 'complex'>
```

j ist eine imaginäre Zahl mit der Eigenschaften $j^2 = -1$. Die Verwendung von j stammt aus den Ingenieurswissenschaften, wohingegen Mathematiker i verwenden, um den Imaginärteil zu kennzeichnen.

3.5 Vorsicht bei Berechnungen

Computer speichern und bearbeiten Zahlen im Binärformat, das nur 0 und 1 kennt. So wird etwa die Zahl 7 als 111 dargestellt:
$7_{10} = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 111_2$
Das funktioniert meist ganz gut, sorgt aber etwa bei Zahlen mit unendlich vielen Nachkommastellen (etwa 1/3, also 0,333333333333...) oder bei bestimmten Brüchen für Probleme, da diese nur annähernd im Binärformat dargestellt und gespeichert werden können.
Hier eine gute Seite für die Umrechnung von Zahlen in verschiedene Formate:
https://www.matheretter.de/rechner/zahlenkonverter

3.6 Zur Zahlendarstellung

Seit kurzem (Python 3.6) können Sie Unterstriche verwenden, um Ziffern zu gruppieren und große Zahlen dadurch leichter lesbar zu machen. Das heißt, 1000000 (eine Million) kann auch so geschrieben werden: 1_000_000.
Zahlen werden in Python normalerweise im Dezimalformat dargestellt. Indem man einer Zahl eines von mehreren Präfixen voranstellt, kann man auch ein anderes Format verwendet werden:

Präfix	System	Beispiel	Entspricht im Dezimalsystem
0b	Binär	0b1111011	123
0o(Null o)	Oktal	0o173	123
0x	Hexadezimal	0x7B	123

Sie können diese alternativen Formate auch bei Berechnungen verwenden:

```
>>> 0xA * 0b1010 # 10 (Hexadizamal) * 10 (Binär)
100
```

4 Video04

4.1 Kommentare

Wenn Sie in Ihrem Python-Code Erklärungen für andere Nutzer oder Erinnerugsstüzen für sich selbst einfügen möchten, können Sie das mit Kommentaren (comments) machen. Diese Kommentare werden bei der Ausführung Ihres Codes ignoriert. Kommentare sind daher auch nützlich, wenn Sie verschiedene Versionen etwa einer Funktion ausprobieren möchten. Die jeweils nicht benötigten Versionen können dann auskommentiert werden.
Kommentare beginnen immer mit "#". Man kann einem Kommentar entweder an das Ende einer Zeile Code oder, was vorzuziehen ist, in seine eigene Zeile schreiben.

OK:
a = 1 # Das ist ein Kommentar.
Besser:
Das ist auch ein Kommentar.

Es gibt in Python nicht, wie in anderen Sprachen, sogenannte Blockkommentare, mit denen man längere Codeblöcke auf einmal auskommentieren kann. Stattdessen müssen alle Zeichen individuell kommentiert werden. Viele Editoren erlauben es Ihnen aber, Blöcke auszuwählen und dann alle darin enthaltenen Zeilen auf einmal auszukommentieren und die "#" dann auch wieder zu entfernen.
In Spyder können Sie etwa folgendermaßen vorgehen:

Zeilen markieren → Klick auf die rechte Maustaste → "Comment/Uncomment" auswählen.

4.2 Docstrings

Später im Kurs werden wir noch Docstrings (document strings) kennenlernen. Diese dienen zur Dokumentation von Funktionen, Methoden, Klassen und Modulen (was das ist, werden wir später noch sehen), während Kommentare eher zu Erklärung von kleinen Code-Fragmenten dienen. Docstrings beginnen und enden mit drei doppelten Anführungszeichen(""") und können über mehrere Zeilen gehen. Beispiel anhand einer Funktion "addieren", die zwei Zahlen addiert:

```
def addieren(a,b):
    """Addiert zwei Zahlen und gibt die Summe zurück."""
    return a + b
```

Wenn Sie dann später mehr über diese Funktion wissen möchten, können Sie sich die Docstrings anzeigen lassen. help(Funktion) gibt dann bsp den gespeicherten docstring aus.

4.3 Unicode

Alle Daten, also auch die Zeichen, die Sie auf Ihrem Bildschirm sehen, werden als Bits (0 oder 1) gespeichert. Anfangs was Speicherplatz extrem teuer und man versuchte wann immer möglich Speicher zu sparen. Aus diesem Grund wurden auch Jahreszahlen oft nur zweistellig gespeichert, was zum Ende des 20.Jahrhunderts einer der Gründe für die aufkommende Angst vor dem Jahr-2000-Problem (Millenium Bug, Y2K-Bug) war. Der befürchtete weitgehende Zusammenbruch digitaler Infrastrukturen blieb jedoch aus.

Zeichensatztabellen (codepages), in denen Zeichen und ihre Bytewerte gespeichert werden, hatten ursprünglich nur wenige Zeichen. ASCII (American Standard Code for Information Interchange) etwa verwendete anfanhs 7 bits, d.h., es können 2^7, oder 128, Zeichen dargestellt werden. Dazu gehören auch Steuerzeichen wie Tabulator.

Spätere Tabellen wurden auf 8 bit (auch eine weitere Form von ASCII) vergrößert, um Sonderzeichen wie etwa die deutschen Umlaute berücksichtigen zu können.

Da sich im Laufe der Zeit immer mehr Tabellen entwickelten, um mehr Sprachen abdecken zu können, musste man immer angeben, welche Tabellen in einer Datei verwendet wurde.

Jedem einzelnen Zeichen immer mehr Speicher zur Verfügung zu stellen hilft auf die Dauer auch nicht, da dadurch in den meisten Fällen Speicher verschwendet wird.

Unicode Lösung: Eine einzelne Tabelle mit jedem bekannten Zeichen und unterschiedliche langen Byte-Sequenzen.

Unicode verwendet verschieden Formate, um Zeichen zu kodieren. Das gebräulichste ist UTF-8, welches bis zu 7Bytes pro Zeichen verwendet (für die ASCII-Zeichen benötigte man nur 1Byte, oder 8bit).

Seit Python3 werden Zeichenketten automatisch als Unicode gespeichert. Sie können also auch Zeichen wie Schriftzeichen aus verschiedenen asiatischen Sprachen oder Emojis direkt in Zeichenketten verwenden.

4.4 Datenstrukturen: Listen

Pythons eingebaute Datenstrukturen erlauben es Ihnen aber auch, mehrere Werte auf einmal abzuspeichern. Ein einfaches Beispiel sind Listen (lists)

```
a = ["apfel", "banane", "banane", "kirsche"]
b = [1, 2, 4, 5, 8, 17]
```

Listen können auch mehrere Datentypen enthalten:

```
a = [1, 2, 4, "apfel", "kirsche", "banane"]
Listen können auch andere Listen enthalten:
b = [1, 2, [3, 4, 5]]
```

Sie können über den Indexwert eines Wertes, d.h. dessen Position in der Liste, auf diesen zugreifen.
Bitte beachten Sie, dass in Python wie in fast allen anderen Programmiersprachen die Zählung bei 0, nicht bei 1, beginnt.

```
a = [1, 2, 8, 17, 5, 4]
erster_wert = a[0]
letzter_wert = a[-1]
```

Es gibt auch eine Vielzahl von nützlichen Funktionen zu Listen, wie etwa sum(), mit der Sie Werte in einer Liste aufaddieren können:

```
a = [1, 2, 8, 17, 5, 4]
b = sum(a)
b
37
```

Auch können Sie etwa Werte in Listen zählen lassen:

```
a = [1, 2, 4, "apfel", "kirsche", "banane"]
a.count("banane")
2
```

Es gibt verschiedene Möglichkeiten, Listen sortieren zu lassen:

```
a = [1, 2, 8, 17, 5, 4]
a.sort()
a
[1, 2, 4, 5, 8, 17]
```

Beachten Sie, dass wir hier die sortierte Liste nicht einer neuen Variable zuweisen. sort() sortiert die Liste in-place, d.h., die Originalliste wie verändert. Wenn Sie die ursprüngliche Liste behalten und stattdessen eine neue, sortierte Liste erzeugen wollen, verwenden Sie sorted():

```
c = sorted(a)       # Die Originalliste a würde nicht verändert werden.
c
[1, 2, 4, 5, 8, 17]
```

Die Funktion sortiert Großbuchstaben vor Kleinbuchstaben. Wenn man das nicht will muus man einen key angeben nach dem es dann sortiert wird.

```
a = ["kirsche", "apfel", "Banane"]
b = sorted(a, key=str.lower)
b
['apfel', 'Banane', 'kirsche']
```

Key wirkt sich hier nicht auf den in der Liste gespeicherten Wert aus, nur darauf, wie er während der Sortierung bahandelt wird.
Wenn Sie zwei Listen zusammenfügen möchten, können Sie das mit dem +− Operator erreichen:

```
a = [1, 2]
b = [3, 4]
c = a + b
c
[1, 2, 3, 4]
```

Listen gehören zu den veränderbaren, oder mutablen (mutable) Datenstrukturen. Das bedeutet, dass sie nach ihrer Erzeugung verändert werden können. Im folgenden Beispiel wird das erste Element der Liste a (”banane”) durch ”birne” ersetzt:

```
a = ['banane', 'ananas', 'apfel', 'kirsche', 'apfel']
a[0] = 'birne'
a
['birne', 'ananas', 'apfel', 'kirsche', 'apfel']
```

Genauso können Sie auch mehrere Werte auf einmal ändern. Im folgenden Beispiel werden der zweite und dritte Wert der Liste durch zwei andere Werte erstzt.

```
a = ['banane', 'ananas', 'apfel','kirsche', 'apfel']
a[1:3] = ['erbse', 'gurke']
a
['banane', 'erbse', 'gurke', 'kirsche', 'apfel']
```

1 steht hier für den ersten Wert der Verändert wird und 3 für den ersten Wert der gleich bleibt.

Mit append() können Sie einzelne Werte an eine Liste anfügen. Falls mehrere Werte (etwa aus einer Liste) dazukommen sollen, verwenden Sie extend():

```
a = ['banane', 'ananas', 'apfel','kirsche', 'apfel']
a.append('erbse')
a
a = ['banane', 'ananas', 'apfel','kirsche', 'apfel', 'erbse']

a.extend(['gurke', 'tomate'])
a
a = ['banane', 'ananas', 'apfel','kirsche', 'apfel', 'erbse', 'gurke', 'tomate']
```

So wie Sie bestimmte Werte aus einer Liste auswählen zu können, so können Sie auch einzelne oder mehrere Werte aus einer Liste löschen:

```
a = ['banane', 'ananas', 'apfel','kirsche', 'apfel', 'erbse', 'gurke', 'tomate']
del a[0] # Löscht den ersten Wert, a hat jetzt 7 Werte.
a
del a[1:3] # Löscht den zweite und dritte Wert. a hat jetzt 5 Werte.

del a[0:5:2] # Löscht ersten, dritten, fünften usw. Wert. a hat jetzt 2 Werte.
a
['apfel', 'gurke']
```

4.5 Tupel

Im Vergleich zu Listen haben Tupel (tupels) zwar eine ähnliche Struktur, bieten aber weniger Flexibílität, da sie unveränderbar (immutabel, immutable) sind. Tupel werden mit runden Klammern erzeugt:

```
a = ('banane', 'ananas', 'apfel', 'kirsche', 'apfel')
```

Werte in einem Tupen können wieder über ihre Indexwerte adressiert werden:

```
a = ('banane', 'ananas', 'apfel', 'kirsche', 'apfel')
a[0]
'banane'

a[0:3]
('banane', 'ananas', 'apfel')
```

Was passiert wenn Sie versuchen, einen Wert aus dem Tupel zu löschen?
Operationen, welche ein Tupel verändern würden, führen zu Fehlermeldungen.
Bei der Erzeugung eines Tupel können Sie die runden Klammern auch weglassen

```
person = 'karl', 'theodor', 'mustermann'
```

Das Zuweisen von Werten zu einem Tupel heißt Tuple Packing. Beim Tuple Unpacking können Sie dann jeden der Werte im Tupel einer Variablen zuweisen und so dann gezielt aufrufen. Auf diese Weise kann man auch in einer Zeile mehrere Variablen erzeugen. Auch beim Tuple Unpacking können die runden Klammern weggelassen werden.

```
vorname, patenname, nachname = person
vorname
'karl'
```

4.6 String-Formattierung

Wenn Sie Werte, die Sie etwa in einer Liste oder einem Tupel gespeichert haben, zu einem Satz zusammenfügen wollen, steht Ihnen die join()-Funktion zu Verfügung:

```
a = ['Das', 'ist', 'ein', 'Satz']
b = " ".join(a)
b
'Das ist ein Satz'
```

Wenn Sie in den Anführungszeichen kein Trennzeichen angeben, werden die Werte ohne Unterbrechung zusammengesetzt:

```
a = ['https://', 'www', '.', 'uni-tuebingen', '.', 'de']
b = "".join(a)
b
'https://www.uni-tuebingen.de'
```

Bei der Ausgabe von Werten kann man diese etwa in Sätzen einfügen. Dazu bietet Python die format()-Funktion an. Dabei werden geschweifte Klammern als Platzhalter eingesetzt, die dann durch die Werte ersetzt werden:

```
"Heute ist {}, der {}.{}.{}.".format("Mittwoch", 6, 11, 2019)
'Heute ist Mittwoch, der 6.11.2019.'
```

Die Reihenfolge, in der die werte dabei eigesetzt werden, richtet sich dabei erst einmal danach, wie sie an format() übergeben werden. Sie können jedem Platzhalter auch einen Namen geben und die Werte dann dadurch adressieren:

```
"Heute ist {wochentag}, der {tag}.{monat}.{jahr}.".format(wochentag="Mittwoch", tag=6, monat=11, jahr=2
'Heute ist Mittwoch, der 6.11.2019.'
```

Soll das Ergebis selbst geschweifte Klammern enthalten, verwenden Sie doppelte Klammern:

```
"Das sind {g} {k}{{}} und das sind {r} {k}().".format(g="geschweifte", r="runde", k="klammern")
'Das sind geschweifte Klammern{} und das sind runde Klammern().'
```

Wie Sie sehen, kann ein Platzhalter auch mehrere Male verwendet werden, wenn Sie ihm einen Namen zugewiesen haben.

Wenn Sie einen der eingeführten Werte noch ändern müssen (etwa eine Zahl runden), kann dies als Teil das Formatierungsprozesses geschehen. Hier etwa wird eine Zahl (von Typ float) mit fünf Nachkommastellen auf zwei Nachkommastellen gerundet, um einen Eurobetrag angeben zu können:

```
a = 36.85739
"Die durchschnittlichen Kosten betrugen {:.2f} Euro.".format(a)
```

Es können auch Werte aus Listen und Tupel in strings einfügen.

```
datum = ["Mittwoch", 6, 11, 2019]
"Heute ist {wert[0]}, der {wert[1]}.{wert[2]}.{wert[3]}."format(wert=datum)
'Heute ist Mittwoch, der 6.11.2019.'
```

4.7 Mengen

Mengen, oder sets, haben die Eigenschaft, dass in ihnen jedes Element nur einmal vorkommen darf. Es gibt zwei verschiedene Methoden, um sets zu erzeugen:
set() erzeugt eine mutable Menge, d.h., eine Menge, die nachträglich verändert werden kann. Hier wird ein neues set mit Werten aus einem tuple befüllt, daher die doppelten runden Klammern.

```
a = set(('a', 'b', 'c'))
a
{'a', 'b', 'c'}
```

frozenset() erzeugt eine immutable Menge, d.h., sie kann nachträglich nicht mehr verändert werden.

```
a = frozenset(('a','b','c'))
a
frozenset({'a', 'b', 'c'})
```

Was passiert, wenn Sie versuchen, mehrmals den gleichen Wert in ein set zu packen?

```
a = (1, 2, 2, 3, 4, 5, 5, 5, 5)
b = set(a)
b
{1, 2, 3, 4, 5}
```

Jeder Wert, der mehrfach vorhanden ist, wird auf eine Erwähnung reduziert.
Was passier wenn Sie versuchen, einmal ein Tupel und dann eine Liste in ein Set eizufügen? Ein Beispiel:

```
a = set((1, (3,4))) # Zweites Element ist ein Tupel
a
{(3, 4), 1}
Das gleiche mit einer Liste statt einer Tupel gibt eine Fehlermeldung aus.
```

Der Grund für die Fehlermeldung ist, dass sets keine mutablen Elemente, wie etwa Listen, enthalten können; man könnte sonst nicht sicherstellen, dass es später zu keinen duplizierten Werten kommt. Tuples verursachen keine Probleme, da sie immuatable sind.
Dafür sind sets selbst aber mutabel, wenn sie mit set() und nicht mit frozenset() erzeugt wurden.

```
a = set((1, 2, 3, 4))
a.add(5)
a
{1, 2, 3, 4, 5}
```

4.8 Operatoren und Methoden für Mengen

Operator	Beispiel	Erklärung	Werte	Äqui Methode
in	x in a	Ist Wert x in der Menge a?	T/F	valente
not in	x not in a	Ist Wert x nicht in der Menge a?	T/F	
<=	a <= b	Ist a eine Teilmenge von b?	T/F	a.issubset(b)
<	a < b	Ist a eine echte Teilmenge von b, d.h., es gibt	T/F	
		mindestens ein Element in b, das nicht in a ist	T/F	
>=	a >=b	Ist b eine Teilmenge von a?	T/F	a.issuperset(b)
>	a > b	Ist b eine echte Teilmenge von a?	T/F	

Beispiel:

```
a = set([1, 2, 3, 4, 5, 6])
b = 5
if b in a:
    print("b ist in a")

b ist in a
```

4.9 Operatoren und Methoden für Mengen II

```
Operator   Beispiel    Erklärung
|          a|b         Neue Menge, die alle Werte von a und b enthält
&          a&b         Schnittmenge von a und b
-          a-b         Menge mit allen Elementen aus a, außer denen die auch in b sind
^          a^b         Menge mit Elementen, die entweder in a oder b sind, aber nicht in beiden
```

Beispiel:

```
a = set([1, 2, 3, 4, 5, 6])
b = set([5, 6, 7, 8, 9, 10])
c = a&b
c
{5, 6}
```

5 Video 05

5.1 Kontrollstrukturen

Bevor wir mit der nächsten Datenstruktur weitermachen, sollte wir über Kontrollstrukturen (control structures) sprechen.
Kontrollstrukturen bestimmen den Ablauf einer Programms. Fallunterscheidungen (conditional statements) führen abghängig davon, ob eine oder mehrere Bedingungen erfüllt sind oder nicht, unterschiedlichen Code aus.
Schleifen (loops) führen den gleichen Code immer wieder aus.
Schleifen und Fallunterscheidungen können auch miteinander kombiniert werden.

5.2 Kontrollstrukturen: Fallunterscheidungen mit if

if testet, ob eine Bedingung erfüllt ist. Wenn ja, wird der unter if eingerückte Code ausgeführt. Wenn die Bedingung nicht erfüllt ist, geht Python zur nächsten nicht eingerückten Zeile weiter:

```
a = 1
if a == 1:
   print("a hat den Wert 1")
print("Hier gehts weiter!")
```

5.3 Exkurs im Exkurs

5.3.1 Vergleichsoperatoren

Sie haben eben schon einen von Pythons Vergleichsoperatoren gesehen, das doppelte Gleichheitszeichen (==). Damit können Sie testen, ob die Werte auf beiden Seiten identisch sind. Hier ist eine Liste mit allen Vergleichsoperatoren:

Operator	Beispiel	Erklärung
==	1 == 2	Ist 1 gleich 2?
! =	1! = 2	Ist 1 ungleich 2?
<	1 < 2	Ist 1 kleiner als 2?
>	1 > 2	Ist 1 größer als 2?
<=	1 <= 2	Ist 1 kleiner oder gleich 2?
>=	1 >= 2	Ist 1 größer oder gleich 2?

5.3.2 Logische Operatoren

Durch Nutzung mehrerer logischer Operatoren können Sie zwei oder mehr Vergleiche kombinieren:

Operator	Beispiel
and	$(a < b)$ and $(c < d)$

Ist 1 kleiner als 2 gleichzeitig 3 kleiner als 4? Beide müssen wahr sein.

Operator	Beispiel
or	$(a < b)$ or $(c < d)$

Ist 1 kleiner als 2 oder 3 kleiner als 4? Nur einer muss wahr sein.

Mit not können Sie einen Vergleich umkehren: not $(a < b)$ ist das gleiche wie $a >= b$

5.4 Fallunterscheidungen:

5.4.1 if

Jetzt wissen wir, wie wir mit Vergleichs- und logischen Operatoren mehrere Bedingungen in einer if-Anweisung testen können:

```
a = 7

if a > 6 and a < 10:
   print("a ist größer als 6 und kleiner als 10")
```

a ist größer als 6 und kleiner als 10

5.4.2 elif

Oft will man aber nicht nur eine Bedingung testen. Wenn Sie einer zweite Bedingung hinzufügen möchten, dann können Sie das mit elif tun:

```
a = 5
if a > 6 and a < 10:
   print("a ist größer als 6 und kleiner als 10")
elif a > 4 and a <= 6:
   print("a ist größer als 4 und kleiner als oder gleich 6")
```

a ist größer als 4 und kleiner als oder gleich 6

Sie können beliebig viele Bedingungen mit elif anfügen, solange Sie am Anfang eine einzelne if-Bedingung stehen haben:

```
a = 1
if a > 6 and a < 10:
   print("a ist größer als 6 und kleiner als 10")
elif a > 4 and a <= 6:
```

```
   print("a ist größer als 4 und kleiner als oder gleich 6")
elif a > 2 and a <= 4:
   print("a ist größer als 2 und kleiner als oder gleich 4")
elif a > 0 and a <= 2:
   print("a ist größer als 0 und kleiner als oder gleich 2")
```

a ist größer als 0 und kleiner als oder gleich 2

5.4.3 else

Schließlich können Sie am Schluß noch mit else alle anderen Fälle abfangen:

```
a = -1
if a > 6 and a < 10:
   print("a ist größer als 6 und kleiner als 10")
elif a > 4 and a <= 6:
   print("a ist größer als 4 und kleiner als oder gleich 6")
elif a > 2 and a <= 4:
   print("a ist größer als 2 und kleiner als oder gleich 4")
elif a > 0 and a <= 2:
   print("a ist größer als 0 und kleiner als oder gleich 2")
else:
   print("Ihre Zahl liegt nicht zwischen 0 und 10")
```

Ihre Zahl liegt nicht zwischen 0 und 10
Wenn man in a versucht einen String anzugeben bekommt man folgende Fehlermeldung:
TypeError: $'>'$ not supported between instances of $'str'$ and $'int'$

5.5 nesting

Sie bekommen eine Fehlermeldung, da Sie versuchen, den Vergleichsoperator mit zwei verschiedenen Datentypen zu verwenden. Hier macht es Sinn, dass man zuerst testet, ob der gewählte Wert eine Zahl ist. Wie Sie im folgenden Beispiel sehen können, kann man if/elif/else ineinander verschachteln (nesting). Wichtig dabei ist, dass für jede Einrückungsstufe die gleiche Anzahl von Leerzeichen verwendet wird (normalerweise 4), was aber normalerweise Ihr Editor für Sie übernimmt.

```
a = 4
if isinstance(a, (int, float)) == True:
    if a > 6 and a < 10:
        print("a ist größer als 6 und kleiner als 10")
    elif a > 4 and a <= 6:
        print("a ist größer als 4 und kleiner als oder gleich 6")
    elif a > 2 and a <= 4:
        print("a ist größer als 2 und kleiner als oder gleich 4")
    elif a >= 0 and a <= 2:
        print("a ist größer als 0 und kleiner als oder gleich 2")
    else:
        print("Ihre Zahl liegt nicht zwischen 0 und 10")
else:
    print("Bitte wählen Sie eine Zahl als Wert")
```

Bitte wählen Sie eine Zahl als Wert

5.6 conditional expression

Mit einem bedingten Ausdruck (conditional expression) können Sie eine if/else- Konstruktion (mit einer Bedingung) kürzen. Die beiden Beispiele geben Ihnen das gleiche Ergebnis:

```
a = 4
print ("Zahl ist gerade" if a % 2 == 0 else "Zahl ist ungerade")
```

Zahl ist gerade

5.7 Kontrollstrukturen:

5.7.1 Schleifen

Mit schleifen können Sie den selben Codeblock mehrmals ausführen lassen. Python bietet Ihnen zwei Wege, um Schleifen zu erzeugen: mit while und for.
Wenn Sie mit while arbeiten, legen Sie eine Bedingung fest; solange die Bedingung erfüllt ist, läuft die Schleife weiter. Erst wenn die Bedingung nicht mehr erfüllt ist, wird die Schleife beendet:

```
a = 1
while a < 5:
    print (a)
    a = a + 1 # Ohne diese Zeile läuft die Schleife endlos weiter da a immer kleiner als 5 bleibt
print("Fertig!")
```

```
1
2
3
4
Fertig!
```

5.7.2 Schleifen mit while

Sie können Schleifen mit break auch vorzeitig beenden:

```
a = 1
while a < 5:
    print (a)
    if a == 3:
        break
    a = a + 1
print ("Fertig!")
```

```
1
2
3
Fertig!
```

5.7.3 Schleifen mit for

Wenn Sie eine Schleife mit for bilden wollen, benötigen Sie ein iterierbares (iterable) Objekt, wie etwa eine Liste oder ein Tupel. Für jedes Objekt läuft die Schleife einmal. Wenn die Schleife für jedes Element durchgelaufen ist, endet sie.

```
a = ["banane", "kirsche", "apfel", "ananas"]
for x in a:
    print (x)
```

```
banane
kirsche
apfel
ananas
```

Sie können genauso gut mit Zahlensequenzen arbeiten. Hier ein Beispiel einer mit range() erzeugten Sequenz aller gerader Zahlen von 20 bis 2:

```
a = range (20, 1, -2)
for x in a:
    print (x)
```

```
20
18
16
14
12
10
8
6
4
2
```

Wie bei while- Schleifen können Sie auch bei for Bedingungen eibauen:

```
a = ["banane", "kirsche", "apfel", "ananas"]
for x in a:
    if x == "apfel":
        print(x.upper())
    else:
        print(x)
```

```
banane
kirsche
APFEL
ananas
```

5.7.4 pass

Wenn Sie an einem Programm arbeiten, gibt es möglicherweise Stellen, an denen Sie vorerst nur einen Platzhalter benötigen. Dafür können Sie pass verwenden. Die Fallunterscheidung oder Schleife wird dann einfach ignoriert:

```
a = ["banane", "kirsche", "apfel", "ananas"]
for x in a:
    if x == "apfel":
        print(x.upper())
    elif x == "banane":
        pass
    else:
        print(x)
```

5.8 Zurück zu Datenstrukturen:

5.8.1 Dictionary

Ein Dictionary ist eine Datenstruktur, in der Schlüssel-Wert-Paare (key-value-pairs) gespeichert werden.
Dabei muss jeder Schlüssel einmalig sein, die Werte Können sich aber wiederholen. Auch muss der Schlüssel durch einen immutablen Datentypen ausgedrückt werden, z.B. durch eine Zeichenkette, Integer, usw. Dictionaries werden in geschweifte Klammern gesetzt. Sie können das Dictionary in eine Zeile schreiben oder für jeden Eintrag eine neue Zeile beginnen, was bei längeren oder verschachtelten Dictionaries übersichtlicher ist.

```
a = {
    "Klaus": 31,
    "Stefanie": 64,
    "Beate": 11,
    "Martin": 71
}
a
```

```
{'Klaus': 31, 'Stefanie': 64, 'Beate' 11, 'Martin': 74}
```

Ein Dictionary kann auch selbst wieder Dicionaries beinhalten. Wenn Sie bspw. für jede Person nicht nur das Alter, sondern auch den Wohnort speichern möchten, können Sie für jede Person ein Dictionary erzeugen, dort Alter und Wohnort speichern und das individuelle Dictionary als Wert mit dem Namen der Person als Schlüssel in einem übergeordneten Dictionary verwenden, wie im folgendem Beispiel:

```
b = {
    "Klaus": {
        "alter": 31,
        "wohnort": "Tübingen"
    },
    "Stefanie": {
        "alter": 64,
        "wohnort": "Rottenburg"
    },
    "Beate": {
        "alter": 11,
        "wohnort": "Reutlingen"
    }
}
print(b)
```

```
{'Klaus': {'alter': 31, 'wohnort': 'Tübingen'}, 'Stefanie': {'alter': 64, 'wohnort': 'Rottenburg'},
'Beate': {'alter': 11, 'wohnort': 'Reutlingen'}}
```

5.8.1.1 Elemente auswählen

Anders als etwa bei einer Liste oder einem Tupel verwenden Sie bei Dictionaries keine Indexwerte, sondern die Schlüssel, um auf einzelne Elemente zuzugreifen:

```
b["Stefanie"]
```

```
{'alter': 64, 'wohnort': Rottenburg'}
```

```
b["Stefanie"]["wohnort"]
```

```
'Rottenburg'
```

5.8.1.2 Schleifen

Da Dictionaries iterabel sind, können sie auch in Schleifen (normalerweise for-Schleifen) eingesetzt werden. Das folgende Beispiel verwendet das einfachere Dictionary, das Sie zu Beginn gesehen haben und gibt für jedes Element den Schlüssel aus:

```
for x in a:
    print(x)
```

```
Klaus
Stefanie
Beate
Martin
```

Sie können sich auch den Wert des Elementes, oder Wert und Schlüssel gleichzeitig ausgeben lassen:

```
for x in a:
    print(a[x])
```

```
31
64
11
71
```

```
for x in a:
    print("{} ist {} Jahre alt.".format(x, a[x]))
```

```
Klaus ist 31 Jahre alt.
Stefanie ist 64 Jahre alt.
Beate ist 11 Jahre alt.
Martin ist 71 Jahre alt.
```

5.8.1.3 Funktionen

Es gibt verschiedene Funktionen, welche die Arbeit mit Dictionaries erleichtern. Im folgenden Beispiel wird items() verwendet; Sie können hiermit sowohl Schlüssel als auch Werten Variablennamen zuweisen:

```
for name, alter in a.items():
    print("{} ist {} Jahre alt.".format(name, alter))
```

```
Klaus ist 31 Jahre alt.
Stefanie ist 64 Jahre alt.
Beate ist 11 Jahre alt.
Martin ist 71 Jahre alt.
```

5.8.1.4 Verschachtelt in Schleifen

Um mit verschachtelten Dictionaries zu arbeiten, können Sie verschachtelte Schleifen verwenden. Zur Erinnerung noch einmal das Dictionary b aus der Sektion Dictionary
Als Beispiel eine verschachtelte Schleife:

```
for k, v in b.items():
    print(k)
    for x, y in v.items():
        print(y)
```

```
Klaus
31
Tübingen
Stefanie
64
Rottenburg
Beate
11
Reutlingen
```

Im obigen Beispiel können Sie auch in der ersten Schleife über die Schlüssel auf individuelle Werte zugreifen:

```
for k, v in b.items():
    print("{} wohnt in {} und ist {} Jahre alt.".format(k, v["wohnort"], v["alter"]))
```

```
Klaus wohnt in Tübingen und ist 31 Jahre alt.
Stefanie wohnt in Rottenburg und ist 64 Jahre alt.
Beate wohnt in Reutlingen und ist 11 Jahre alt.
```

5.8.1.5 Bitte beachten

Einen Punkt, den Sie bei Dicionaries beachten sollten, ist dass sie ungeordnet sind, d.h. Sie können sich nicht darauf verlassen, dass die Elemente immer in der gleichen Reihenfolge ausgegeben werden. Wenn die Reihenfolge für Sie wichtig ist, können Sie ein geordnetes Dictionary (OrdnetDict) verwenden:

```
import collections # Dazu gleich mehr!

c = collections.OrderedDict() # Dazu gleich mehr!
c["a"] = "A"
c["b"] = "B"
c["c"] = "C"

for k, v in c.items():
    print("{}: {}".format(k, v))
```

```
a: A
b: B
c: C
```

5.9 Module und Pakete importieren

Am Beginn des vorherigen Beispiels haben Sie die folgende Zeile gesehen:
import collections
Mit dieser Zeile haben Sie das Modul collections in Ihr Programm eingebunden. Module sind einzelne .py-Dateien, die zusätzliche oder modifizierte Funktionen anbieten. Manche kommen mit der Standardinstallation, andere müssen Sie zusätzlich installieren. Pakete enthalten zumeist mehrere Module. Bevor Sie ein Modul oder Paket in Ihrem Programm verwenden, müssen Sie es erst einbinden. Das machen Sie mit import. Zwecks besserer Übersichtlichket sollte die Einbindung zu Beginn Ihres Programms geschehen.

Wenn Sie Module/Pakete auf diese Weise importieren, wird ein sog. Namensraum (namespace) geschaffen. Das verhindert, dass eine Funtion aus dem importierten Modul oder Paket, welche den gleichen Namen wie eine bereits existierende Funktion hat, diese überschreibt. Deshalb müssen Sie dann vor den Namen der importierten Funktion noch den Namen des Moduls/Paket setzen:
c = collections.OrderedDict()

Wenn Sie möchten, können Sie auch den Namen ändern, unter dem Sie in Ihrem Programm auf das Modul/Paket verweisen:

```
import collections as neuerName
c = neuerName.OrderedDict()
```

Es gibt in Python auch einen globalen Namensraum (global namespace). Den verwenden die Funktionen, die Sie bisher verwendet haben. Sie können Module/Pakete auch teilweise oder komplett in diesem Namensraum einbinden:

```
from collections import *

from collections import OrderedDict

from collections import OrderedDict as sorted
```

Sie müssen zwar bei dieser Methode den Namen des Moduls/Pakets nicht mehr extra nennen:

```
from collections import *
c = OrderedDict()
```

Wenn es aber im globalen Namensraum bereits eine Funktion oder eine Variable mit dem gleichen Namen gibt, kann es passieren, dass diese überschrieben wird.

5.10 Datentabelle:

5.10.1 pandas

Wir werden uns als nächstes mit Datentabellen (DataFrames) befassen. Es gibt verschiedene Pakete, welche ähnliche Datenstrukturen bereitstellen; wir werden pandas verwenden. Da es nicht zur "Grundausstattung" von Python gehört, müssen Sie es erst intallieren (was bei einer Pythoninstallation mit Anaconda gleich mitgeschieht) und dann am Anfang Ihres Programms importieren:

```
import pandas as pd
```

Datentabellen ähneln in Ihrer Struktur Exceltabellen: Daten sind zweidimensional organisiert, eine Zeile für jeden Fall und eine Spalte für jede Variable.

Datentabellen können auf verschiedene Weisen generiert werden. Eine häufig genutzte Möglichkeit besteht darin, ein Dictionary, welches mehrere Listen gleicher Länge enthält, zu einer Datentabelle zu konvertieren:

```
daten1 = {'stadt': ['Berlin', 'Hamburg', 'München'],
'bundesland': ['Berlin', 'Hamburg', 'Bayern'],
'einwohnerzahl':[3520031,1787408,1450381]}
df1 = pd.DataFrame(daten1)
```

```
      stadt bundesland  einwohnerzahl
0    Berlin     Berlin        3520031
1   Hamburg    Hamburg        1787408
2   München     Bayern        1450381
```

Sie können jederzeit neue Reihen hinzufügen. Hier kommt eine neue Stadt hinzu. Beachten Sie den Parameter ignore_index. Die hinzugefügende Zeile wird als eine series interpretiert, ein eindimensionaler Datentyp, der auch durch pandas implementiert wird und bei dem jeder Wert mit einem Indexwert verknüpft wird. Durch den Parameter wird der Indexwert der series ignortiert:

```
daten2 = {'stadt': 'Köln', 'bundesland': 'Nordrhrein-Westfalen', 'einwohnerzahl': 1060582}
df1 = df1.append(daten2, ignore_index=True)
```

```
      stadt            bundesland  einwohnerzahl
0    Berlin                Berlin        3520031
1   Hamburg               Hamburg        1787408
2   München                Bayern        1450381
3      Köln  Nordrhrein-Westfalen        1060582
```

Wenn Sie mehrere Reihen hinzufügen möchten, können Sie diese zuerst in eine Datentabelle umwandeln und dann mit append ans Ende hängen:

```
daten3 = {'stadt': ['Frankfurt', 'Stuttgart'],
'bundesland': ['Hessen', 'Baden-Württemberg'],
'einwohnerzahl': [732688,623738]}
df2 = pd.DataFrame(daten3)
df1 = df1.append(df2, ignore_index=True)
```

```
        stadt            bundesland  einwohnerzahl
0      Berlin                Berlin        3520031
1     Hamburg               Hamburg        1787408
2     München                Bayern        1450381
3        Köln  Nordrhrein-Westfalen        1060582
4   Frankfurt                Hessen         732688
5   Stuttgart    Baden-Württemberg         623738
```

5.10.2 Daten laden

Sie können mit Pandas auch Daten aus einer externen Datei in eine Datentabelle laden. Dazu bietet Pandas verschiedene Methoden an, welche unterschiedliche Datenformate laden können (z.b. Excel, JSON, SAS, Stata). Hier verwenden wir die read_csv-Funktion. Diese lädt normalerweise Textdateien mit comma-separated values, d.h., Dateien, in denen die Werte der einzelnen Spalten durch Kommas voneinander getrennt werden. Man kann aber mit dem sep-Parameter aber auch ein anderes Zeichen auswählen; im Beispiel ist das ein Tabulatorzeichen. Bitte beachten Sie, dass der Dateipfad von Computer zu Computer unterschiedlich sein kann.

Bei sehr langen Datensätzen können Sie sich mit der Funktion head auch nur die ersten paar Zeilen anzeigen lassen:
df2.head()

5.10.3 Datentabellen erweitern

Nachdem Sie eine Datentabelle erzeugt haben, können Sie diese auch erweitern. So kann man dem Datensatz df2 etwa eine Spalte hinzufügen, in der das Jahr der Weltmeisterschaft aufgeführt wird, falls man irgendwann auch frühere Veranstaltungen berücksichtigen möchte:

```
df2["wm_jahr"] = 2019
df2.head()
```

Man kann aber auch Spalten einfügen, die auf bereits existierenden Spalten basieren. So können wir etwa in einer neuen Spalte angeben, wie groß die jeweilige Person in Metern und nicht in Zentimetern ist.

```
df2["groesse_meter"] = df2["groesse"] / 100
```

5.10.4 Spalten auswählen

Wenn Sie aus einer Datentabelle eine individuelle Spalte auswählen möchten, können Sie dies tun, indem Sie den Spaltennamen in eckige Klammern setzen:

```
df2["name"].head()
```

Bei mehreren Spalten benötigen Sie zwei Klammern:

```
df2[["name", "geburtsjahr"]].head()
```

5.10.5 shape und columns

Datentabellen haben Attribute, die Eigenschaften der Datentabelle beschreiben. Mit shape können Sie sich beispielsweise die Dimensionen (Anzahl der Zeilen und Spalten) der Datentabelle anzeigen lassen, während columns die Variablennamen ausgibt.

```
df2.shape
```

```
df2.columns
```

5.10.6 value_counts()

Bei der Arbeit mit Datentabellen stehen Ihnen zahlreiche Funktionen zur Auswahl. So können Sie sich etwa mit value_count() anzeigen lassen, wie oft jeder Wert in einer Spalte vorkommt. Mit dem folgenden Beispiel kann man etwa für jede Größenangabe sehen, wie viele Spielerinnen in diese Kategorie fallen.

```
df2["groesse"].value_counts().head()
```

5.10.7 min() und max()

min() und max() zeigen Ihnen die Minimal- und Maximalwerte einer Spalte:

```
df2["groesse"].min()
```

5.10.8 mean()

Mit mean() können Sie den Durchschnittswert ausgewählten Spalten berechnen, solange die Spalten numerische Datentypen enthalten:

```
df2.mean()
```

5.10.9 idxmax() und idxmin()

idxmax() und idxmin() geben Ihnen den Indexwert des höchsten bzw. niedrigsten Wertes. Wenn es mehrere Zeilen mit dem gleichen Wert gibt, wird nur die erste Zeile berücksichtigt:

5.10.10 loc und iloc

Wenn Sie dann die Indexwerte haben, können Sie den Indexoperator loc verwenden, um die Zeile aus der Datentabelle auszuwählen:

```
df2[["name", "groesse"]].loc[df2["groesse"].idxmax()]
```

5.10.11 Zeilen auswählen

Bei mehreren gleichen Höchstwerten können Sie einen Vergleich und die Funktion max() verwenden (oder min() bei Niedrigstwerten):

```
df2[["name", "groesse"]].loc[df2["groesse"] == df2["groesse"].max() - 10]
```

5.10.12 groupby()

groupby ist immer hilfreich, wenn Sie Ihren Datensatz nach Werten in einer oder mehreren Spalten gruppieren und dann etwa Durchschnittswerte für jede dieser Gruppen berechnen wollen:

```
df2.groupby("mannschaft")["geburtsjahr"].mean().sort_values()
```

5.10.13 sort_values()

Mit sort_values() können Sie einen Datensatz sortieren:

```
df2.sort_values(by=["mannschaft", "groesse"]).head()
```

5.10.14 astype()

Mit astype() verändern Sie den Datentyp einer Spalte. So kann man etwa die Werte in einer geeigneten Spalte in Faktoren umwandeln. Am Ende können Sie dann die unterschiedlichen Ausprägungen (categories) des Faktors sehen:

```
df2["mannschaft"].astype("category").head()
```

6 Video06