Informatik 1

Python II - Grundlagen

Jonas Miederer

DHBW Stuttgart - Campus Horb

3. Dezember 2020

DHBW
Duale Hochschule
Baden-Württemberg

Outline:

DHBW
Duale Hochschule
Baden-Württemberg

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Variablen - Deklaration

In Python ist kein spezieller Befehl nötig, um Variablen zu erzeugen und zu deklarieren. Es muss lediglich festgelegt werden, wie der Name der Variable lautet und welcher Wert ihr zugewiesen wird.

```
>>> x = 1
>>> y = 3.14
>>> z = "Hello world"
```

In obigen Beispiel wurden 3 Variablen mit den Namen x, y und z erzeugt und ihnen jeweils die Werte 1, 3.14 und `"Hello world"` zugewiesen.

Mehrfachzuweisung

Auf der linken Seite der Variablenzuweisung steht immer der Variablenname, dann folgt ein `=`-Zeichen und auf der rechten Seite wird der Wert festgelegt.

In Python folgt nach jeder Anweisung eine neue Zeile, Folgendes ist also nicht erlaubt:

```
>>> x = 1, y = 3.14
```

Es können jedoch trotzdem mehrere Variablen innerhalb einer Zeile deklariert werden *(Multiple Assignment)*:

```
>>> x, y = 1, 3.14
```

Auf diese Weise wird ebenso wie oben der Variable `x` der Wert `1` und der Variable `y` der Wert `3.14` zugewiesen

DHBW Duale Hochschule Baden-Württemberg

Wertüberschreibung

Nachdem einer Variablen ein Wert zugewiesen wurde, kann diese im Anschluss ohne Umstände auch mit einem anderen Wert überschrieben werden:

```
>>> x = 1
>>> x = 5
>>> x
5
```

Der Variablen `x` wurde zunächst der Wert `1` zugewiesen und anschließend mit dem Wert `5` überschrieben. `x` besitzt nun also den Wert `5`. Eine Variable kann auch mit einem Wert eines anderen Datentyps überschrieben werden:

```
>>> x = 1
>>> x = "Hello world"
>>> x
"Hello world"
```

DHBW Duale Hochschule Baden-Württemberg

Ausgabe

Um sich den aktuellen Wert einer Variablen anzeigen zu lassen, muss im interaktiven Modus des Python-Interpreters lediglich der Name der Variable eingegeben werden, der Wert wird dann als Ausgabe in der nächsten Zeile angezeigt:

```
>>> x = 1
>>> x
1
```

Alternativ kann auch die `print()` -Funktion genutzt werden, um Werte auszugeben:

```
>>> x = 1
>>> print(x)
1
```

Ausgabe

Achtung

Die 1. Methode, bei der lediglich der Variablenname "ausgeführt" wird, funktioniert nur im interaktiven Modus des Interpreters. Wird der Code innerhalb einer Datei o.ä. ausgeführt, muss immer die `print()` -Funktion genutzt werden.

Benamungsregeln & -konventionen

Regeln: Variablennamen

- **müssen** mit einem Buchstaben oder einem Underscore (_) beginnen
- **dürfen nicht** mit einer Zahl/Ziffer beginnen
- **dürfen lediglich** aus alphanumerischen Zeichen (A-Z, a-z, 0-9) und Underscores (_) bestehen
- sind case-sensitive (Groß-/ Kleinschreibung wird unterschieden, z.B. $x \neq X$)

Konventionen: Variablennamen

- **sollten** möglichst aussagekräftig sein (z.B. ist der Variablenname `x` kaum aussagekräftig, `age` dagegen schon)
- bestehen i.d.R. nur aus Kleinbuchstaben, wobei einzelne Wörter durch Underscores getrennt werden (z.B. `age_of_person`), wird als *Snake case (snake_case)* bezeichnet[1]

[1] In manchen anderen Programmiersprachen (z.B. Java) wird überwiegend *Camel case (camelCase)* genutzt, wobei Wörter nicht durch Underscores getrennt werden, sondern der erste Buchstabe eines Wortes groß geschrieben wird (z.B. `ageOfPerson`)

Aufgaben

1. Wie werden in Python Variablen erzeugt?
2. Wie können den Variablen `name` und `age` innerhalb einer Zeile verschiedene Werte zugewiesen werden?
3. Sind Variablenwerte konstant? Wie können sie verändert werden?
4. Welche der folgenden Variablennamen sind valide?
   - `_name`
   - `name_`
   - `Name`
   - `int`
   - `int$`
   - `name_and_age`
   - `nameAndAge`
   - `name and age`
   - `1variable`
   - `variable1`
5. Was muss bei der Ausgabe von Werten im interaktiven Interpreter bzw. der Ausführung des Codes in Dateien beachtet werden?
6. Wie sollten Variablen in Python benamt werden?

DHBW Duale Hochschule Baden-Württemberg

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Kommentare (I)

Wie in den meisten anderen Programmiersprachen besteht in Python die Möglichkeiten, den Code mit Kommentaren anzureichern.

Kommentare dienen dazu, Informationen im Code zu ergänzen, um beispielsweise die Funktionalität zu beschreiben bzw. dokumentieren, anderen Entwicklern eine Hilfestellung zu geben, den Code besser verstehen zu können oder auch komplexe Sachverhalte in natürlicher Sprache wiederzugeben, die in Codeform nur schwer verständlich sind.

Kommentare werden vom Interpreter nicht berücksichtigt, daher können Kommentare einen beliebigen Inhalt besitzen.

DHBW
Duale Hochschule
Baden-Württemberg

Kommentare (II) - Einzeilige Kommentare

Kurze Kommentare werden mit einem `#`-Zeichen vorangestellt. Diese Kommentare gelten nur bis zum jeweiligen Zeilenende.

```
# Das ist ein Kommentar.
x = 1 #  Das ist ein Kommentar, der erst nach der Variablenzuweisung beginnt
```

Kommentare (III) - Mehrzeilige Kommentare

Manchmal (insbesondere bei der Beschreibung komplexer Sachverhalte) können sich Kommentare auch über viele Zeilen erstrecken. Um nicht explizit jede Zeile mit `#` voranstellen zu müssen, gibt es in Python die Möglichkeit, mehrzeilige Kommentare zu erzeugen.
Mehrzeilige Kommentare werden mit 3 Anführungszeichen (`"""`) eingeleitet und auch wieder beendet. Alles was dazwischen enthalten ist wird als Kommentar behandelt und vom Interpreter ignoriert.

```
"""
Das ist ein mehrzeiliger Kommentar.

Alles was zwischen den Anführungszeichen steht wird vom Interpreter ignoriert.
Hier ist der Kommentar beendet
"""
```

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Datentypen

Da Python eine typisierte Sprache ist, besitzt jede Variable auch einen Datentypen. Datentypen beschreiben, welche Art von Wert in einer Variablen gespeichert ist.

Hinweis

Um zu überprüfen, welchen Datentypen eine Variable besitzt, gibt es in Python die `type()` -Funktion. Um bspw. den Datentypen der Variablen `x` festzustellen, kann die Funktion folgendermaßen aufgerufen werden: `type(x)`

Numerische Datentypen (I)

Numerische Datentypen beschreiben Zahlenwerte unterschiedlicher Art:

- **integer**: Positive und negative Ganzzahlen (keine Nachkommastellen).
  Bezeichnung in Python: `int`

```
1 >>> x = 5
2 >>> type(x)
3 <class 'int'>
```

- **float**: Positive und negative reelle Zahlen mit Fließkommadarstellung (mit Nachkommastellen).
  Bezeichnung in Python: `float`

```
1 >>> x = 3.14
2 >>> type(x)
3 <class 'float'>
```

DHBW Duale Hochschule Baden-Württemberg

Numerische Datentypen (II)

- **complex**:Komplexe Zahlen mit einem reellen und einem imaginären Anteil. Sowohl der reelle als auch der komplexe Anteil wird durch `float`-Zahlen dargestellt. In der Ausgabe wird der imaginäre Teil durch ein `j` gekennzeichnet, um deutlich zu machen, dass es sich dabei um eine imaginäre Zahl handelt. Im Beispiel unten wird eine komplexe Zahl mit dem reellen Anteil `1` und dem imaginären Anteil `0` erzeugt.

  Bezeichnung in Python: `complex`

```
1  >>> x = complex(1,0)
2  >>> x
3  (1+0j)
4  >>> type(x)
5  <class 'complex'>
```

Numerische Datentypen (III)

Achtung

Bei der Verwendung von Kommazahlen muss auf die korrekte Verwendung des Dezimaltrennzeichens geachtet werden. Da die Schreibweisen in der Informatik meist auf dem englischen System basieren, wird als Dezimaltrennzeichen ein Punkt und kein Komma verwendet (z.B. `3.14` anstatt `3,14`).

DHBW Duale Hochschule Baden-Württemberg

Boolesche Datentypen

- **boolean**: Benannt nach George Boole. Stellt einen einfachen Zustandswert dar. Es gibt lediglich zwei Zustände: Wahr und Falsch (analog: Binärdarstellung (1 / 0), Schalterdarstellung (An / Aus)).
  Bezeichnung in Python: `bool`

```
1  >>> x = True
2  >>> y = False
3  >>> type(x)
4  <class 'bool'>
5  >>> type(y)
6  <class 'bool'>
```

Achtung

`True` und `False` sind eingebaute Schlüsselwörter in Python und daher case-sensitive. `true` bzw. `false` sind keine validen booleschen Werte.

Zeichenketten (I)

- **string**: Zeichenketten, meist *strings* genannt, sind eine Folge von einzelnen Zeichen, Buchstaben und Ziffern, die als Buchstaben behandelt werden, die von einfachen (`'`) oder doppelten (`"`) Anführungszeichen eingeschlossen werden.
  Bezeichnung in Python: `str`

```
1  >>> x = 'Hello world'
2  >>> type(x)
3  <class 'str'>
4  >>> y = "1234"
5  >>> type(y)
6  <class 'str'>
```

Achtung

Ein String wird entweder von einfachen oder doppelten Anführungszeichen eingeschlossen, eine Kombination innerhalb eines Strings ist aber **nicht** erlaubt (z.B. `"Hello world'`)

DHBW Duale Hochschule Baden-Württemberg

Zeichenketten (II)

Zeichenketten müssen nicht zwangsläufig statisch angelegt werden, sondern können auch dynamisch erzeugt werden, d.h. mit den Werten anderer Variablen ergänzt werden.

Hierfür können sogenannte **f-Strings** verwendet werden. Um diese zu nutzen, muss die Zeichenkette mit einem "f" ("formatiert") vorangestellt werden. Variablenwerte, die in den String eingefügt werden sollen, werden innerhalb des Strings mit geschweiften Klammern (`{}`) injiziert.

```
>>> first_name = 'John'
>>> age = 42

>>> print(f"Hallo {first_name}, du bist {age} Jahre alt")
"Hallo John, du bist 42 Jahre alt"
```

Zeichenketten (III)

Alternativ können Zeichenketten auch mit einen `+`-Zeichen konkateniert (zusammengesetzt) werden.

Achtung

String-Konkatenation mithilfe des Addition-Operators funktioniert nur zwischen Strings, andere Datentypen müssen vor der Konkatenation zunächst in den String-Datentyp konvertiert werden.

```
>>> first_name = 'John'
>>> age = 42

>>> print("Hallo " + first_name + ", du bist " + age +  " Jahre alt")
TypeError: can only concatenate str (not "int") to str

>>> print("Hallo " + first_name + ", du bist " + str(age) +  " Jahre alt")
"Hallo John, du bist 42 Jahre alt"
```

DHBW Duale Hochschule Baden-Württemberg

Zeichenketten (IV) I

Zur Zeichenkettenverarbeitung stehen weitere Funktionen zur Verfügung, die bereits standardmäßig in Python enthalten sind sind.

- `upper()` : Wandelt String in Großbuchstaben um (analog dazu: `lower()`)

```
>>> "Hello".upper()
"HELLO"
```

- `startswith()` : Überprüft, ob ein String mit einer bestimmten Zeichenkette beginnt und liefert dementsprechend einen booleschen Wert zurück (analog dazu: `endswith()`)

```
>>> "Hello John".startswith("Hello")
True
```

Zeichenketten (IV) II

- `split()` : Trennt einen String anhand der gewünschten Zeichen und gibt die Bestandteile als Liste zurück. Falls kein Parameter angegeben wird, wird standardmäßig anhand von Leerzeichen getrennt

```
1  >>> "Hello John".split()
2  ["Hello", "John"]
3
4  >>> "This.sentence.is.dot.separated".split(".")
5  ['This', 'sentence', 'is', 'dot', 'separated']
```

In Python stehen viele weitere String-Funktionen zur Verfügung. Eine vollständige Beschreibung dieser Funktionen kann in der offiziellen Dokumentation eingesehen werden.

DHBW Duale Hochschule Baden-Württemberg

None

Um zu kennzeichnen, dass eine Variable existiert, diese jedoch keinen Wert beinhaltet, also *leer* ist, gibt es den Datentyp und das Schlüsselwort `None`.

Achtung

`None` besitzt eine andere semantische und logische Bedeutung als der Wert `0`. `0` bedeutet, dass ein Wert vorhanden ist, dieser aber 0 beträgt, also einen numerischen Wert trägt. `None` dagegen bedeutet, dass überhaupt kein Wert gesetzt ist, daher handelt es sich auch nicht um einen numerischen Datentyp.

```
>>> x = None
>>> type(x)
<class 'NoneType'>
>>> print(x)
None
>>> x is None
True
>>> x == None
True
>>> x == 0
False
```

Weitere Datentypen

- **Sequence** Datentypen: Sammlung mehrerer Werte (gleicher oder unterschiedlicher Datentypen) innerhalb einer Variable, z.B. `list`, `tuple`, `range`
- **Mapping** Datentypen: Sammlung mehrerer Werte (gleicher oder unterschiedlicher Datentypen) nach dem Key-Value Prinzip, z.B. `dict` (Dictionary/Wörterbuch)
- **Set** Datentypen: Sammlung mehrerer Werte (gleicher oder unterschiedlicher Datentypen) ohne Duplikate innerhalb einer Variable, z.B. `set`
- **Binäre** Datentypen: Darstellung von Zahlen im Binärsystem, z.B. `bytes`.

Sequence, Mapping und Set Datentypen werden im Abschnitt Einfache Datenstrukturen näher erläutert, binäre Datentypen werden in diesem Rahmen nicht besprochen.

DHBW
Duale Hochschule
Baden-Württemberg

Konvertierungen (I)

Da Python schwach typisiert ist, können Variablen prinzipiell ohne weiteres in andere Datentypen umgewandelt werden. Hierfür stehen bereits standardmäßig Funktionen zu Verfügung, die genutzt werden können:

Umwandlung in ...	Integer	Float	Complex	Boolescher Wert	String
Funktion	int()	float()	complex()	bool()	str()

```
>>> x = 5
>>> type(x)
<class 'int'>
>>> y = float(x)
>>> type(y)
<class 'float'>
>>> y
5.0
```

Konvertierungen (II)

Bei Typkonvertierungen muss immer darauf geachtet werden, ob die Konvertierung problemlos möglich ist, u.U. mit Datenverlust oder Fehlern gerechnet werden muss oder grundsätzlich nicht möglich ist.

Von… \ zu…	int	float	complex	bool	str

int		Zahl besitzt zusätzlich Dezimalstelle
		`float(5)` ⇒ `5.0`
			Integerzahl als reelle Zahl, kein imaginärer Anteil
			`complex(5)` ⇒ `(5+0j)`
				0 wird zu `False` ausgewertet, alle anderen Zahlen zu `True`
				`bool(5)` ⇒ `True`
				`bool(0)` ⇒ `False`
					String mit der Zahl als Inhalt
					`str(5)` ⇒ `'5'`

float	Datenverlust, falls Zahl einen Nachkommawert ≠ 0 hat (Nachkommastellen werden abgeschnitten)
	`int(5.0)` ⇒ `5`
	`int(5.9)` ⇒ `5`
			Floatzahl als reelle Zahl, kein imaginärer Anteil
			`complex(5.0)` ⇒ `5.0+0j`
				0.0 wird zu `False` ausgewertet, alle anderen Zahlen zu `True`
				`bool(5.0)` ⇒ `True`
				`bool(0.0)` ⇒ `False`
					String mit der Zahl als Inhalt
					`str(5.0)` ⇒ `'5.0'`

complex	nicht möglich	nicht möglich		Wenn sowohl der reelle als auch der imaginäre Teil 0 ist, wird der Ausdruck zu `True` ausgewertet, ansonsten zu `False`
				`bool(complex(5,0))` ⇒ `True`
				`bool(complex(0,0))` ⇒ `False`
					String mit der komplexen Zahl als Inhalt
					`str(complex(5,2))` ⇒ `'(5+2j)'`

bool	`True` wird zu `1` ausgewertet, `False` zu `0`
	`int(True)` ⇒ `1`
	`int(False)` ⇒ `0`
		`True` wird zu `1.0` ausgewertet, `False` zu `0.0`
		`float(True)` ⇒ `1.0`
		`float(False)` ⇒ `0.0`
			`True` wird zu `1.0+0j` ausgewertet, `False` zu `0.0+0j`
			`complex(True)` ⇒ `1.0+0j`
			`complex(False)` ⇒ `0.0+0j` ⇒ `0j`
					String mit dem booleschen Wert als Inhalt
					`str(True)` ⇒ `'True'`
					`str(False)` ⇒ `'False'`

str	Fehler, falls String einen nicht-numerischen Wert enthält
	`int('5')` ⇒ `5`
	`int('hello')` ⇒ `Error`
		Fehler, falls String einen nicht-numerischen Wert (exkl. Dezimaltrennzeichen) enthält
		`float('5.0')` ⇒ `5.0`
		`float('5.hello')` ⇒ `Error`
			Fehler, falls String nicht der Darstellung einer komplexen Zahl entspricht oder der String Leerzeichen enthält
			`complex('5+2j')` ⇒ `5+2j`
			`complex('5')` ⇒ `5+0j`
			`complex('5 + 2j')` ⇒ `Error`
				Leerer String wird zu `False` ausgewertet, alle anderen Strings zu `True`
				`bool('hello world')` ⇒ `True`
				`bool('')` ⇒ `False`	

DHBW Duale Hochschule Baden-Württemberg

Aufgaben

1. Ist Python eine typisierte oder typenlose Sprache?
2. Ist Python stark oder schwach typisiert?
3. Mit welcher Funktion kann der Typ einer Variablen überprüft werden?
4. Welche numerischen Datentypen gibt es und worin unterscheiden sie sich?
5. Was muss bei der Dezimaldarstellung von Zahlen beachtet werden?
6. Wofür werden boolesche Datentypen genutzt?
7. Können die Datentypen ineinander umgewandelt werden? Was ist dabei zu beachten? Wie lauten die Funktionen?
8. Was ist das Ergebnis folgender Umwandlungen?:

```
1 >>> float(50)
2 >>> str(50)
3 >>> int(complex(3,1))
4 >>> complex("3+ 1j")
5 >>> str('int')
6 >>> bool(3.14)
```

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Grundrechenarten

Mithilfe von Python können unterschiedlichste mathematische Operationen durchgeführt werden. Dazu gehören z.B. die Grundrechenarten Addition, Subtraktion, Multiplikation und Division:

```
>>> x = 10
>>> y = 5
>>> x + y
15
>>> x - y
5
>>> x * y
50
>>> x/y
5.0
```

Grundrechenarten (II)

Beachte

Falls die Rechenoperation lediglich aus Integerwerten besteht, ist auch das Ergebnis der Addition, Subtraktion und Multiplikation stets ein Integerwert. Lediglich bei der Division kann das Ergebnis eine Fließkommazahl sein, sodass der Datentyp einer Division von zwei Integerwerten immer ein Floatwert ist.
Bei Operationen mit Floatwerten ist das Ergebnis immer ein Floatwert, unabhängig von der Rechenart.

Weitere Rechenoperationen (I)

Neben den Grundrechenarten sind in Python auch weitere Rechenoperationen möglich:

- Modulo: Berechnet den Dezimalpart eines Quotienten ("Rest einer Division"). Operator: `%`

```
>>> x = 10
>>> y = 4
>>> x / y
2.5
>>> x % y
2
```

- Potenz: Eine Potenz wird durch zwei Multiplikationszeichen dargestellt. Davor befindet sich die Basis, dahinter der Exponent. Operator: `**`

```
>>> 2 ** 4
16
>>> 5 ** 0
1
```

Weitere Rechenoperationen (II)

- Integer-Division: Rundet den Quotienten auf den nächsten Integerwert ab. Operator: `//`

```
>>> x = 9
>>> y = 2
>>> x / y
4.5
>>> x // y
4
```

- Weitere (komplexere) mathematische Funktionen wie `sin()`, `abs()`, `round()`, ... werden durch das `math`-Modul bereitgestellt.

Kurzschreibweise I

Falls dem Wert einer Variablen `x` ein beliebiger anderer Wert hinzuaddiert werden soll und das Ergebnis wiederum der Variablen `x` zugewiesen werden soll, so kann dies in Python ausgedrückt werden mit:

```
>>> x = 5
>>> x = x + 2 # Der Variablen x wird 2 hinzuaddiert und das Ergebnis der Variablen x
 ↪  zugewiesen
```

Für diese Operation existiert in Python eine Kurzschreibweise, die häufig genutzt wird:

```
>>> x = 5
>>> x += 2 # Der Variablen x wird 2 hinzuaddiert und das Ergebnis der Variablen x
 ↪  zugewiesen, äquivalent zu obiger Schreibweise
```

Diese Kurzschreibweise ist nicht nur auf Additionen anwendbar, sondern auf alle weiteren mathematischen Operationen:

DHBW Duale Hochschule Baden-Württemberg

Kurzschreibweise II

```
>>> x = 5
>>> x %= 2 # Entspricht x = x % 2
>>> print(x)
1

>>> x = 5
>>> x *= 2
>>> print(x)
10
```

Ausführungsreihenfolge

Die Ausführungsreihenfolge ("*Order of operations*") ist identisch zur mathematischen Reihenfolge nach dem PEMDAS-Prinzip:

1. Parantheses (Klammern)
2. Exponents (Potenzen, Wurzeln, Logarithmen, ...)
3. Multiplikation & Division
4. Addition & Substraktion

Bei Operationen gleicher Präzedenz (Operatorrangfolge) (z.B. Multiplikation und Division) wird von links nach rechts ausgewertet.

Exkursion: Fließkomma-Arithmetik (I)

```
>>> 10/3
3.3333333333333335
```

Woher kommt die 5 am Ende der Fließkommazahl?

Dieses Problem entsteht aus der Fließkomma-Arithmetik beim Dividieren von Zahlen. Zahlen werden im Computer grundsätzlich durch Binärzahlen dargestellt. Da das binäre System aber lediglich aus Ganzzahlen (0, 1, 2, 4, 8, ...) besteht, müssen Kommazahlen durch Divisionen der Binärzahlen abgebildet werden. Beispielsweise kann die Zahl `0.125` aus `1/8` gebildet werden. `0.75` kann dagegen aus `1/2 + 1/4` gebildet werden.

Exkursion: Fließkomma-Arithmetik (II)

Da `10/3` jedoch eine Zahl mit unendlich vielen Nachkommastellen darstellt, wären auch unendlich viele Summanden notwendig, um die Zahl korrekt darzustellen. Da dies nicht möglich ist, wird versucht, sich der Zahl so genau wie möglich anzunähern, eine Restungenauigkeit bleibt jedoch immer.

Hinzu kommt, dass innerhalb einer Variable nur begrenzt viel Platz zur Verfügung steht (32 bzw. 64 Bit), aus diesem Grund wird die Zahl abgeschnitten, sobald die maximale Länge erreicht ist. Auch hierzu kommt es also zu einer kleinen Ungenauigkeit ($3 < 3.3 < 3.33 < 3.333 < ...$).

Python rechnet intern mit der Zahl `3.3333333333333334813630699500208720564842224121094`, diese wird aber zur besseren Lesbarkeit in der Ausgabe gekürzt und gerundet, sodass die Zahl `3.3333333333333335` angezeigt wird.

Aufgaben

1. Welchen Datentyp hat eine Addition, Subtraktion, Multiplikation und Division von Integer-, Float- und Complex-Werten
2. Was ist eine Modulorechnung und was ist der entsprechende Operator?
3. Wie wird der Term 3^{-5} in Python ausgedrückt?
4. Wofür kann die Integer-Division nützlich sein?
5. Wie geht der Entwickler vor, wenn er bspw. eine Sinusberechnung durchführen muss?
6. Wofür steht PEMDAS und wo wird es angewendet?
7. Warum werden manche Zahlen ungenau angezeigt? Worin liegt das Grundproblem?

DHBW Duale Hochschule Baden-Württemberg

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Boolesche Operationen

Boolesche Werte (`True` und `False`) können genutzt werden, um Vergleiche anzustellen und Aussagen auf ihren Wahrheitswert zu überprüfen.

Hierfür sind die Wahrheitstabellen der Aussagenlogik relevant.

Es gibt unterschiedliche logische Verknüpfungen, zunächst werden die Verknüpfungen basierend auf zwei booleschen Werten betrachtet.

DHBW
Duale Hochschule
Baden-Württemberg

AND-Verknüpfung

Wert 1	Wert 2	Ergebnis (AND)
True	True	True
True	False	False
False	True	False
False	False	False

Das Ergebnis ist also nur dann `True`, wenn **beide Werte** `True` sind.

Hierfür steht in Python das Schlüsselwort `and` zur Verfügung.

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
```

OR-Verknüpfung

Wert 1	Wert 2	Ergebnis (OR)
True	True	True
True	False	True
False	True	True
False	False	False

Das Ergebnis ist also nur dann `True`, wenn **mindestens ein Wert** `True` ist.

Hierfür steht in Python das Schlüsselwort `or` zur Verfügung.

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
```

XOR-Verknüpfung

Wert 1	Wert 2	Ergebnis (XOR)
True	True	False
True	False	True
False	True	True
False	False	False

Das Ergebnis ist also nur dann `True`, wenn **genau ein Wert** `True` ist.

Hierfür steht in Python der Operator `^` zur Verfügung.

```
>>> True ^ True
False
>>> True ^ False
True
>>> False ^ False
False
```

not-Operator

Auch eine Negierung (Verneinung, Wertumkehr) ist mit booleschen Werten möglich.

Hierfür steht in Python das Schlüsselwort `not` zur Verfügung.

```
>>> not True
False
>>> not False
True
```

Verknüpfung der Operationen

Die Operationen können auch miteinander Verknüpft werden, um komplexere Aussagen zu testen.

```
>>> not ((True and False) or (True and True)) ^ True
True
```

Test auf (Un-)Gleichheit

Variablenwerte können verglichen werden, um so herauszufinden, ob eine Aussage wahr oder falsch ist.
Es kann ebenso wie in der Mathematik sowohl auf Gleichheit als auch auf Ungleichheit getestet werden.

Test auf Gleichheit

Mit dem Test auf Gleichheit kann überprüft werden, ob Variablen einander entsprechen, also den gleichen Wert besitzen.

Achtung

Um zwei Variablen miteinander zu vergleichen reicht kein einfaches `=`, da dieses Zeichen bereits zur Wertzuweisung von Variablen genutzt wird. Daher wird für den Vergleich ein doppeltes Gleichheitszeichen (`==`) verwendet.

```
>>> x = 5
>>> x == 5
True
>>> "hallo" == "welt"
False
>>> x == 5.0
True
```

Test auf Ungleichheit

Analog zum Test auf Gleichheit kann ebenso überprüft werden, ob sich Variablenwerte unterscheiden.

Dabei kann sowohl allgemein auf Ungleichheit (`!=`), als auch auf Verhältnisse (`>` , `<` , `>=` , `<=`) getestet werden.

```
>>> x = 5
>>> x != 4
True
>>> x >= 7
False
>>> "hallo" != "welt"
True
```

Verknüpfung der Operationen

Nun lassen sich alle booleschen Operationen (Verknüpfung, Negierung, Gleichheit, Ungleichheit) miteinander verbinden

```
>>> x>=5 and x<=10 ^ (not 5<10)
True
```

Ausführungsreihenfolge

Auch für boolesche Ausdrücke muss eine Ausführungsreihenfolge definiert sein, um mehrdeutige Ergebnisse auszuschließen. Höchste bis niedrigste Präzedenz:

1. Klammern
2. Potenzfunktionen (Potenz, Wurzel, Logarithmus, ...)
3. Multiplikation, Division
4. Addition, Subtraktion
5. Boolesches XOR (`^`)
6. Vergleiche (`==` , `!=` , `<` , `>` , `<=` , `>=`)
7. Boolesches `not`
8. Boolesches `and`
9. Boolesches `or`

DHBW Duale Hochschule Baden-Württemberg

Aufgaben

1. Wofür werden boolesche Operationen genutzt?
2. Was sind Wahrheitstabellen?
3. Was ist `True AND False`?
4. In welchen Grundregeln können die Wahrheitstabellen für `AND`, `OR` und `XOR` zusammengefasst werden?
5. Wie wird XOR in Python ausgedrückt?
6. Warum kann kein einfaches Gleichheitszeichen verwendet werden, um auf Gleichheit zu testen?
7. Die Variable `temperature` hält den Wert der aktuellen Temperatur. Wie kann überprüft werden, ob die Temperatur zwischen 15 und 21,5 Grad liegt?
8. Auf welche 2 Arten kann `x != 5` noch dargestellt werden?
9. Was ergibt folgender Ausdruck?:

```
1 >>> True or False and False ^ True and (not 5 or 10 <=15)
```

DHBW Duale Hochschule Baden-Württemberg

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Einfache Datenstrukturen - Einführung

Wie bereits vorgestellt können in Python Variablen mit unterschiedlichen Datentypen angelegt werden. Jedoch ist der Entwickler in Python nicht auf die beschriebenen Grunddatentypen beschränkt, sondern es lassen sich auch Datentypen/-strukturen erzeugen, die aus elementaren Datenelementen zusammengesetzt sind.

Diese werden im Folgenden beschrieben.

Listen - Definition

Listen sind einer der am häufigsten genutzten Datentypen in Python. Mithilfe einer Liste können mehrere Werte innerhalb einer einzigen Variable gespeichert werden.

Liste

Eine Liste ist eine *veränderbare, geordnete und indizierte Sammlung von Objekten*

Eigenschaften

- **veränderbar**: Die Objekte / Elemente / Werte, die in einer Liste gespeichert sind, sind keine konstanten Werte, sondern sie können aus der Liste entfernt, überschrieben, verändert, ... werden
- **geordnet**: Die Elemente einer Liste sind nicht zufällig angeordnet, sondern folgen einer festen Ordnung. Zunächst sind die Objekte nach Einfügereihenfolge sortiert, die Sortierung kann aber auch geändert werden (z.B. bei Zahlen nach aufsteigenden Werten oder bei Strings nach dem Alphabet)
- **indiziert**: Da innerhalb einer Liste mehrere Variablenwerte gleichzeitig gespeichert sind, muss es eine Möglichkeit geben, die jeweiligen Werte aus der Liste auszulesen. Dies erfolgt durch *Indizes*.

DHBW Duale Hochschule Baden-Württemberg

Einsatz von Listen

Listen können unterschiedliche Arten von Elementen aufnehmen. Eine Liste kann auch nicht nur aus Variablen eines Datentyps bestehen, sondern auch aus heterogenen Datentypen.

Dies hat zur Folge, dass Listen sehr universell und flexibel eingesetzt werden können.

Erzeugung (I)

Eine Liste kann auf unterschiedliche Arten erzeugt werden:

- `list()`-Konstruktor: Mithilfe des Befehls `list()` kann eine neue, zunächst leere Liste erzeugt werden. Ein Befehl dieser Art wird *Konstruktor* genannt, da hierdurch ein neues Listen-Objekt erzeugt wird [2]

```
1 >>> new_list = list()
2 >>> print(new_list)
3 []
4 >>> type(new_list)
5 <class 'list'>
```

[2] Objekte und Konstruktoren werden im Abs. *Objektorientierte Programmierung* im Detail beschrieben

Erzeugung (II)

- Kurzschreibweise: Elemente, die in einer Liste enthalten sind, werden in Python durch **eckige Klammern** (`[ ]`) umgeben. Als Kurzschreibweise kann anstatt `list()` daher auch einfach `[]` genutzt werden.

```
>>> new_list = []
>>> print(new_list)
[]
>>> type(new_list)
<class 'list'>
```

Speichern von Werten (I)

In den beiden vorherigen Beispielen wurde gezeigt, wie Listen erzeugt werden können, jedoch enthalten diese Listen noch keine Werte, sind also *leer*. Um Werte in den Listen abzuspeichern gibt es unterschiedliche Möglichkeiten.

Speichern von Werten (II)

- **Erzeugung einer vorgefüllten Liste:** Bei der Initialisierung einer neuen Liste können direkt Objekte angegeben werden, die in dieser Liste gespeichert werden sollen:

```
1 >>> names = ['Alice', 'Bob', 'Carol', 'Dave']
2 >>> print(names)
3 ['Alice', 'Bob', 'Carol', 'Dave']
4 >>> type(names)
5 <class 'list'>
```

Hier wird eine neue Liste erzeugt, die in der Variablen `names` gespeichert wird. Die Liste wird zugleich mit 4 Werten gefüllt, sodass in der Variablen nun ein Wert vom Typ `list` gespeichert ist, die wiederum 4 Werte vom Typ `string` enthält.

Innerhalb der eckigen Klammern werden die einzelnen Werte in Listen durch Komma getrennt.

Speichern von Werten (III)

Achtung

Das gleichzeitige Erzeugen einer Liste und Speichern von Werten in ebendieser Liste ist nur mit der Kurzschreibweise möglich. Wird eine Liste mit `list()` erzeugt, können die Werte nicht direkt in der Liste gespeichert werden, hierzu ist ein extra Schritt notwendig.

Speichern von Werten (IV)

- **append**: Existiert die Liste bereits und soll dieser Liste ein Element hinzugefügt werden, kann die `append()` -Funktion genutzt werden. Die Syntax ist:

```
<Liste>.append(<Element>)
```

```
1  >>> names = ['Alice', 'Bob', 'Carol', 'Dave']
2  >>> print(names)
3  ['Alice', 'Bob', 'Carol', 'Dave']
4  >>> names.append('Eve')
5  >>> print(names)
6  ['Alice', 'Bob', 'Carol', 'Dave', 'Eve']
```

Mit `append()` wird das Element immer am Ende der Liste eingefügt.

Speichern von Werten (V)

- **insert**: Soll das Element nicht am Ende der Liste eingefügt werden, sondern an einer beliebigen Stelle, kann `insert` genutzt werden. Die Syntax ist:

```
<Liste>.insert(<Index>, <Element>)
```

```
1  >>> names = ['Alice', 'Carol', 'Dave']
2  >>> print(names)
3  ['Alice', 'Carol', 'Dave']
4  >>> names.insert(1, 'Bob')
5  >>> print(names)
6  ['Alice', 'Bob', 'Carol', 'Dave']
```

Mit `insert()` wird das Element an die gewünschte Stelle der Liste eingefügt.

Zero-based Indexing

Achtung

Wie in den meisten Programmiersprachen beginnen Indizes ("Nummerierungen" innerhalb von Listen und ähnlichen Konstrukten) bei 0 ("*Zero-based Indexing*").

Index	0	1	2	3	4
Element	Alice	Bob	Carol	Dave	Eve

Abbildung 1: Indizierung einer Liste mit 5 Elementen

Im Beispiel hat also das erste Element "Alice" den Index 0 und das letzte Element "Eve" den Index 4.

Speichern von Werten (VI)

- **extend**: Einer Liste können auch Werte hinzugefügt werden, indem die Werte einer weiteren Liste angehängt werden. Hierzu kann die `extend()` -Funktion genutzt werden. Die Syntax ist:

```
<Liste>.extend(<Liste>)
```

```
1 >>> names = ['Alice', 'Bob' 'Carol']
2 >>> print(names)
3 ['Alice', 'Bob', 'Carol']
4 >>> names.extend(['Dave', 'Eve'])
5 >>> print(names)
6 ['Alice', 'Bob', 'Carol', 'Dave', 'Eve']
```

Mit `extend()` werden alle Elemente der zweiten Liste ans Ende der ersten Liste angehängt.

Auslesen von Werten (I)

Um eine Liste sinnvoll nutzen zu können, ist es nicht nur wichtig, Werte in einer Liste abzuspeichern, sondern diese darüber hinaus auch wieder auslesen ("abfragen") zu können.

Ebenso wie die Erzeugung einer Liste erfolgt die Abfrage eines Elements der Liste durch eckige Klammern, die den jeweiligen Index des gewünschten Elements einschließen.

```
>>> names = ['Alice', 'Bob', 'Carol', 'Dave']
>>> second_name = names[1]
>>> print(second_name)
'Bob'
```

Hier wird das Element mit dem Index 1 aus der Liste `names` ausgelesen und in der Variablen `second_name` gespeichert. Da die Indizierung 0-basiert ist, ist das Element mit dem Index 1 *Bob* und nicht *Alice*.

DHBW Duale Hochschule Baden-Württemberg

Auslesen von Werten (II)

Achtung

Der Programmierer muss stets darauf achten, einen validen Index anzugeben. Übersteigt der Index bspw. die Anzahl der in der Liste enthaltenen Elemente, erscheint ein Fehler

```
>>> names = ['Alice', 'Bob', 'Carol', 'Dave']
>>> names[4]
IndexError: list index out of range
```

Auslesen von Werten (III)

Die Abfrage von Werten mithilfe von Indizes kann auch mit negativen Indizes erfolgen. Hierdurch kann die Liste in umgekehrter Reihenfolge abgefragt werden. Dies macht beispielsweise das Auslesen des letzten Listenelements sehr bequem.

```
>>> names = ['Alice', 'Bob', 'Carol', 'Dave']
>>> last_name = names[-1]
>>> print(last_name)
'Dave'
>>> names[1] == names[-3]
True
```

Achtung

Das letzte Listenelement besitzt den Index `-1` und nicht `-0`, da `-0` identisch zur Zahl `0` ist und daher schon für das erste Listenelement vergeben ist

Auslesen von Werten (IV)

Anstatt einzelner Elemente können auch Bereiche innerhalb einer Liste abgefragt werden. Hierfür wird das sogenannte **Slicing** genutzt. Die Syntax ist Ähnlich zur Abfrage einzelner Werte, jedoch wird hier ein Bereich von Indizes, getrennt durch `:` angegeben.

```
>>> names = ['Alice', 'Bob', 'Carol', 'Dave', 'Eve']
>>> names[1:3]
['Bob', 'Carol']
>>> names[0:4]
['Alice', 'Bob', 'Carol', 'Dave']
```

Die erste Zahl stellt dabei den Index dar, bei dem gestartet wird (inklusive), die zweite Zahl stellt den Endindex (exklusive) dar. Im 1. Beispiel wird also bei Index 1 ('Bob') begonnen und bei Index 3 ('Dave') gestoppt, da der Endindex jedoch exklusiv ist, werden lediglich Index 1 und 2 berücksichtigt.

Auslesen von Werten (IV)

Das Slicing kann noch um einen weiteren Wert, den **Step** erweitert werden. Indem neben Start und Stop ein dritter Wert angegeben wird, kann bestimmt werden, in welcher Schrittgröße die Elemente aus der Liste gelesen werden sollen. Wird kein Step angegeben (wie im vorherigen Beispiel), wird standardmäßig eine Schrittweite von 1 verwendet.

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7]
>>> numbers[1:6:2]
[1, 3, 5]
```

Es wird also der Bereich zwischen den Indizes 1 (inklusiv) und 6 (exklusiv) betrachtet und hiervon wird jede 2. Zahl berücksichtigt. Beginnend mit der 1 folgt dann die Zahl 3 sowie die Zahl 5.

Auslesen von Werten (IV)

Wird kein Start-Index angegeben, wird automatisch bei Index 0 begonnen. Wird kein Stopindex angegeben, wird automatisch beim letzten Index gestoppt. Daher ist z.B. auch folgende Schreibweise denkbar:

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7]
>>> numbers[3:]
[3, 4, 5, 6, 7]
>>> numbers[:-2]
[0, 1, 2, 3, 4, 5]
>>> numbers[::2]
[0, 2, 4, 6]
```

Im ersten Beispiel werden alle Elemente beginnend beim 3. Index bis zum Ende der Liste (da kein Endindex angegeben ist) abgefragt (Schrittweite 1).
Im zweiten Beispiel werden alle Elemente beginnend beim Anfang der Liste (da kein Startindex angegeben ist) bis zum vorvorletzten Element abgefragt (Schrittweite 1).
Im dritten Beispiel werden alle Elemente beginnend beim Anfang der Liste (da kein Startindex angegeben ist) bis zum letzten Element (da kein Endindex angegeben ist) in 2-er Schritten abgefragt (*jedes 2. Element der Liste*)

DHBW Duale Hochschule Baden-Württemberg

Auslesen von Werten (IV)

Mithilfe von Slicing kann bspw. auch ganz einfach die Reihenfolge einer Liste umgekehrt werden:

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7]
>>> numbers[::-1]
[7, 6, 5, 4, 3, 2, 1, 0]
```

Hier wird vom Anfang der Liste bis zum Ende der Liste in Schritten der Größe *-1*, also rückwärts, gegangen.

Löschen von Werten (I)

Neben der Möglichkeit, Werte zu einer Liste hinzuzufügen, können die Werte auch wieder gelöscht/entfernt werden, da Listen generell veränderbar sind.

Auch hierfür stehen unterschiedliche Möglichkeiten zur Verfügung

Löschen von Werten (II)

- `delete`-Operator: Um ein Element aus der Liste zu Löschen kann der `delete`-Operator genutzt werden. Hierfür steht das Schlüsselwort `del` zur Verfügung. Die Syntax ist:

```
del <Liste>[<Index>]
```

```
1 >>> names = ['Alice', 'Bob', 'Carol', 'Dave']
2 >>> del names[0]
3 >>> print(names)
4 ['Bob', 'Carol', 'Dave']
5 >>> del names[-1]
6 ['Bob', 'Carol']
```

Löschen von Werten (III)

- **pop**: Mithilfe der `pop()`-Funktion wird das Element entsprechend des übergebenen Indizes aus der Liste gelöscht und zurückgegeben. Wird kein Index explizit angegeben, wird das letzte Element der Liste gelöscht und zurückgegeben. Die Syntax ist:

`<Liste>.pop()` bzw. `<Liste>.pop(<Index>)`

```
1  >>> names = ['Alice', 'Bob', 'Carol', 'Dave']
2  >>> names.pop()
3  'Dave'
4  >>> print(names)
5  ['Alice', 'Bob', 'Carol']
6  >>> names.pop(1)
7  Bob
8  >>> print(names)
9  ['Alice', 'Carol']
```

Löschen von Werten (IV)

- **remove**: Mithilfe der `remove()`-Funktion wird das erste Element aus der Liste gelöscht, das dem übergebenen Element entspricht. Ebenso wie bei der Abfrage von Werten muss stets darauf geachtet werden, ob der Wert überhaupt in der Liste enthalten ist, da ansonsten ein Fehler auftritt. Die Syntax ist:

```
<Liste>.remove(<Element>)
```

```
1  >>> names = ['Alice', 'Bob', 'Carol', 'Dave']
2  >>> names.remove('Bob')
3  >>> print(names)
4  ['Alice', 'Carol', 'Dave']
5  >>> names.remove('Bob')
6  ValueError: list.remove(x): x not in list
```

in-Keyword

Um zu überprüfen, ob sich ein bestimmter Wert in einer Liste befindet (z.B. bevor die `remove()` -Funktion ausgeführt wird), gibt es in Python das Schlüsselwort `in`.
Die Auswertung ist dabei stets ein boolescher Wert: Falls das jeweilige Element in der Liste enthalten ist, wird `True` zurückgegeben, andernfalls `False`.
Die Syntax ist:

```
<Element> in <Liste>
```

```
>>> names = ['Alice', 'Bob', 'Carol', 'Dave']
>>> 'Bob' in names
True
>>> names.remove('Bob')
>>> print(names)
['Alice', 'Carol', 'Dave']
>>> 'Bob' in names
False
```

Elementanzahl (I)

Um zu überprüfen, wie viele Elemente in einer Liste enthalten sind, kann die `len()` -Funktion genutzt werden. Diese gibt an, wie viele Werte insgesamt in der Liste gespeichert sind, es handelt sich also immer um einen Integer-Wert.
Die Syntax ist:

```
len(<Liste>)
```

```
>>> names = ['Alice', 'Bob', 'Carol', 'Dave']
>>> len(names)
4
>>> names.remove('Bob')
>>> print(names)
['Alice', 'Carol', 'Dave']
>>> len(names)
3
```

Elementanzahl (II)

Achtung

Im Gegensatz zur mathematischen Zählweise bei der Indizierung wird bei der reinen Anzahl der Elemente die natürliche Zählweise genutzt. Wenn also in einer Liste 4 Elemente enthalten sind, ist die Anzahl der Elemente, die durch `len()` ermittelt wird, 4 und nicht 3.

Dies führt in Kombination mit der Indizierung häufig zu Verwirrung und stellt eine häufige Fehlerursache dar.

Hilfreiche Funktionen (I)

Da Listen universell und flexibel einsetzbar sind, besitzen sie viele Hilfsfunktionen, die bereits in Python eingebaut sind und ohne weiteres vom Entwickler genutzt werden können.

Eine Auswahl dieser Hilfsfunktionen wird im Folgenden vorgestellt, ein Überblick über alle Listenfunktionen gibt die offizielle Dokumentation.

Hilfreiche Funktionen (II)

- **sort**: Mithilfe der `sort()`-Funktion können Listen sortiert werden.
  Die Syntax ist:

  ```
  <Liste>.sort()
  ```

  Sind in der Liste lediglich numerische Werte enthalten, so wird die Liste standardmäßig nach aufsteigende Werte sortiert:

```
1  >>> numbers = [3, 6, 1, 9, 18, 4]
2  >>> numbers.sort()
3  >>> print(numbers)
4  [1, 3, 4, 6, 9, 18]
```

Hilfreiche Funktionen (III)

Sind in der Liste lediglich Strings enthalten, so wird die Liste standardmäßig alphabetisch aufsteigend sortiert:

```
>>> addresses = ['Palm Springs 210', 'Sunset Blvd. 10', 'Eastriver Drive 50', '11th Street
 ↪  09']
>>> addresses.sort()
>>> print(addresses)
['11th Street 09', 'Eastriver Drive 50', 'Palm Springs 210', 'Sunset Blvd. 10']
```

Sind in der Liste unterschiedliche Datentypen enthalten, so kann die Liste nicht sortiert werden, da unklar ist, wie etwa Strings und Integer-Werte miteinander verglichen werden sollen [3]:

```
>>> names_and_ages = ['Alice', 23, 'Bob', 21, 'Carol', 34, 'Dave', 19]
>>> names_and_ages.sort()
TypeError: '<' not supported between instances of 'int' and 'str'
```

[3] In diesen Fällen kann der Entwickler eine eigene Vergleichs-/Sortierfunktion angeben

Hilfreiche Funktionen (IV)

Die Listen können auch absteigend sortiert werden, indem der Parameter `reverse=True` mit übergeben wird:

```
>>> numbers = [3, 6, 1, 9, 18, 4]
>>> numbers.sort(reverse=True)
>>> print(numbers)
[18, 9, 6, 4, 3, 1]
```

Hilfreiche Funktionen (V)

- **reverse**: Analog zum Sortieren kann die Reihung der Elemente innerhalb einer Liste auch invertiert werden, die Liste wird also "umgedreht". Hierfür gibt es die `reverse()` -Funktion.
  Die Syntax ist:

  `<Liste>.reverse()`

```
1  >>> names = ['Alice', 'Bob', 'Carol', 'Dave']
2  >>> names.reverse()
3  >>> print(names)
4  ['Dave', 'Carol', 'Bob', 'Alice']
```

Hilfreiche Funktionen (VI)

- **min/max/sum**: Auch mathematische Funktionen, die auf Listen angewandt werden können, sind vorhanden.

```
>>> numbers = [3, 6, 1, 9, 18, 4]
>>> max(numbers)
18
>>> min(numbers)
1
>>> sum(numbers)
41
```

Aufgaben I

1. Was sind Listen? Wofür werden sie genutzt? Was ist der Vorteil ggü. zuvor besprochener Datentypen?
2. Welche Eigenschaften besitzt eine Liste und was bedeuten diese?
3. Wie können Listen erzeugt werden?
4. Erzeugen Sie eine vorgefüllte Liste mit den Werten `1`, `3`, `5` und `7`
5. Fügen sie dieser Liste den Wert `9` an
6. Wie können mehrere Werte auf einmal angehängt werden? Fügen Sie die Werte `11`, `13` und `17` in einem Schritt an.
7. Da der Wert `15` ausgelassen wurde, soll dieser nun ebenfalls zur Liste hinzugefügt werden, sodass die Sortierung der Werte intakt bleibt.
8. Gegeben ist `x = [3,5,1,2,4,6]`. Welchen Wert hat `x[2]`? Welchen `x[-2]`?
9. Mit welchem Befehl können in der zuvor erzeugten Liste der ungeraden Zahlen die Zahlen von 5 - 13 abgefragt werden?
10. Wie kann diese Liste invertiert werden?
11. Wie kann jeder dritte Wert dieser Liste abgefragt werden?
12. Worin besteht der Unterschied zwischen `del`, `pop` und `remove`?
13. Was ist das Ergebnis von `[1,1,2,2,3,3].remove(2)`?

DHBW Duale Hochschule Baden-Württemberg

Aufgaben II

14. Wofür wird das `in` -Keyword genutzt? Was muss bei der Verwendung von Keywords beachtet werden?
15. Worauf muss bei der Indizierung und der Längenangaben von Listen geachtet werden?
16. Welche weitere Möglichkeit gibt es, die Reihenfolge einer Liste (permanent) umzukehren?
17. Was ist das Ergebnis?: `max(['Bert', 'cArl', 'adam'])`

Tuples - Definition

Neben den Listen existiert ein weiterer häufig verbreiteter Sequenzdatentyp in Python: Das Tuple. Ein Tuple verhält sich sehr ähnlich zu einer Liste.

Tuple

Ein Tuple ist eine unveränderbare, geordnete und indizierte Sammlung von Objekten

Der primäre Unterschied zu Listen besteht also darin, dass Tuples unveränderbar sind, Werte können also nicht überschrieben und bestehende Tuples nicht abgeändert werden.

DHBW Duale Hochschule Baden-Württemberg

Einsatz von Tuples

Ebenso wie Listen können auch in einem Tuple unterschiedliche Datentypen enthalten sein.

Aufgrund der Unveränderbarkeit der Werte werden Tuples häufig als Rückgabewerte von Funktionen genutzt, sofern mehrere Werte zurückgegeben werden sollen.

Erzeugung (I)

Ein Tuple kann auf unterschiedliche Arten erzeugt werden:

- `tuple()` -Konstruktor: Mithilfe des Befehls `tuple()` kann ein neues, zunächst leeres Tuple erzeugt werden.

```
1 >>> new_tuple = tuple()
2 >>> print(new_tuple)
3 ()
4 >>> type(new_tuple)
5 <class 'tuple'>
```

Erzeugung (II)

- Kurzschreibweise: Elemente, die in einem Tuple enthalten sind, werden in Python durch runde Klammern (`( )`) umgeben. Als Kurzschreibweise kann anstatt `tuple()` daher auch einfach `()` genutzt werden.

```
1  >>> new_tuple = ()
2  >>> print(new_tuple)
3  ()
4  >>> type(new_tuple)
5  <class 'tuple'>
```

Speichern von Werten (I)

In den beiden vorherigen Beispielen wurde gezeigt, wie Tuples erzeugt werden können, jedoch enthalten diese Tuples noch keine Werte, sind also *leer*. Um Werte in den Tuples abzuspeichern gibt es die Möglichkeit, Werte direkt bei der Erzeugung anzugeben.

Speichern von Werten (II)

- **Erzeugung eines vorgefüllten Tuples:** Bei der Initialisierung eines neuen Tuples können direkt Objekte angegeben werden, die in diesem Tuple gespeichert werden sollen:

```
>>> names = ('Alice', 'Bob', 'Carol', 'Dave')
>>> print(names)
('Alice', 'Bob', 'Carol', 'Dave')
>>> type(names)
<class 'tuple'>
```

Bei der Erzeugung von Tuples können die Klammern auch weggelassen werden. Jedoch sollten sie aus Gründen der Übersichtlichkeit immer genutzt werden.

```
>>> names = 'Alice', 'Bob', 'Carol', 'Dave'
>>> type(names)
<class 'tuple'>
```

DHBW Duale Hochschule Baden-Württemberg

Speichern von Werten (III)

Soll ein Tuple mit lediglich einem Wert angelegt werden, ist das Komma trotzdem notwendig, um zwischen einer einfachen Wertzuweisung und einer Erzeugung eines Tuples unterscheiden zu können.

```
>>> name_without_comma = ('Alice')
>>> name_with_comma = ('Alice',)
>>> type(name_without_comma)
<class 'str'>
>>> type(name_with_comma)
<class 'tuple'>
```

Speichern von Werten (IV)

Achtung

Das gleichzeitige Erzeugen eines Tuples und Speichern von Werten in ebendiesem Tuple ist nur mit der Kurzschreibweise möglich. Wird ein Tuple mit tuple() erzeugt, können die Werte nicht direkt im Tuple gespeichert werden, auch ein späteres Hinzufügen ist nicht möglich (da Tuples unveränderbar sind).

Speichern von Werten (V)

Achtung

Funktionen wie `append()`, `insert()` oder `extend()` sind bei Tuples nicht vorhanden, da Tuples grundsätzlich unveränderbar sind.

Tuple-Operationen (I)

Es können verschiedene Operationen verwendet werden, um neue Tuple zu erzeugen:

- **Tuple Konkatenation**: Mithilfe des `+`-Operators können zwei Tuple konkateniert (verbunden) werden. Da die Tuple unveränderbar sind, entsteht dadurch ein neues, drittes Tuple

```
1  >>> names1 = ('Alice', 'Bob')
2  >>> names2 = ('Carol', 'Dave')
3  >>> names3 = names1 + names2
4  >>> print(names3)
5  ('Alice', 'Bob', 'Carol', 'Dave')
6  >>> type(names3)
7  <class 'tuple'>
```

Tuple-Operationen (II)

- **Multiple Konkatenation**: Mithilfe des `*`-Operators kann der Inhalt eines Tuples beliebig oft wiederholt werden. Da das Tuple unveränderbar ist, entsteht dadurch ein neues Tuple

```
1  >>> names1 = ('Alice', 'Bob')
2  >>> names2 = names1 * 4
3  >>> print(names2)
4  ('Alice', 'Bob', 'Alice', 'Bob', 'Alice', 'Bob', 'Alice', 'Bob')
5  >>> type(names2)
6  <class 'tuple'>
```

Auslesen von Werten

Das Auslesen der Tuple-Werte verhält sich identisch zum Auslesen von Werten aus Listen.

Achtung

Auch wenn Tuples mit runden Klammern erzeugt werden, erfolgt das Auslesen von Werten (wie bei allen Sequenz-Typen) mit eckigen Klammern!

Auch Tuples sind indexbasiert, die wieder 0-basiert sind. Es muss auch beachtet werden, dass der Tuple-Index bei der Abfrage nicht größer ist als die Länge des Tuples

```
>>> names = ('Alice', 'Bob', 'Carol', 'Dave')
>>> print(names[1])
'Bob'
>>> print(names[4])
IndexError: tuple index out of range
>>> print(names[-1])
'Dave'
```

Löschen von Werten

Da Tuples unveränderlich sind, können Werte aus einem Tuple nicht entfernt werden. Wird trotzdem versucht, einen Wert zu löschen, erscheint ein Fehler:

```
>>> names = ('Alice', 'Bob', 'Carol', 'Dave')
>>> del names[0]
TypeError: 'tuple' object doesn't support item deletion
```

Soll trotzdem ein Wert aus einem Tuple gelöscht werden, kann dies über einen Workaround gelöst werden: Es kann ein neues Tuple, bestehend aus den Werten des alten Tuples exklusive des zu löschenden Wertes, erzeugt werden:

```
>>> names = ('Alice', 'Bob', 'Carol', 'Dave')
>>> names_without_bob = (names[0],) + names[2:]
>>> print(names_without_bob)
('Alice', 'Carol', 'Dave')
```

DHBW Duale Hochschule Baden-Württemberg

in-Keyword

Ebenso wie bei den Listen kann mithilfe des `in`-Keywords festgestellt werden, ob ein Wert in einem Tuple enthalten ist.

```
>>> names = ('Alice', 'Bob', 'Carol', 'Dave')
>>> 'Bob' in names
True
```

Elementzahl

Ebenso wie bei den Listen kann mithilfe der `len()`-Funktion festgestellt werden, wie viele Werte in einem Tuple enthalten sind.

```
>>> names = ('Alice', 'Bob', 'Carol', 'Dave')
>>> len(names)
4
```

Hilfreiche Funktionen

Da Tuples unveränderbar sind, stehen hier keine der Funktionen zur Verfügung, die die Struktur der Tuples direkt verändern (z.B. `sort()`, `reverse()`, ...).

Dennoch gibt es auch hier Tuple-Funktionen, die die Struktur nicht beeinflussen, wie etwa `min()`, `max()`, `sum()`,...

```
>>> ages = (23, 52, 21, 33)
>>> max(ages)
52
>>> sum(ages)
129
```

Aufgaben I

1. Worin besteht der primäre Unterschied zwischen Listen und Tuples?
2. Wann werden Listen verwendet? Wann Tuples?
3. Welche Möglichkeiten gibt es, ein Tuple mit Werten zu füllen?
4. Welchen Zweck erfüllt der Tuple-Konstruktor?
5. Mit welchen der folgenden Ausdrücke wird ein Tuple erzeugt? Was ist jeweils das Ergebnis von `i[0]` ?:

```
>>> i = 0
>>> i[0]
```

```
>>> i = (0)
>>> i[0]
```

```
>>> i = (0),
>>> i[0]
```

Aufgaben II

- ```
  >>> i = (0,)
  >>> i[0]
  ```

- ```
  >>> i = (0,),
  >>> i[0]
  ```

- ```
  >>> i = 0,
  >>> i[0]
  ```

- ```
  >>> i = "0"
  >>> i[0]
  ```

- ```
  >>> i = "0,"
  >>> i[0]
  ```
  ```

Aufgaben III

6. Was ist bei Tuples bezüglich Kommas und Klammern zu beachten?
7. Wie kann dem Tuple `('Alice','Bob')` der Wert `'Carol'` angehängt werden?
8. Welche Ausgabe erzeugt folgender Befehl?:

```
1 >>> tuple1 = ('na',)
2 >>> tuple2 = 'Batman',
3 >>> tuple1*10 + tuple2
```

9. Für den Zugriff auf Tuplewerte durch Indizes werden eckige Klammern verwendet. Warum werden diese nicht auch zur Erzeugung eines Tuples verwendet?
10. Wie können Werte aus einem Tuple gelöscht werden? Was ist dabei zu beachten?
11. Welche Funktionen sind bei Listen und Tuples anwendbar, welche lediglich bei Listen? Warum gilt diese Einschränkung?

DHBW Duale Hochschule Baden-Württemberg

Dictionaries - Definition

Zu den erweiterten Grunddatentypen gehören bei Python auch die sogenannten *Dictionaries*, die im deutschen auch seltener als *Wörterbücher* bezeichnet werden.

Dictionary

Ein Dictionary ist eine veränderbare ungeordnete Sammlung von Werten in einer Key-Value Struktur

Key-Value Struktur

Eine *Key-Value Struktur* ist eine in der Programmierung häufig genutzte Struktur, um eindeutige Relationen zwischen einem Schlüsselwert (**Key**) und einer entsprechenden Wertzuweisung (**Value**) herzustellen.

Beispiel

Relation Länder →Hauptstädte:
"Deutschland" →"Berlin", "Frankreich" →"Paris", "Italien" →"Rom", ...

Relation Name →Körpergröße:
"Alice" →179, "Bob" →181, "Eve" →165, ...

Relation Smarthome-Element →Zustand:
"Licht an" →True, "Fenster offen" →False, "Türe geschlossen" →True, ...

Eigenschaften

Ebenso wie Listen und im Gegensatz zu Tuples sind Dictionaries veränderbar, d.h. Werte innerhalb der Dictionaries können überschrieben und die Struktur der Dictionaries verändert werden, ohne dafür explizit ein neues Objekt erzeugen zu müssen.

Im Gegensatz zu Listen und Tuples sind Dictionaries ungeordnet, d.h. die Elemente liegen nicht zwingend in der Reihenfolge vor, in der sie angelegt wurden. [4]
Da Dictionaries im Gegensatz zu den Sequence-Types nicht indexbasiert, sondern keybasiert ist, ist die Reihenfolge in den meisten Fällen irrelevant.

[4] Erst ab Python 3.7 sind Dictionaries immer *insertion-ordered*

Eigenschaften von Keys und Values

Die Keys und Values eines Dictionaries müssen gewisse Eigenschaften erfüllen:

- **Keys:**
  - **Uniqueness:** Keys müssen *unique* (eindeutig) sein. Es darf kein Wert mehrmals als Key innerhalb eines Dictionaries verwendet werden
  - **Hashable:** Keys müssen *immutable* (unveränderbar) sein. Basisdatentypen (int, float, string, ...) sind also erlaubt, ebenso reine Tuples. Listen und Dictionaries sind veränderbar, daher können sie nicht als Key genutzt werden.
  - **Numerische Gleichheit:** Sofern zwei Zahlen als *gleich* betrachtet werden, verweisen sie auf den selben Key (Bsp.: Da in Python gilt `1 == 1.0`, stellen beide Zahlen den gleichen Key dar)
- **Values:**
  - Annähernd alle Objekte und Datentypen dürfen als Values innerhalb eines Dictionaries verwendet werden, z.B. Grunddatentypen (int, float, string, ...), erweiterte Datentypen (Sequenzen (Listen, Tuples), Maps (Dictionaries), Sets), Klassen, Objekte, ...

DHBW Duale Hochschule Baden-Württemberg

Erzeugung (I)

Ein Dictionary kann auf unterschiedliche Arten erzeugt werden:

- `dict()`-Konstruktor: Mithilfe des Befehls `dict()` kann ein neues, zunächst *leeres* Dictionary erzeugt werden

```
>>> new_dictionary = dict()
>>> print(new_dictionary)
{}
>>> type(new_dictionary)
<class 'dict'>
```

Erzeugung (II)

- Kurzschreibweise: Elemente, die in einem Dictionary enthalten sind, werden in Python durch geschweifte Klammern (`{ }`) umgeben. Als Kurzschreibweise kann anstatt `dict()` daher auch einfach `{}` genutzt werden

```
>>> new_dictionary = {}
>>> print(new_dictionary)
{}
>>> type(new_dictionary)
<class 'dict'>
```

Speichern von Werten (I)

In beiden vorherigen Beispielen wurde gezeigt, wie Dictionaries erzeugt werden können, jedoch enthalten diese Dictionaries noch keine Werte, sind also *leer*. Um Werte in den Listen abzuspeichern, gibt es unterschiedliche Möglichkeiten.

Speichern von Werten (II)

- **Erzeugung eines vorgefüllten Dictionaries:** Bei der Initialisierung eines neuen Dictionaries können direkt Objekte angegeben werden, die in diesem Dictionary gespeichert werden sollen:

```
1  >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2  >>> print(capitals)
3  {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
4  >>> type(capitals)
5  <class 'dict'>
```

Hier wird ein Dictionary erzeugt, das jeweils einem Land (Key) eine Hauptstadt (Value) zuweist. Die Syntax eines Dictionaries sieht folgendermaßen aus:

```
{<Key>: <Value>, <Key>: <Value>, ...}
```

Einzelne Einträge werden also durch Komma getrennt, Values werden den Keys durch Doppelpunkt zugewiesen.

Speichern von Werten (III)

Achtung

Das gleichzeitige Erzeugen eines Dictionaries und Speichern von Werten in ebendiesem Dictionary ist nur mit Kurzschreibweise möglich. Wird ein Dictionary mit `dict()` erzeugt, können die Werte nicht direkt im Dictionary gespeichert werden, hierzu ist ein extra Schritt notwendig.

Speichern von Werten (IV)

- **insert:** Anders als bei den Listen gibt es keine dedizierten `append()`, `insert()`, ... Funktionen, sondern um einen Wert im Dictionary zu ergänzen bzw. zu überschreiben, genügt folgender Wertzuweisung:

  `<Dictionary>[<Key>] = <Value>`

  Dies entspricht der normalen Wertzuweisung von Variablen.

```
1  >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2  >>> print(capitals)
3  {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
4  >>> capitals['Spain'] = 'Barcelona'
5  >>> print(capitals)
6  {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome', 'Spain': 'Barcelona'}
7  >>> capitals['Spain'] = 'Madrid'
8  >>> print(capitals)
9  {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome', 'Spain': 'Madrid'}
```

Speichern von Werten (V)

- **update:** Darüber hinaus existiert auch die `update()` -Methode, um Werte dem Dictionary hinzuzufügen bzw. bestehende Werte zu aktualisieren. Die Syntax ist:

  `<Dictionary>.update(<Dictionary>)`

  Falls das jeweilige Key-Value-Paar im Dictionary bereits vorhanden ist, wird es überschrieben, andernfalls wird es ergänzt.

```
1  >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome', 'Spain':
   ↪  'Barcelona'}
2  >>> print(capitals)
3  {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome', 'Spain': 'Barcelona'}
4  >>> capitals.update({'Spain':'Madrid', 'Austria': 'Vienna'})
5  >>> print(capitals)
6  {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome', 'Spain': 'Madrid',
   ↪  'Austria': 'Vienna'}
```

Auslesen von Werten (I)

Ebenso wie bei Listen und Tuples können Werte aus den Dictionaries mit eckigen Klammern abgefragt werden. Jedoch sind Dictionaries nicht indexbasiert, sondern keybasiert. D.h. auf die jeweiligen Werte kann durch den zugehörigen Key zugegriffen werden.

```
>>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
>>> print(capitals['Germany'])
'Berlin'
```

Auslesen von Werten (II)

Achtung

Der Programmierer muss stets darauf achten, einen validen Key anzugeben. Ist der jeweilige Key nicht im Dictionary enthalten, erscheint ein Fehler

```
>>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
>>> print(capitals['Norway'])
KeyError: 'Norway'
```

Auslesen von Werten (III)

Achtung

Im Dictionary können Wert lediglich anhand des Keys abgefragt werden. Eine Abfrage nach Wert ist nicht möglich (Analogie: Wörterbuch)

```
>>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
>>> print(capitals['Berlin'])
KeyError: 'Berlin'
```

Auslesen von Werten (IV)

Dem Programmierer steht darüber hinaus die `get()` -Methode zur Verfügung. Ebenso wie bei der zuvor gezeigten key-basierten Abfrage mithilfe eckiger Klammern können so die Werte zu einem Key abgefragt werden. Außerdem kann ein Default-Wert angegeben werden, der als Ergebnis zurückgegeben wird, falls der Key nicht vorhanden ist.

```
>>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
>>> capital = capitals.get('France')
>>> print(capital)
'Paris'
>>> capital = capitals.get('Norway')
>>> print(capital)
None
>>> capital = capitals.get('Norway', 'nicht enthalten')
>>> print(capital)
'nicht enthalten'
```

Löschen von Werten (I)

Neben der Möglichkeit, Werte zu einem Dictionary hinzuzufügen, können die Werte auch wieder gelöscht/entfernt werden, da Dictionaries generell veränderbar sind.

Auch hierfür stehen unterschiedliche Möglichkeiten zur Verfügung

Löschen von Werten (II)

- delete-Operator: Um ein Element aus der Liste zu löschen kann der delete-Operator genutzt werden. Hierfür steht das Schlüsselwort `del` zur Verfügung. Die Syntax ist:

```
del <Dictionary>[<Key>]
```

```
1 >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2 >>> print(capitals)
3 {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
4 >>> del capitals['Germany']
5 >>> print(capitals)
6 {'France': 'Paris', 'Italy': 'Rome'}
```

Löschen von Werten (III)

- **pop**: Mithilfe der `pop()` -Methode wird ein Key-Value-Paar anhand des Key-Wertes aus dem Dictionary gelöscht. Zudem wird der Wert des gelöschten Paares zurückgegeben. Die Syntax ist:

```
<Dictionary>.pop(<Key>)
```

```
1 >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2 >>> print(capitals)
3 {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
4 >>> removed_capital = capitals.pop('France')
5 >>> print(removed_capital)
6 'Paris'
```

Löschen von Werten (IV)

- **popitem**: Mithilfe der `popitem()` -Methode wird ein zufälliges Key-Value-Paar aus dem Dictionary gelöscht. Zudem wird das Key-Value-Paar des gelöschten Elements als Tuple zurückgegeben. Die Syntax ist:

```
<Dictionary>.popitem()
```

```
1 >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2 >>> print(capitals)
3 {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
4 >>> removed_capital = capitals.popitem()
5 >>> print(removed_capital)
6 ('Italy', 'Rome')
```

Löschen von Werten (V)

- **clear**: Mithilfe der `clear()` -Methode werden alle Key-Value-Paare aus dem Dictionary gelöscht. Die Syntax ist:

```
<Dictionary>.clear()
```

```
1 >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2 >>> print(capitals)
3 {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
4 >>> capitals.clear()
5 >>> print(capitals)
6 {}
```

in-Keyword

Ebenso wie bei Listen und Tuples, kann das `in` -Keyword genutzt werden, um zu überprüfen, ob sich ein Key in einem Dictionary befindet.

```
>>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
>>> 'Italy' in capitals
True
>>> 'Berlin' in capitals
False
```

DHBW Duale Hochschule Baden-Württemberg

Elementanzahl

Um zu überprüfen, wie viele Key-Value-Paare sich in einem Dictionary befinden, kann die `len()` -Funktion genutzt werden.

```
>>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
>>> len(capitals)
3
>>> capitals.clear()
0
```

Hilfreiche Funktionen (I)

Da Dictionaries universell und flexibel einsetzbar sind, besitzen sie viele Hilfsfunktionen, die bereits in Python eingebaut sind und ohne weiteres vom Entwickler genutzt werden können.

Eine Auswahl dieser Hilfsfunktionen wird im Folgenden vorgestellt, ein Überblick über alle Dictionaryfunktionen gibt die offizielle Dokumentation

Hilfreiche Funktionen (II)

- **items:** Mithilfe der `items()` -Methode werden alle Key-Value-Paare in einem sogenannten *View-Objekt*[5] zurückgegeben. Jedes Key-Value-Paar wird durch ein Tuple repräsentiert. Die Syntax ist:

```
<Dictionary>.items()
```

```
1 >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2 >>> items = capitals.items()
3 >>> print(items)
4 dict_items([('Germany', 'Berlin'), ('France', 'Paris'), ('Italy', 'Rome')])
```

[5] View-Objekte stellen eine dynamische Repräsentation der Dictionaries dar. Sie können wie Listen verwendet werden

Hilfreiche Funktionen (III)

- **keys:** Mithilfe der `keys()` -Methode werden alle Keys des Dictionaries in einem *View-Objekt* zurückgegeben. Die Syntax ist:

```
<Dictionary>.keys()
```

```
1 >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2 >>> keys = capitals.keys()
3 >>> print(keys)
4 dict_keys(['Germany', 'France', 'Italy'])
```

Hilfreiche Funktionen (IV)

- **values:** Mithilfe der `values()` -Methode werden alle Values des Dictionaries in einem *View-Objekt* zurückgegeben. Die Syntax ist:

```
<Dictionary>.values()
```

```
1 >>> capitals = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2 >>> values = capitals.values()
3 >>> print(values)
4 dict_values(['Berlin', 'Paris', 'Rome'])
```

Hilfreiche Funktionen (V)

Auf Dictionaries (bzw. deren Werte) können auch Mengenoperationen durchgeführt werden. Die folgenden Beispiele sind jeweils für die `items()` -Methode anwendbar.

- **Schnittmenge:** Mithilfe des `&` -Operators können Schnittmengen zwischen Dictionaries erstellt werden. Das Ergebnis der Schnittmengen-Operation sind diejenigen Elemente, die in beiden Dictionaries vorhanden sind (**AND**-Verknüpfung).

```
1  >>> capitals1 = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2  >>> capitals2 = {'Germany': 'Berlin', 'Spain': 'Madrid'}
3  >>> intersection =  capitals1.items() & capitals2.items()
4  >>> print(intersection)
5  {('Germany', 'Berlin')}
```

Hilfreiche Funktionen (VI)

- **Vereinigung:** Mithilfe des `|`-Operators können Mengen-Vereinigungen erstellt werden. Das Ergebnis der Vereinigungs-Operation sind diejenigen Elemente, die in mindestens einem der beiden Dictionaries vorhanden sind (**OR**-Verknüpfung).

```
1  >>> capitals1 = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2  >>> capitals2 = {'Germany': 'Berlin', 'Spain': 'Madrid'}
3  >>> union =  capitals1.items() | capitals2.items()
4  >>> print(union)
5  {('Spain', 'Madrid'), ('Italy', 'Rome'), ('Germany', 'Berlin'), ('France', 'Paris')}
```

Hilfreiche Funktionen (VII)

- **Symmetrische Differenz:** Mithilfe des `^`-Operators können symmetrische Differenzen erstellt werden. Das Ergebnis der Operation sind diejenigen Elemente, die in genau einem der beiden Dictionaries vorhanden sind, aber nicht in beiden (**XOR**-Verknüpfung).

```
1 >>> capitals1 = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2 >>> capitals2 = {'Germany': 'Berlin', 'Spain': 'Madrid'}
3 >>> symm_diff =  capitals1.items() ^ capitals2.items()
4 >>> print(symm_diff)
5 {('Spain', 'Madrid'), ('Italy', 'Rome'), ('France', 'Paris')}
```

Hilfreiche Funktionen (VIII)

- **Differenz:** Mithilfe des `-`-Operators können Differenzen erstellt werden. Das Ergebnis der Differenz-Operation sind diejenigen Elemente, die im ersten Dictionary, nicht aber im zweiten Dictionary enthalten sind (**not**-Operation).

```
1  >>> capitals1 = {'Germany': 'Berlin', 'France': 'Paris', 'Italy': 'Rome'}
2  >>> capitals2 = {'Germany': 'Berlin', 'Spain': 'Madrid'}
3  >>> diff =  capitals1.items() - capitals2.items()
4  >>> print(diff)
5  {('Italy', 'Rome'), ('France', 'Paris')}
```

Aufgaben I

1. Welcher Analogie folgen die Dictionaries in Python bzw. allgemein Key-Value-Stores in der Programmierung?
2. Worin unterscheiden sich Dictionaries von Listen bzw. Tuples? Was sind die Gemeinsamkeiten?
3. Was wird durch eine Key-Value-Struktur repräsentiert?
4. In welchen anderen Bereichen der bereits besprochenen Themengebiete treten Key-Value-Strukturen auf?
5. Warum sind Dictionaries nicht zwingend geordnet?
6. Welche Eigenschaften müssen die Keys eines Dictionaries erfüllen? Welche die Values? Warum sind die Regeln der Values nicht so strikt?
7. Folgendes Snippet ist gegeben:

```
1  >>> x = {3.0: 'Alice'}
2  >>> x['Alice'] = 3.0
3  >>> x['Bob'] = 3
4  >>> x[3] = 'Bob'
```

Welchen Wert besitzt `x`?

Aufgaben II

8. Welche Unterschiede bestehen zwischen dem `dict()`-Konstruktor und der Kurzschreibweise `{}` zum Erzeugen eines Dictionaries?
9. Warum gibt es im Gegensatz zur Liste, die ebenfalls veränderbar ist, keine Basisfunktion wie `append()` zum Hinzufügen neuer Werte zum Dictionary?
10. Welchen Vorteil bietet die `update()`-Methode gegenüber der normalen Wertzuweisung?
11. Was passiert, wenn mit `update()` ein Key-Value-Paar aktualisiert werden soll, das im Dictionary nicht existiert?
12. Was passiert, wenn Key-Value-Paare anhand ihres Values ausgelesen werden?
13. Kann bei Dictionaries ebenso wie bei Listen und Tuples per negativen Index zugegriffen werden? Ist Slicing möglich¿
14. Welche Möglichkeiten gibt es, einen Wert aus dem Dictionary auszulesen, wobei der Entwickler nicht sicher ist, ob der jeweilige Key existiert?
15. Wird mit dem `del`-Keyword lediglich der entsprechende Wert anhand des Schlüssels aus dem Dictionary gelöscht oder der komplette Key-Value-Paar?
16. Nach welcher Logik werden mit `popitem()` Elemente aus dem Dictionary entfernt? Wozu könnte diese Funktion hilfreich sein?

Aufgaben III

17. Bezieht sich das `in` -Keyword auf Keys oder Values?
18. Auf was bezieht sich `len()` -Funktion?
19. Wie kann eine Liste mit allen Values eines Dictionaries ausgegeben werden?
20. Auf welche boolesche Operationen beziehen sich die Schnittmenge, Vereinigung, symmetrische Differenz und Differenz von Dictionaries? Warum werden diese nicht durch die zuvor besprochenen Keywords ausgedrückt?

Sets - Definition

Sets verhalten sich sehr ähnlich zu Listen und Tuples. Der primäre Unterschied besteht in der Eigenschaft, dass Sets keine identischen Werte enthalten können.

Set

Ein Set ist eine *veränderbare* und *ungeordnete* Sammlung von *unterschiedlichen* (*"unique"*) Objekten.

Einsatz von Sets

Sets werden i.d.R. analog zu Listen verwendet, können jedoch eine Zeitersparnis darstellen bzw. eine Einhaltung der Eindeutigkeitslogik erzwingen, indem der Entwickler nicht selbst überprüfen muss, ob sich ein Element bereits in der Liste befindet, da die Eindeutigkeit durch Sets garantiert wird.

Erzeugung von Sets (I)

Auch Sets können auf unterschiedliche Arten erzeugt werden:

- `set()`-Konstruktor: Mithilfe des `set()`-Konstruktors kann ein neues Set-Objekt erzeugt werden

```
1  >>> new_set = set()
2  >>> print(new_set)
3  set()
4  >>> type(new_set)
5  <class 'set'>
```

Erzeugung von Sets (II)

- Kurzschreibweise: Als Kurzschreibweise können anstatt `set()` auch einfach geschweifte Klammern (`{ }`) genutzt werden

```
>>> new_set = {'Alice', 'Bob', 'Carol', 'Alice'}
>>> print(new_set)
{'Alice', 'Carol', 'Bob'}
>>> type(new_set)
<class 'set'>
```

Achtung

Für Sets werden ebenso wie für Dictionaries geschweifte Klammern genutzt. Falls es sich um eine reine Aneinanderreihung von Einzelwerten handelt, wird ein Set erzeugt. Werden Key-Value-Paare übergeben, wird ein Dictionary erzeugt. Da der Befehl `{ }` schon zur Erzeugung eines leeren Dictionaries verwendet wird, kann damit kein leeres Set erzeugt werden. Lediglich ein vorgefülltes Set kann mit der Kurzschreibweise erzeugt werden.

DHBW Duale Hochschule Baden-Württemberg

Speichern von Werten (I)

- **add**: Um einen (hashbaren) Wert der Liste hinzuzufügen, kann die `add()` -Methode genutzt werden. Der übergebene Wert wird nur dann hinzugefügt, falls er noch nicht in der Liste vorhanden ist. Die Syntax ist:

```
<Set>.add(<Element>)
```

```
1  >>> new_set = {'Alice', 'Bob', 'Carol', 'Alice'}
2  >>> print(new_set)
3  {'Alice', 'Carol', 'Bob'}
4  >>> new_set.add('Dave')
5  >>> print(new_set)
6  {'Alice', 'Dave', 'Carol', 'Bob'}
7  >>> new_set.add('Dave')
8  {'Alice', 'Dave', 'Carol', 'Bob'}
```

DHBW Duale Hochschule Baden-Württemberg

Speichern von Werten (I)

- **update**: Um mehrere (nicht-hashbare) Werte der Liste hinzuzufügen, kann die `update()` -Methode genutzt werden. Die übergebenen Werte werden nur dann hinzugefügt, falls sie noch nicht in der Liste vorhanden sind. Die Syntax ist:

```
<Set>.update(<Elements>)
```

```
1 >>> new_set = {'Alice', 'Bob', 'Carol', 'Alice'}
2 >>> print(new_set)
3 {'Alice', 'Carol', 'Bob'}
4 >>> new_set.update({'Dave'})
5 >>> print(new_set)
6 {'Carol', 'Alice', 'Bob', 'Dave'}
```

Speichern von Werten (I)

Achtung

Um einen einzelnen Wert zum Set hinzuzufügen, wird die `add()` -Methode genutzt. Ein Hinzufügen von mehreren Werten gleichzeitig ist damit nicht möglich, solange sie nicht-hashable (*veränderbar*) sind, z.B. Listen und Sets. Ein Hinzufügen von einem Tuple dagegen ist möglich (da unveränderbar).

Sollen mehrere Werte, die nicht-hashable sind, hinzugefügt werden, muss `update()` verwendet werden. Diese Methode akzeptiert eine unbegrenzte Anzahl von *Argumenten*. Dafür können hiermit keine "Einzelwerte" (nicht-iterierbar) hinzugefügt werden.

```
>>> new_set = {'Alice', 'Bob'}
>>> new_set.add('Carol')
>>> print(new_set)
{'Carol', 'Alice', 'Bob'}
>>> new_set.add(['Dave', 'Eve'])
TypeError: unhashable type: 'list'
>>> new_set.update(['Dave', 'Eve'])
>>> print(new_set)
{'Eve', 'Dave', 'Carol', 'Alice', 'Bob'}
>>> new_set.update(5)
TypeError: 'int' object is not iterable
```

Auslesen von Werten (I)

Da Sets weder index- noch keybasiert sind und darüber hinaus keine feste Ordnung besitzen, gibt es keine Abfrage von Werten, die mit dem Auslesen von Listen, Tuples oder Dictionaries vergleichbar ist.

Da aber eine reine Speicherung von Werten ohne der Möglichkeit, den Inhalt der Werte auslesen zu können, sinnlos ist, gibt es dennoch Wege, an die entsprechenden Werte zu gelangen.

Dies ist beispielsweise möglich, indem durch das Set *iteriert* wird. [6]

[6] Diese Funktionalität bezieht sich auf *Schleifen*, die im nächsten Foliensatz besprochen wird

Löschen von Werten (I)

- **discard**: Mit der `discard()` -Methode kann ein bestimmtes Element aus dem Set entfernt werden. Existiert der entsprechende Wert nicht im Set, wird der Aufruf ignoriert.
  Die Syntax ist:

```
<Set>.discard(<Element>)
```

```
1  >>> new_set = {'Alice', 'Bob', 'Carol'}
2  >>> print(new_set)
3  {'Carol', 'Alice', 'Bob'}
4  >>> new_set.discard('Bob')
5  >>> print(new_set)
6  {'Carol', 'Alice'}
7  >>> new_set.discard('Eve')
8  >>> print(new_set)
9  {'Carol', 'Alice'}
```

Löschen von Werten (II)

- **remove**: Mit der `remove()` -Methode kann ein bestimmtes Element aus dem Set entfernt werden. Existiert der entsprechende Wert nicht im Set, wird ein Fehler geworfen.
  Die Syntax ist:

```
<Set>.remove(<Element>)
```

```
1  >>> new_set = {'Alice', 'Bob', 'Carol'}
2  >>> print(new_set)
3  {'Carol', 'Alice', 'Bob'}
4  >>> new_set.remove('Bob')
5  >>> print(new_set)
6  {'Carol', 'Alice'}
7  >>> new_set.remove('Eve')
8  KeyError: 'Eve'
```

Löschen von Werten (III)

- **pop**: Mit der `pop()` -Methode kann ein zufälliges Element aus dem Set entfernt werden. Falls das Set leer ist, wird ein Fehler geworfen.
  Die Syntax ist:

```
<Set>.pop()
```

```
1  >>> new_set = {'Alice', 'Bob'}
2  >>> print(new_set)
3  {'Alice', 'Bob'}
4  >>> name = new_set.pop()
5  >>> print(new_set)
6  {'Bob'}
7  >>> print(name)
8  'Alice'
9  >>> new_set.pop()
10 >>> new_set.pop()
11 KeyError: 'pop from an empty set'
```

Löschen von Werten (IV)

- **clear**: Mit der `clear()`-Methode werden alle Elemente aus dem Set entfernt. Die Syntax ist:

```
<Set>.clear()
```

```
>>> new_set = {'Alice', 'Bob', 'Carol'}
>>> print(new_set)
{'Carol', 'Alice', 'Bob'}
>>> new_set.clear()
>>> print(new_set)
set()
```

in-Keyword

Auch bei Sets kann mit dem `in` -Operator überprüft werden, ob ein bestimmtes Element darin enthalten ist.

```
>>> new_set = {'Alice', 'Bob', 'Carol'}
>>> 'Alice' in new_set
True
>>> 'Eve' in new_set
False
```

Elementanzahl

Auch bei Sets kann mit der `len()` -Funktion überprüft werden, wie viele Elemente im Set enthalten sind.

```
>>> new_set = {'Alice', 'Bob', 'Carol'}
>>> len(new_set)
3
```

Hilfreiche Funktionen (I)

Auf Sets sind die gleichen Funktionen anwendbar, die auch auf Tuples angewandt werden können.

Hilfreiche Funktionen (II)

Auf Sets sind die gleichen Mengenoperationen anwendbar, die auch auf Dictionaries angewandt werden können.

```
>>> names1 = {'Alice', 'Bob', 'Carol'}
>>> names2 = {'Carol', 'Eve'}
>>> names1 & names2
'Carol'
>>> names1 | names2
{'Eve', 'Carol', 'Alice', 'Bob'}
>>> names1 ^ names2
{'Eve', 'Alice', 'Bob'}
>>> names1 - names2
{'Alice', 'Bob'}
```

Aufgaben I

1. Worin bestehen die grundlegenden Unterschiede zwischen Sets und den Collection- bzw. Map-Types?
2. Sind Sets index- oder keybasiert?
3. Wie kann in der Kurzschreibweise (ohne Verwendung von `set()`) ein leeres Set erzeugt werden? Was ist das Problem?
4. Wann wird die `add()`-Methode verwendet, wann `update()`?
5. Welche der beiden Methoden muss verwendet werden, wenn folgende Werten einem Set hinzugefügt werden sollen?:
   - `1`
   - `[1,2]`
   - `'Alice'`
   - `{'Alice', 'Bob'}`
   - `{'name1':'Alice', 'name2':'Bob'}`
   - `True`
6. Wie wird auf einzelne Werte eines Sets zugegriffen? Was ist das Problem? Wie könnte das Problem gelöst werden?
7. Was ist der Unterschied zwischen `discard()`, `remove()` und `pop()`?

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Bibliographie I