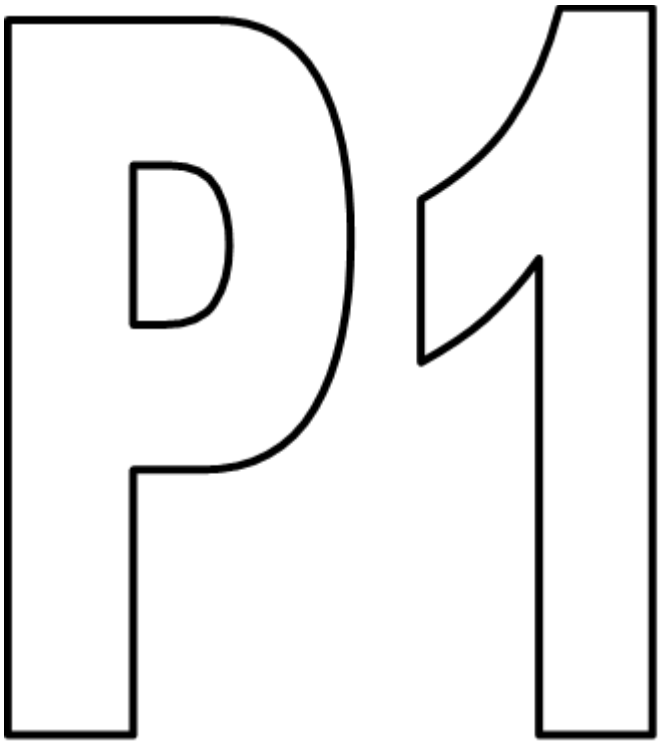
Softwaretechnologie und Datenmanagement

Unterstufe

Einführung Python



Inhaltsverzeichnis

1 Vorstellung Python

Python (aktuelle Hauptversion Python 3) wurde 1994 von Guido van Rossum veröffentlicht und ist eine beliebte und aktuelle Skriptsprache mit einem breiten Anwendungsbereich. Python ist für alle gängigen Plattformen (Windows, Linux, Mac usw.) frei verfügbar. Zum Einsatz kommt Python u.a. bei der Verarbeitung von Big Data (Data Analysis, Maschine Learning) und bei der (serverseitigen) Webentwicklung.

2 Entwicklungsumgebung und Infos zu Python

Eingesetzte Entwicklungsumgebung (IDE) am HNBK: PyCharm (Community Edition von JetBrain)
Online Interpreter über: https://repl.it

Angebot von Online Kursen und Tutorials:

https://www.edukatico.org/de/report/python-programmieren-lernen-liste-der-wichtigsten-online-kurse	https://www.python-kurs.eu/kurs.php	https://www.w3schools.com/python/default.asp

3 Python Schnellstart

Das erste kleine Programm (, um zu testen, dass alles läuft):

Datei: Start.py

```
# erstes Skript
print ("Hello world")
```

Python Programme (=Skripte) sind Textdateien mit Endung .py. Mit print kann eine Ausgabe erzeugt werden. Texte werden hierbei in Anführungszeichen gesetzt. Das # leitet einen Kommentar ein.

4 Kommentare

Kommentare dienen der Quellcode-Erläuterung und verbessern die Lesbarkeit und damit Wartbarkeit der Skripte. Sie werden bei der Ausführung eines Skriptes übersprungen.

```
# dies ist ein Kommentar
```

5 Variable und Datentypen

Eine **Variable** ist im allgemeinsten Sinne ein Behälter bzw. Speicher zum Aufbewahren von Werten. Diese Werte können z. B. Zahlen oder Zeichenketten sein. Im Laufe des Programmes kann man die dort abgelegten Werte auslesen oder einen neuen Wert zuweisen. Jede Variable hat hierfür einen eindeutigen Namen, über den sie angesprochen werden kann.

ACHTUNG: Eine **Konstante**, also eine Variable, der nach erster Wertzuweisung kein anderer Wert mehr zugewiesen werden kann, existiert nicht in Python.

Ein **Datentyp** spezialisiert den Speicher einer Variablen. Er bestimmt die Größe des Speicherbereiches, die Art der gespeicherten Werte (z. B. Zahl, Zeichenkette usw.) und was man mit diesen Werten machen kann (z. B. Addieren bei Zahlen).

Variablen in Python haben keinen festgeschriebenen Datentyp. Eine Variable wird erzeugt, wenn man ihr zum ersten Mal einen Wert zuweist (Initialisierung) und kann durch Werte verschiedenster Datentypen anschließend überschrieben werden. Der Speicher und damit der Datentyp passen sich dynamisch an.

Datentypen in Python:
Es kann grundsätzlich zwischen sogenannten **primitive**n Datentypen (d. h. der Speicher enthält maximal einen Wert) oder komplexen **Datentypen** (d.h. der Speicher kann mehrere Werte enthalten) unterschieden werden. Beispiele für komplexe Datentypen in Python sind z. B. die Klassen str, tuple, list usw. (siehe unten).

Primitive Datentypen	
int float complex bool	Ganze Zahl Zahl mit Nachkommastellen (Fließkommazahl) Komplexe Zahl (z.B. i) Enthält **True** oder **False** als Werte
Komplexe Datentypen (Details siehe letztes Kapitel)	
str	Folge von einzelnen Zeichen, die indiziert sind. Sowohl die einzelnen Zeichen als auch die Anzahl der Zeichen kann nicht verändert werden.
tuple	Folge von einzelnen Werten, die indiziert sind. Sowohl die einzelnen Werte als auch die Anzahl der Werte kann nicht verändert werden.
list	Folge von einzelnen Werten, die indiziert sind. Sowohl die einzelnen Werte als auch die Anzahl der Werte kann verändert werden.
set	Menge von Werten, deren Werte nicht indiziert sind. Werte sind nicht veränderbar. Anzahl der Werte kann verändert werden
dict	Folge von einzelnen Schlüsseln (keys) mit zugehörigen Werten (values). Beides und auch die Anzahl der Elemente kann verändert werden.

Variablen und Datentypen

Der Datentyp wird ermitteln mit der Funktion type()	`zahl = 5` `print (type(zahl))  # Ausgabe: <type 'int'>`

An der Wertzuweisung erkennt Python den Datentyp	`zahl = 5.5;  # = float`
	`x = True  # = bool`
	`text1 = "Hallo"  # = str`
	`text2 = 'Hallo'  # = str (wahlweise mit ' oder mit ")`

Der Datentyp wird dynamisch angepasst	`zahl = 5`
	`print (type(zahl))  # Ausgabe: <type 'int'>`
	`zahl = "hallo"`
	`print (type(zahl))  # Ausgabe: <type 'str'>`

Namen von Variablen

Syntax von Namen	• Buchstaben (a-z, A-Z), Ziffern (0-9) und "_" (Unterstrich). • nicht mit einer Ziffer beginnen. • keine Umlaute (ä, ö, ü) und weitere Sonderzeichen. • keine Übereinstimmung mit Schlüsselwörtern von Python. ACHTUNG: Namen sind case-sensitiv (d. h. es wird zwischen Groß- und Kleinschreibung unterschieden
Konventionen für Namen	• sprechende Namen – verdeutlichen gespeicherten Wert. • klein schreiben - bei zusammengesetzten Wörtern angehängte Wörter groß beginnen lassen (z. B. anzahlAutos). Dies bezeichnet man als camel case Schreibweise.

Casting = Festlegen eines Datentyps

Casting auf ganze Zahl mit int()	`a = int(10)  # a ist 10`
	`b = int(20.6)  # b ist 20`
	`c = int("30")  # c ist 30`

Casting auf Fließkommazahl mit float()	`a = float (10)  # a ist 10.0`
	`b = float (20.6)  # b ist 20.6`
	`c = float("30")  # c ist 30.0`

Casting auf Zeichenketten mit str()	`a = str("a1")  # a ist "a1"`
	`b = str(2)  # b ist "2"`
	`c = str(3.0)  # c ist "3.0"`

6 Ausgabe und Eingabe

Ausgabe mit print

```
zahl1 = 4.5
zahl2 = 4
text1 = "Hallo"
text2 = 'Python'

# Ausgabe einer Variablen
  print (zahl1)                                  # 4.5

# Ausgabe von mehreren Variablen (inklusive Leerzeichen!)
  print (text1, zahl2)                           # Hallo 4

# Ausgabe eines Textes
  print ("Jetzt starten")                        # Jetzt starten
```

Ausgabe-Kombinationen mit **+**

```
# Zeichenkette + Zeichenkette bedeutet Verknüpfung
  print (text1 + text2)                          # HalloPython
  print (text1 + "  " + text2)                   # Hallo Python

# Zahl + Zahl bedeutet Addition
  print (zahl1 + zahl2)                          # 8.5

# Zahl + Zeichenkette oder umgekehrt erzeugt einen FEHLER
  print (zahl1 + text1)                          # FEHLER
```

Ausgabe mit chr() (Zahl wird zur Ziffer)

```
# chr(Zahl) + Zeichenkette, für Ausgabe mit Zahlen und Zeichen
  print (chr(zahl1) + text1).                    # 4.5Hallo
```

Ausgabe mit Escape Zeichen

```
# Ausgabe von von " und \
  print ("Dies \\ ist \"wichtig\"")         # Dies \ ist "wichtig"
```

Ausgabe mit Zeilenumbruch

```
# Ausgabe mit \n (new line)
  print ("Dies ist \n in der 2. Zeile")     # Dies ist
                                            #  in der 2. Zeile
```

Ausgabe mit Platzhaltern und format()

```
preis = 5
obst = "Birnen"

# Platzhalter mit {Index}, beginnend mit 0 setzen
  text1 = "Der Preis ist {0}"
  print text1.format(preis)            # Der Preis ist 5

  text2 = "Die {0} kosten {1} EUR"
  print text2.format(obst, preis)      # Die Birnen kosten 5 EUR
```

Dezimalzahl mit 2 Nachkommazahlen

```
  text3 = "Die {0} kosten {1:.2f} EUR"
  print text3.format(obst, preis)      # Die Birnen kosten 5.00 EUR
```

Eingabe mit input ()

```
#  Mit Textaufforderung
  name = input ("Bitte Namen eingeben: ")
  print ("Eingabe: " + name)           # z.B. Eingabe: Peter

#  Ohne Textaufforderung
  name = input ()
  print ("Eingabe: " + name)           # z.B. Eingabe: Peter
```

7 Ausdrücke und Operatoren (ohne Bitoperatoren)

Ausdrücke gehören zu den kleinsten ausführbaren Einheiten eines Programms. Sie dienen dazu, Variablen einen Wert zuzuweisen, numerische Berechnungen durchzuführen oder logische Bedingungen zu formulieren.

7.1 Arithmetische Operatoren

Diese Operatoren führen mathematische Berechnungen durch. Sie erwarten eine Ganzzahl (Integer) oder eine Fließkommazahl als Operanden und liefern ein numerisches Ergebnis zurück.

Operator	Name	Beispiel	
+	Addition	a + b	
-	Subtraktion	a - b	
*	Multiplikation	a * b	
**	Exponentiation	a** b	
/	Division	a / b	
//	Division ohne Rest	a // b	Nachkommastellen werden abgeschnitten
%	Modulo	a % b	Rest der ganzzahligen Division von a und b. Beispiel: 4 % 2 ergibt 0 oder 41 % 10 ergibt 1

7.2 Zeichenkettenoperator

Zeichenketten-Operatoren werden verwendet, um Zeichenketten miteinander zu verknüpfen.

Operator	Name	Beispiel

+	Verkettung	`a = "Hallo"`
		`b = "Python"`
		`print(a + b)  # HalloPython`

7.3 Zuweisungsoperatoren

Zuweisungsoperatoren werden verwendet, um einer Variablen einen Wert zu zuweisen.

Operator	Bedeutung
=	Vom Operator rechtsstehende Werte werden einer linksstehenden Variablen zugewiesen.
	Kurzform für folgende Zuweisungen:
+=	`a += b bedeutet a = a + b`
-=	`a -= b bedeutet a = a – b`
*=	`a *= b bedeutet a = a * b`
**=	`a **= b bedeutet a = a ** b`
/=	`a /= b bedeutet a = a / b`
//=	`a //= b bedeutet a = a // b`
%=	`a %= b bedeutet a = a % b`

7.4 Vergleichsoperatoren

Diese Operatoren vergleichen Ausdrücke miteinander und liefern als Ergebnis entweder TRUE (wahr) oder FALSE (falsch). Vergleichsoperatoren werden bei der Formulierung von Bedingungen (s. Kontrollstrukturen) benötigt.

Operator	Name	Bedeutung
==	Gleichheit	a == b ergibt TRUE, wenn a und b denselben Wert enthalten
!=	Ungleichheit	a != b ergibt TRUE, wenn a und b ungleich sind.
<	Kleiner	a < b ergibt TRUE, wenn a kleiner ist als b.
>	Größer	a > b ergibt TRUE, wenn a größer ist als b.
<=	Kleiner gleich	a <= b ergibt TRUE, wenn a kleiner oder gleich ist als b.
>=	Größer gleich	a >= b ergibt TRUE, wenn a größer oder gleich ist als b.

7.5 Logische Operatoren

Mit logischen Operatoren lassen sich Ausdrücke logisch verknüpfen. Wie bei den Vergleichsoperatoren liefert auch die Auswertung mit logischen Operatoren ein Ergebnis TRUE oder FALSE. Sie werden benötigt um Bedingungen zu formulieren, die aus mehr als einer einzelnen Bedingung bestehen (sogenannte komplexe Bedingung).

Operator	Name	Bedeutung
not	NICHT	not a ergibt die Umkehrung des Wahrheitswertes
and	UND	a and b ergibt nur TRUE wenn sie beide TRUE sind. Andere Kombinationen sind FALSE.
or	ODER	a or b ergibt nur FALSE wenn sie beide FALSE sind. Andere Kombinationen sind TRUE.

7.6 Rangfolge von Operatoren

Zusammengesetzte Ausdrücke werden nach bestimmten Regeln abgearbeitet bzw. errechnet. Operatoren mit höherem Rang werden vor direkt benachbarten Operatoren mit niedrigerem Rang ausgeführt. Sind zwei benachbarte Operatoren vom gleichen Rang, so werden sie von links nach rechts abgearbeitet.

Die Rangfolge der Operatoren kann grundsätzlich durch runde Klammern verändert werden. Dabei wird der in runde Klammern eingeschlossene Teil zuerst bearbeitet. Treten zwei oder mehrere runde Klammernpaare hintereinander auf, so werden sie von links nach rechts abgearbeitet. Treten die Klammernpaare verschachtelt auf, so werden sie von innen nach außen abgearbeitet.

Rangfolge	Operator	Erläuterung
1 (hoch)	( )	Klammern zur Gruppierung
2	not ++ --, (int) (float) (str)	Logisches Nicht, Prä- und Postin- /-dekrement, Datentypumwandlungen
3	* / % // **	Arithmetische Operatoren
4	+ -	Arithmetische Operatoren, Zeichenkettenoperator
5	< <= > >=	Vergleichsoperatoren
6	== !=	Vergleichsoperatoren
7	and	Logischer Operator
8	or	Logischer Operator
9 (niedrig)	= *= +* -= /= //= **=	Zuweisungsoperatoren

8 Kontrollstrukturen

Kontrollstrukturen bestimmen den Programmablauf. Man unterscheidet zwischen: Folge (Sequenz), Alternative (Verzweigung, Selektion), Wiederholung (Iteration, Schleifen) und Unterprogramme (Funktionen, Methoden). Bei Alternativen und Wiederholungen steuern **Bedingungen** (logischer Ausdruck) den konkreten Ablauf. Eine Bedingung liefert als Ergebnis entweder True (wahr) oder False (falsch) zurück.

ACHTUNG: In Python werden Anweisungen, die zu einem Block gehören an derselben Code-Einrückung erkannt!

8.1 Alternativen

Bezeichnung	Allgemeine Syntax	Beispiel
Bedingte Verarbeitung (if)	`if Bedingung:` `  Anweisung(en)`	`if note == "":` `  print ("Note fehlt")`

Einfache Alternative (if-else)	`if Bedingung:`
	`  Anweisung(en)1`
	`else:`
	`  Anweisung(en)2`
		`if note <= 4:`
		`  print ("bestanden")`
		`else:`
		`  print ("wiederholen")`

Mehrfache Alternative (if-elif…)	`if Bedingung 1:`
	`  Anweisung(en)`
	`elif Bedingung 2:`
	`  Anweisung(en)`
	
	`# weitere elif(s) möglich`
	
	`else: #kann auch fehlen`
	`  Anweisung(en)`
		`if  wert == 0:`
		`  print ("Null")`
		`elif wert < 0:`
		`  print ("negativ")`
		`else:`
		`  print ("positiv")`

Eine mehrfache Alternative (switch-case), wie bei anderen Programmiersprachen, existiert in Python nicht.

8.2 Wiederholungen

Bezeichnung	Allgemeine Syntax	Beispiel

Kopfgesteuerte Schleife (while)	`while Bedingung:`
	`  Anweisung(en)`
		`wert = -3;`
		`while wert < 0:`
		`  print (wert, end=' ')`
		`  wert += 1  # -3 -2 -1`
		
		`# end=' ' Werte sind nebeneinander`

Kopfgesteuerte Schleife, Zählschleife (for)	`for Variable in Struktur:`
	`  Anweisung(en)`
	
	`for Schleife dient zum Durchlaufen von Objekten eines komplexen Datentyps`
		`fach = ["AW", "IT","WG"]`
		`for x in fach:`
		`  print(x, end=' ')  # AW IT WG`

	`For Variable in range()`
	`Funktion`
	
	`1. Wert: Startwert`
	`2. Wert: Endwert (nicht enthalten)`
	`3. Wert: Schrittweite`
		`for x in range(4):`
		`  print(x, end=' ')  # 0 1 2 3`
		
		`for x in range(4, 8):`
		`  print(x, end=' ')  # 4 5 6 7`
		
		`for x in range(4, 12, 3):`
		`  print(x, end=' ')  # 4 7 10`

Eine fußgesteuerte Schleife, wie bei anderen Programmiersprachen, existiert in Python nicht.

9 Komplexe Datentypen

9.1 Übersicht

Bei Variablen mit einem komplexen Datentyp handelt es sich um eine Speicherstelle, an denen mehrere Werte gleichzeitig abgelegt sind. Der Vorteil, dass man Werte zusammenfasst und nicht für jeden Wert eine einzelne Variable erzeugt, ist, dass man die Werte auch zusammen (sozusagen als Gruppe) verarbeiten kann.

Für jede dieser komplexen Datentypen ist festgelegt,

- wie die Werte genau abgelegt sind (z.B. in einer Reihenfolge, sortiert oder nicht, mit oder ohne Doppelte)
- und welche Möglichkeiten (Methoden) es gibt, mit diesen Werten in der jeweiligen Struktur zu arbeiten.

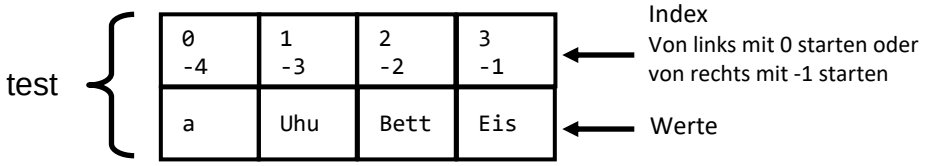
Abhängig davon, welche Werte benötigt werden und wie diese Werte verarbeitet werden, erfolgt also die Auswahl des Datentyps. Die folgende Tabelle gibt hierüber einen Überblick:

Datentyp	**Indiziert (und damit sortiert)**	**veränderbar**	**Doppelte erlaubt**
String	Ja	Nein	Ja
Liste	Ja	Ja	Ja
Tuple	Ja	Nein	Ja
Set	Nein	Nur Zufügen oder Löschen	Nein
Dictionary	Nein	Ja	Nein

9.2 Indizierung

Der Index gibt die Position im Speicher an. Verwendet wird eine fortlaufende Nummer.

Beispiel Liste mit 4 Elementen: test = ["a", "Uhu", "Bett", "Eis"]



Zugriff auf ein Element mit Hilfe eines Indexes	`print(test[0])  # a` `print(test[-4])  # a`

Zugriff auf einen Bereich von Werten mit [x:y]. Y nicht mehr enthalten!
 	
Fehlt x oder y reicht der Bereich vom Anfang bzw. bis zum Ende.
	`print(test[1:3])  # Uhu Bett`
	`print(test[-4:-2])  # a Uhu`
	
	`print(test[:3])  # a Uhu Bett`
	`print(test[2:])  # Bett Eis`

9.3 Datentypen (String, Tuple, List, Set und Dictionary)

Datentyp String: eine Zeichenkette. Folge von Zeichen, die indiziert und NICHT veränderbar sind.

Erzeugung eines Strings	`# mit einfachen oder doppelten Anführungszeichen`
	`str1 = "Hallo"`
	`str2 = 'Hallo'`

Ausgabe eines Strings	`print (str1)` `# Hallo`
Zugriff auf einzelnen Zeichen über Indizierung (siehe oben)	`print (str1[0])` `# H`
Zeichen können nicht überschrieben werden (Nicht veränderbar)	~~`str1[0] = "J"`~~ `# Geht so nicht!`

Einige Methoden zu Strings
len() - Länge ermitteln
strip() - Whitespaces an Enden entfernen
lower() - alle Buchstaben in Kleinbuchstaben
upper() - alle Buchstaben in Großbuchstaben
replace() – ersetzt String bzw. Zeichen
	`str3 = "_ _Hallo_Python_".` `# Hinweis: _ = Leerzeichen`
	`print(len(str3))` `# 15`
	`print(str3.strip())` `# Hallo_Python`
	`print(str3.lower())` `# hallo_python`
	`print(str3.upper())` `# HALLO_PYTHON`
	`print(str3.replace("H", "J")` `# JALLO_PYTJON`

in oder **not in** prüft auf Teil-String	`str4 = "Heute ist frei"`
	`x = "te" in str4`
	`print(x)` `#True`
	`x = "te" not in str4`
	`print(x)` `#False`

Datentyp Liste: Folge von einzelnen Werten, deren Werte indiziert und veränderbar sind (ähnlich einem Array aus anderen Programmiersprachen).

Erzeugen einer Liste (1)
Erzeugen einer Liste (2)
	`# Elemente in [ ]`
	`list1 = ["H", 7, True, "Eis"]`
	`list2 = list(("a", 8))` `# mit list Konstruktor`

Ausgabe einer Liste	`print (list1)` `# ['H', 7, True,'Eis']`
Zugriff auf einzelne Werte über Indizierung (siehe oben)	`# Beispiel` `print (list1[0])` `# H`
Werte können überschrieben werden (Veränderbar)	`list1[0] = "J"` `print (list1[0])` `# J`
Mit einer Schleife eine Liste durchlaufen	`for x in list1:` `  print(x, end=' ')` `# 'H' 7 True 'Eis'`
Listen verbinden	`liste3 = liste1 + liste2` `print (liste3, end=' ')` `# 'H'7 True 'Eis' 'a' 8`

Einige Methoden zu Listen
len() – Anzahl Listen-Elemente ermitteln
append() – Element anhängen
 	
insert () – Element an Index einfügen
 	
remove() – Element löschen
 	
pop() – Löscht Element an Index. Ohne Index-Angabe wird letztes Element gelöscht.
	`list4 = ["H", 7, True]`
	`print(len(list4))` `# 3`
	`list4.append("Ja")`
	`print (list4)` `# ['H', 7, True, 'Ja']`
	`list4.insert(0, 8)`
	`print(list4)` `# [8, 'H' 7, True, 'Ja']`
	`list4.remove(True)`
	`print(list4)` `# [88, 'H', 77,'Ja']`
	`list4.pop(2)`
	`print(list4)` `# [88, 'H', 'Ja']`

Datentyp Tuple: Folge von einzelnen Werten, deren Werte indiziert und NICHT veränderbar sind.

Erzeugen eines Tuples(1)
Erzeugen eines Tuples(2)
	`# Elemente in ( )`
	
	`tup1 = ("Katze", "Hund", "Maus")`
	`tup2 = tuple(('a', 'b')).  # mit Konstruktor`

Ausgabe eines Tupels	`print(tup1)  # ('Katze','Hund', Maus')`
Zugriff auf Listen-Element über Indizierung (siehe oben)	`print (tup1[2])  # Maus`
Werte können nicht überschrieben werden (Nicht veränderbar)	~~`Tup1[0] = "Löwe"`~~ `# Geht so nicht!`
Mit einer Schleife ein Tuple durchlaufen	`for x in tup1:` `  print(x)  # ('Katze','Hund','Maus')`
Tuples verbinden	`tup3 = tup1 + tup2` `print (tup3)  # ('Katze','Hund','Maus','a','b')`

Einige Methoden zu Tuplen
len() – Anzahl Tuple-Elemente ermitteln
count() – Zählt wie oft ein bestimmtes Element im Tuple ist
index () – Sucht ein angegebenes Element und gibt den Index zurück
	`print(len(tup3))  # 5`
	`print(tup3.count('a'))  # 1`
	
	`print (tup3.index('a'))  # 3`

Anmerkung:
Um ein Tuple zu ändern, kann es in eine Liste konvertiert werden und nach der gewünschten Änderung zurück in ein Tuple überführt werden.
	`x = (1, 2, 3)`
	`y = list(x)`
	`y[1] = 77`
	`x = tuple(y)`
	
	`print(x)  #(1, 77, 3)`

Datentyp Set: Menge von Werten, deren Werte nicht indiziert sind und die nicht veränderbar sind.

Erzeugen eines Sets(1)
Erzeugen eines Sets(2)
	`Elemente in { }`
	
	`set1 = {1, 2, 3}`
	`set2 = set(('a','b','c'))  # mit Konstruktor`

Ausgabe eines Sets	`print(set1)  # 1,2,3`
Zugriff auf Set-Elemente über Indizierung (siehe oben) nicht möglich	~~`print (set[2])`~~ `# Geht so nicht!`

Mit einer Schleife ein Set durchlaufen	`for x in set1:`
	`  print(x)  # {3,2,1}`
	`  Reihenfolge willkürlich`

Sets verbinden	`set3 = set1.union(set2)`
	`print (set3)  # {3,'b',1,'c','a',2}`
	`  Reihenfolge willkürlich`

Einige Methoden zu Sets
len() – Anzahl Set-Elemente ermitteln
remove() , **discard()** – Element löschen
	`print(len(set3))  # 6`
	`print(set3.remove('a'))  #{3, 'b', 1, 'c', 2}`
	`  Reihenfolge willkürlich`

Datentyp Dictionary: Folge von einzelnen Schlüsseln (keys) mit zugehörigen Werten (values). Beides und auch die Anzahl der Elemente kann verändert werden. Ähnlich einer Liste, außer dass der Index keine Zahl ist, sondern ein Name. Ein Dictionary entspricht einem assoziativen Array aus anderen Programmiersprachen.

Erzeugen eines Dictionarys (1)
Erzeugen eines Dictionarys (2)
	# Elemente in { }
	
	`dic1 = {`
	`"Name": "Meier"`
	`"Vorname": "Peter"`
	`"Jahr": 1988`
	`}`
	
	`dic2 = dict (Name="Meier",Gehalt=5000) # mit Konstruktor`

Ausgabe eines Dictionarys	`print (dic2) # {'Name':'Meier,'Gehalt':5000}`
Zugriff auf einzelne Werte über Indizierung (siehe oben), Index ist ein Name	`print (dic2["Name"]) # Meier`
Werte können überschrieben werden (Veränderbar)	`dic2["Name"] = "Schmidt"` `print (dic2["Name"]) # Schmidt`

Mit einer Schleife eine Liste durchlaufen	`for x in dic2:`
	`print(x) # Ausgabe der Indices: Name Gehalt`
	
	`for x in dic2: # Ausgabe der Werte: Schmidt 5000`
	`print(dic2[x])`

Einige Methoden zu Listen
len() – Anzahl Dict-Elemente ermitteln
pop() – Element löschen
Element anhängen
	`print(len(dic2)) # 2`
	`print(dic2.pop("Gehalt")) # {'Name':'Schmidt}`
	`dic2["vip"] = True`
	`print(dic2) # {'Name':'Schmidt,'vip':True}`