



# Grundlagen der Programmierung für Nicht-Informatiker

Tag 2

Giuseppe Accaputo

[g@accaputo.ch](mailto:g@accaputo.ch)



## Inhaltsverzeichnis – Gesamter Kurs

1. Grundlagen der Programmierung
2. Variablen, Anweisungen, Ausdrücke, und alles dazwischen
3. Funktionen Teil 1 – Ein Einstieg
4. Bedingte Anweisungen («Conditionals»)
5. Funktionen Teil 2 – Rückgabewerte, Wiederverwendbarkeit, und mehr
6. Iterationen – Werkzeuge für repetitive Aufgaben
7. Datenstrukturen – Listen, Strings, Tupel, und Dictionaries
8. ~~Dateien~~



## Inhaltsverzeichnis – Heutiger Kurstag

1. Grundlagen der Programmierung
2. Variablen, Anweisungen, Ausdrücke, und alles dazwischen
3. Funktionen Teil 1 – Ein Einstieg
4. Bedingte Anweisungen («Conditionals»)
5. Funktionen Teil 2 – Rückgabewerte, Wiederverwendbarkeit, und mehr
6. Iterationen – Werkzeuge für repetitive Aufgaben
7. Datenstrukturen – Listen, Strings, Tupel, und Dictionaries
8. ~~Dateien~~



## Changes im Vergleich zu Tag 1

- Mehr Live Coding (vorallem Beispiele)
- Mehr Aufgaben



# Neu: Python 2.7 Spickzettel für diesen Kurs

Python 2.7 Spickzettel	
Giuseppe Accaputo <a href="mailto:g@accaputo">g@accaputo</a>	
Kurs: Grundlagen der Programmierung für Nicht-Informatiker, FS18	
Variablen	
variablen_name = <wert>	
typ_der_variable = type(variable)	
Datentypen	
Integer (Int)	-25, 2, 14, 100, -20
Float	2.4123, -1.1312, 4.14123
String	"Hallo 1", 'Hallo 2'
Boolean	True, False
List	[1, 1.2423, "zwei"]
Tupel	(1, 2, 3, "vier")
Dictionary	{"key1": wert1, "key2": wert2}
Eingabe	
eingabe = raw_input('Bitte Name eingeben:')	
eingabe = int(raw_input('Bitte Zahl eingeben:'))	
eingabe = float(raw_input('Bitte Zahl eingeben:'))	
Ausgabe	
print 'Wert der Variable:', variable	
print 'Typ der Variable', type(variable)	
print 'Keine neue Zeile am Ende dieses Texts',	
Strings – Teil 1	
ein_string = "Hallo, Welt!"	Variable vom Typ String definieren
ein_string[1:5]	Segment von Index 1 bis und mit Index 4 selektieren
ein_string[-1]	Auf das letzte Zeichen zugreifen
ein_string.lower()	In Kleinbuchstaben umwandeln
ein_string.upper()	In Grossbuchstaben umwandeln
ein_string.replace(alt, neu)	alt durch neu in ein_string ersetzen
string1 == string2	Ist string1 gleich string2?

zahl_als_string = str(1.234)	Typumwandlung zu String
hallo = "Hallo, " welt = "Welt!" hallo_welt = hallo + welt	+ Operator: Zwei Strings verknüpfen
area = "Area " area_zahl = 51 area_51 = area + str(area_zahl)	+ Operator und str(): String und Zahl verknüpfen
Zahlen	
int(variable)	Typumwandlung zu Int
float(variable)	Typumwandlung zu Float
Mathematische Operatoren	
x ** y	Exponent, $x^y$
x % y	Modulus; berechnet den Rest der Division x geteilt durch y
x / y	Division
x * y	Multiplikation
x - y	Subtraktion
x + y	Addition
x op y, und x, y sind beide Ints	Resultat der Operation ist ein Int (op kann +, -, *, / sein)
x op y, und x oder y ist Float	Resultat der Operation ist ein Float (op kann +, -, *, / sein)
Funktionen	
def hallo(): print "Hallo!" hallo() # Aufruf	Eine Funktion ohne Rückgabewert
def hallo(name): print "Hallo, ", name hallo("Klasse") # Aufruf	Eine Funktion mit einem Argument
def summe(x, y, z): return x + y + z print summe(1,2,3) # Aufruf	Eine Funktion mit einem Rückgabewert
import math math.sqrt(zahl) # Wurzel math.log(zahl) # Logarithmus	Mathematische Funktionen verwenden



**Universität  
Zürich** <sup>UZH</sup>

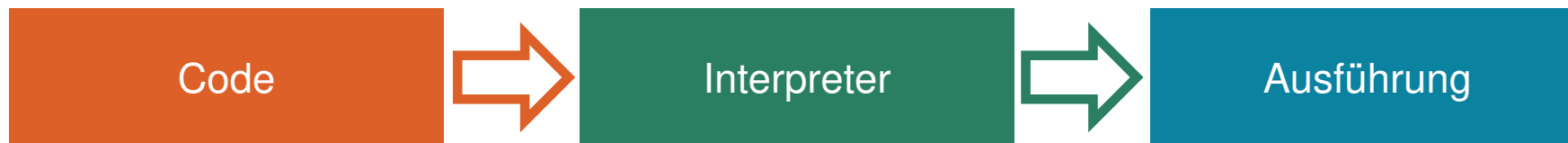
Zentrale Informatik – IT Fort- und Weiterbildungen

# Repetition Tag 1





## Dem Computer erfolgreich Anweisungen geben





## Variablen

- Eine *Variable* ist wie eine beschriftete Box, in welcher wir einen Wert verstauen (*speichern*) können
  - Eine Variable hat also einen *Namen* und einen *Wert*

```
answer = 42
mein_name = 'Giuseppe'

print answer
print mein_name
print type(mein_name)
```

CODE

INTERPRETER

```
42
Giuseppe
<type 'str'>
```

PYCHARM





## Aufgabe • Datentypen zuweisen

[Aufgabe]

- Für jede der folgenden Anweisungen, versucht zu bestimmen welcher Datentyp ausgegeben wird: Int, Float, String, oder Bool

```
print type(1 < 2)
print type(1.234)
print type(1/2.0)
print type("1.4123")
print type(1/4)
print type(1 + 2.0)
print type(0 == 0 and 1 < 2)
```

CODE



## Anweisungen und Ausdrücke

- Eine *Anweisung* ist eine Instruktion (oder Befehl), welche der Python-Interpreter ausführen kann
  - Beispiel: Die Wertzuweisung `sprache = 'Python'` ist eine Anweisung
- Ein *Ausdruck* ist eine Kombination von Werten, Variablen, und Operatoren
  - Ein Ausdruck kann immer ausgewertet werden

```
zahl1 = 4           # Anweisung  CODE
zahl2 = zahl1 + 3  # Anweisung
print zahl2        # Anweisung
3                  # Ausdruck
```

INTERPRETER

7

PYCHARM



## Operatoren

- *Operatoren* sind spezielle Symbole, die z.B. Berechnungen wie die Addition oder Multiplikation darstellen
- Werte, die durch Operatoren verknüpft werden, heissen *Operanden*
- Die Bedeutung eines Operators hängt vom Datentyp der Operanden ab
  - Z.B. Der + Operator angewendet auf zwei Zahlen addiert diese zusammen; der + Operator angewendet auf zwei Strings verkettet sie hingegen

```
print 3 + 4 * 10  
print 3600 / 60  
print 'Ein' + ' Beispiel'
```

CODE

INTERPRETER

```
43  
60  
Ein Beispiel
```

PYCHARM



## Operatoren und die Vorrangregeln

Operator	Operation
**	Exponent
%	Modulus
/	Division
*	Multiplikation
-	Subtraktion
+	Addition
<, <=, >, >=, !=, ==	Vergleichsoperatoren
not	Negation
and	UND Verknüpfung
or	ODER Verknüpfung



## Funktionen

- Eine Funktion ist wie ein Miniprogramm im Programm selbst

```
def hello(name):  
    print 'Hello,' + name + '!'  
  
hello('Giuseppe')
```

CODE

INTERPRETER

```
Hello, Giuseppe!
```

PYCHARM



## Aufgabe • Mehrere Funktionen

[Aufgabe]

- Gegeben sei folgender Code:

```
def gewinnende_zahl_anzeigen():  
    gewinnende_zahl = 4923  
    print gewinnende_zahl  
  
def gewinnende_zahl_ankuenden():  
    print 'Die gewinnende Zahl lautet:'  
    gewinnende_zahl_anzeigen()  
  
gewinnende_zahl_ankuenden()
```

CODE

- Was wird ausgegeben?



## Bedingte Anweisungen

- Wir möchten gewisse Sachen nur dann ausführen, wenn eine bestimmte Bedingung erfüllt ist

CODE

```
if x < y:  
    print x + ' ist kleiner als ' + y  
elif x > y:  
    print x + ' ist grösser als ' + y  
else:  
    print x + ' und ' + y + ' sind gleich gross'
```



## Aufgabe • Bedingte Anweisungen

[Aufgabe]

- Gegeben sei folgender Code:

```
x = 10
y = 20

if x > y or x%2 == 0:
    print "A"
elif x < y and y%2 == 0:
    print "B"
else
    print "C"
```

CODE

- Wird **A**, **B**, oder **C** ausgegeben?





## Aufgabe • Bedingte Anweisungen

[Aufgabe]

- Gegeben sei folgender Code:

```
a = True
b = (1 > 2)

if a and b:
    print "A"
elif a or b:
    print "B"
else:
    print "C"
```

CODE

- Wird **A**, **B**, oder **C** ausgegeben?



**Universität  
Zürich** <sup>UZH</sup>

**Zentrale Informatik – IT Fort- und Weiterbildungen**

# **Funktionen Teil 2 – Rückgabewerte, Wiederverwendbarkeit, und mehr**





## Lernziele

- Nach dieser Einheit wisst ihr...
  - ...wie wir aus einer Funktion aus einen Wert zurückgeben können
  - ...wie wir mit einem Rückgabewert weiterarbeiten können
  - ...mathematische Funktionen schreiben, die gewisse Berechnungen durchführen und Resultate liefern



## Rückgabewerte

- Funktionen können Resultate produzieren
- Ein Funktionsaufruf kann einen Wert erzeugen, welchen wir einer Variable zuweisen können oder als Teil eines Ausdrucks verwenden können

```
euler_konstante = math.exp(1.0)  
wurzel_vier = math.sqrt(4.0)
```

CODE

```
print euler_konstante  
print wurzel_vier
```

INTERPRETER

```
2.71828182846  
2.0
```

PYCHARM



## Rückgabewert

- Beispiel: Funktion, die Mittelwert aus vier Zahlen berechnen und *zurückgibt*

CODE

```
def mittelwert(w, x, y, z):  
    avg = (w + x + y + z) / 4.0  
    return avg
```

```
mittelwert_1 = mittelwert(1,1,1,1)  
mittelwert_2 = mittelwert(1,2,3,4)  
print 'mittelwert_1 = ', mittelwert_1  
print 'mittelwert_2 = ', mittelwert_2
```

```
neuer_mittelwert = (mittelwert_1 + mittelwert_2) / 2  
print 'neuer_mittelwert = ', neuer_mittelwert
```



## Die `return` Anweisung ohne Ausdruck – Funktionsausführung terminieren

- `return` Anweisung ermöglicht es, eine Funktion frühzeitig zu beenden
- Möglicher Grund: Eintreten einer Fehlerbedingung

CODE

```
def wurzel(x):  
    if x < 0:  
        print 'Nur positive Zahlen und die 0 sind erlaubt'  
        return  
  
    resultat = sqrt(x)  
    print 'wurzel(x) = ' + str(resultat)
```



## Die `return` Anweisung inklusive Ausdruck – Einen Wert zurückgeben

- `return` Anweisung inklusive Ausdruck: «Verlasse die Funktion sofort und verwende dabei den folgenden Ausdruck als Rückgabewert der Funktion»
  - *Reminder Nr. 1:* Ausdruck kann Kombination aus Werten, Variablen, und Operatoren sein
  - *Reminder Nr. 2:* Ein Ausdruck hat auch einen Typ, d.h. wir können Funktion definieren, die Floats, Strings, Ints, Booleans, etc. zurückgeben können

```
def mittelwert(w, x, y, z):  
    avg = (w + x + y + z) / 4.0  
    return avg
```

CODE



## LC 4.1 • Kreisfläche berechnen

{Live Coding}

- Wir implementieren eine Funktion `kreisflaeche`, die basierend auf einen Kreisradius  $r$  die dazugehörige Kreisfläche  $A(r)$  berechnet und zurückgibt:

$$A(r) = \pi r^2$$





## Die `return` Anweisung inklusive Ausdruck – Einen Wert zurückgeben

- Es ist auch möglich mehrere `return` Anweisungen in eine Funktion zu haben – z.B. bei Funktionen, welche bedingte Anweisungen haben
- Dabei ist es wichtig zu versichern, dass dabei jeder Pfad / Zweig ein Ergebnis zurückgibt

CODE

```
def abs(x):  
    if x < 0:  
        return -x  
    else:  
        return x
```



## LC 4.2 • Der Absolutwert

{Live Coding}

- Der Absolutwert einer reellen Zahl  $x$  ist gegeben als
  - $|x| = x$  falls  $x \geq 0$  oder
  - $|x| = -x$  falls  $x < 0$
- Eine Implementation des Absolutwerts ist in folgender Form gegeben:

CODE

```
def absolut_wert(x):  
    if x < 0:  
        return -x  
    if x > 0:  
        return x
```

- Was geschieht, wenn wir `absolut_wert(0)` aufrufen?



## Funktionen ohne Rückgabewerte

- Falls bis zum Ende der Funktion kein `return` mit Anweisung ausgeführt wird, so gibt die Funktion `None` «zurück»

```
def hello():  
    print 'Hello!'  
  
hello_var = hello()  
print hello_var  
print type(hello_var)
```

CODE

INTERPRETER

```
Hello!  
None  
<type 'NoneType'>
```

PYCHARM



## Aufgabe 4.1 • Zahlen vergleichen

[Aufgabe]

- Schreibe eine Funktion **vergleiche**, die folgende Werte zurückgibt:
  - **1**, falls  $x > y$
  - **0**, falls  $x == y$
  - **-1**, falls  $x < y$
- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
<code>vergleiche(3,1)</code>	1
<code>vergleiche(-1,4)</code>	-1
<code>vergleiche(42,42)</code>	0



## Boolesche Funktionen

- Wir können also auch Funktionen definieren, welche Boolesche Werte zurückgeben:

CODE

```
# Gibt zurück, ob x durch y (ohne Rest) teilbar ist
def ist_teilbar(x,y):
    if (x % y) == 0:
        return True
    else:
        return False
```



## Boolesche Funktionen

- Da das Resultat eines Vergleichs mittels `==` ein Boolescher Wert ist (z.B. evaluiert `3 == 3` zu `True`), so können wir die `ist_teilbar` Funktion ein bisschen prägnanter schreiben:

```
# Gibt zurück, ob x durch y teilbar ist  
def ist_teilbar(x,y):  
    return (x % y) == 0
```

CODE



## LC 4.3 • Sandwich-Funktion Teil 2

{Live Coding}

- Wir schreiben die Funktion `ist_sandwich` aus Aufgabe 3.3 so um, dass sie `True` zurückgibt, falls `x <= y <= z`, und `False` in allen anderen Fällen
  - **Frage:** `x <= y <= z` ist keine gültige Python Syntax; wie können wir diese Bedingung in Python ausdrücken? (Stichwort: logische Operatoren)
- Des Weiteren möchten wir, dass das Programm basierend auf dem Rückgabewert der `ist_sandwich` Funktion eine Meldung auf dem Bildschirm ausgibt, ob das Tripel `x`, `y`, und `z` ein Sandwich darstellt oder nicht



## Inkrementelle Programmentwicklung

- Immer wieder bisschen Code schreiben / hinzufügen, und dann gleich testen
- Verwende temporärere Variablen um Werte aus Zwischenschritten auszugeben und zu überprüfen
- Wenn Programm funktioniert, kann Code evtl. weiter vereinfacht werden
  - nur soweit vereinfachen, dass der Code doch noch lesbar bleibt





## LC 4.4 • Distanz zwischen zwei Punkte berechnen

{Live Coding}

- Wir schreiben die Funktion `distanz` welche die Distanz zwischen zwei Punkte  $(x_1, y_1)$  und  $(x_2, y_2)$  auf dem kartesischen Koordinatensystem berechnet
- *Wichtig*: Speichert das Programm unter `euklid.py` ab, da wir die Funktion später wiederverwenden werden
- Die Formel für die Distanz  $d$  ist gegeben durch

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

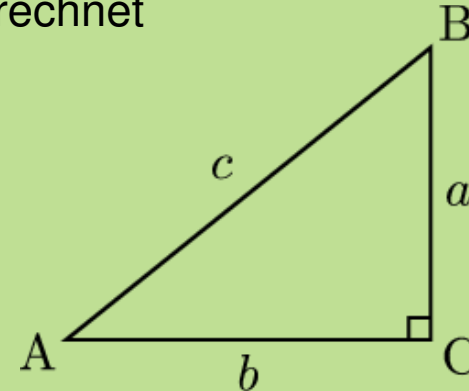
- Die Formel für  $d$  kann mittels Satz von Pythagoras hergeleitet werden

## Aufgabe 4.2 • Hypotenuse berechnen

[Aufgabe]

- Schreibe eine Funktion **hypotenuse**, welche abhängig von den Seitenlängen **a** und **b** der Katheten eines rechtwinkligen Dreiecks die Länge **c** der Hypotenuse berechnet
- Die Formel für die Länge **c** der Hypotenuse ist gegeben durch

$$c = \sqrt{a^2 + b^2}$$



- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
hypotenuse(0, 0)	0
hypotenuse(3, 4)	5
hypotenuse(-1, 2)	None (negative Längen sind nicht erlaubt)



## Wiederverwendung von Code mittels Funktionen

- Code Segmente, welche an verschiedenen Stellen des Programms in gleicher Form vorkommen / benötigt werden können wir in eine Funktion packen
- Funktionen können innerhalb von anderen Funktionen aufgerufen werden
- Wiederverwendung von Code durch Funktionen macht Code übersichtlicher und erlaubt es leichter Änderungen vorzunehmen
  - Bei Anpassungen müssen wir nur den Code der Funktion anpassen statt jede Stelle im Programm einzeln
  - Wir vermeiden unnötige Duplikation von Code



## LC 4.5 • Ein einfacher Taschenrechner

{Live Coding}

- In einem ersten Schritt implementieren wir einen simplen Taschenrechner, der dem Benutzer erlaubt zwei Zahlen einzugeben und danach die Summe daraus ausgibt
- In einem zweiten Schritt erweitern wir den Taschenrechner so, dass der Benutzer nun zusätzlich zu den zwei Zahlen auch die gewünschte mathematische Operation angeben kann. Dabei soll unser Taschenrechner die Addition, Subtraktion, Multiplikation, sowie Division zweier Zahlen unterstützen.



## Generalisierung von Funktionen

- Wir können auch mehrere Funktionen mit dem gleichen Namen zur Verfügung stellen in einem Programm
  - Die Funktionen müssen sich in der Anzahl der Parameter unterscheiden

```
def summe(x,y):  
    return x + y
```

```
def summe(x,y,z):  
    return summe(x,y) + z
```

CODE



## Gültigkeitsbereich von Variablen

- Variable innerhalb einer Funktion ist nur lokal gültig (kann ausserhalb der Funktion nicht verwendet werden)
  - Parameter sind Funktions-lokale Variablen

```
# Globale Variable CODE
nachname = 'Accaputo'

def hello(vorname):
    # vorname ist eine lokale Variable
    print 'Hello,' + vorname +
          ' ' + nachname + '!'

hello('Giuseppe')
print vorname # nicht möglich!
```

INTERPRETER

```
Hello, Giuseppe Accaputo!
```

PYCHARM



## Lernziele – Check (mit Farben)

- Nach dieser Einheit wisst ihr...
  - ...wie wir aus einer Funktion aus einen Wert zurückgeben können
  - ...wie wir mit einem Rückgabewert weiterarbeiten können
  - ...mathematische Funktionen schreiben, die gewisse Berechnungen durchführen und Resultate liefern



## Aufgabe 4.3 • Kugelvolumen berechnen

[Aufgabe]

- Schreibe eine Funktion **kugel\_volumen**, die anhand der Punkte  $(x_1, y_1)$  und  $(x_2, y_2)$  das Volumen  $V(r)$  einer Kugel mit Zentrum im Punkt  $(x_1, y_1)$  berechnet. Der zweite Punkt  $(x_2, y_2)$  befindet sich auf der Oberfläche der Kugel, d.h. der Radius  $r$  der Kugel entspricht der Distanz zwischen den beiden Punkten.

- **Tipp:** Verwende die Funktion **distanz** aus LC 4.4

- Die Formel für das Volumen  $V$  lautet

$$V(r) = \frac{4}{3} \cdot \pi \cdot r^3$$

- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
<code>kugel_volumen(0,0,0,1)</code> (Einheitskugel)	4.18879020479 = $\frac{4}{3} \cdot \pi$
<code>kugel_volumen(1,2,3,4)</code>	94.7815026807





**Universität  
Zürich** <sup>UZH</sup>

Zentrale Informatik – IT Fort- und Weiterbildungen

# Iterationen – Werkzeuge für repetitive Aufgaben





## Lernziele

- Nach dieser Einheit wisst ihr...
  - ...was für einen Einfluss mehrere Anweisungen auf dieselbe Variable haben
  - ...wie man repetitive Aufgaben mittels **while**-Schleifen implementieren kann
  - ...warum es wichtig ist, die Schleifen-Bedingung nach einigen Schleifendurchgängen nicht mehr zu erfüllen (Stichwort *unendliche Schleifen*)



## Mehrere Zuweisungen

- Derselben Variable können wir mehrere Werte zuweisen

```
name = 'Giuseppe'  
print name  
name = 'Marco'  
print name
```

CODE

INTERPRETER

```
Giuseppe  
Marco
```

PYCHARM

## Variablen aktualisieren

- Eine oft-vorkommende Form von vermehrten Zuweisungen ist die *Aktualisierung einer Variable*, wobei der neue Wert der Variable vom alten Wert abhängt
  - Bevor man eine Variable aktualisiert, muss man sie initialisieren

```
x = 0
x = x + 1
x = x + 1
print x
```

CODE

INTERPRETER

2

PYCHARM

- Variable um 1 erhöhen: *Inkrement*
- Variable um 1 verringern: *Dekrement*



## Die `while` Schleife

- Repetitive Aufgaben (wie z.B. das Inkrementieren einer Variable) können wir mittels `while`-Schleife ausführen

```
def countdown(n):  
    while n > 0:  
        print n  
        n = n - 1  
  
    print 'LOS!'
```

CODE

- Das obige Programm kann man wie folgt *lesen*: "Solange `n` grösser als `0` ist, gib den Wert von `n` aus und dekrementiere `n` um `1`. Wenn `n` den Wert `0` hat, gib **LOS!** aus"



## Die `while` Schleife – Ablauf

1. Evaluiere Bedingung, die entweder **True** or **False** ergibt
2. Falls die Bedingung nicht erfüllt ist – also **False** ergibt –, verlasse die **while**-Schleife und führe die nächste Anweisung aus
3. Falls die Bedingung erfüllt ist – also **True** ergibt –, führe den Block in der Schleife aus und gehe zu Schritt 1

```
def countdown(n):  
    while n > 0: # Schleifen-Bedingung  
        print n    # Schleifen-Block  
        n = n - 1  # Schleifen-Block  
  
    print 'LOS!' # Erste Anweisung nach while-Schleife
```



## «Bis zur Unendlichkeit und noch viel weiter»

- Im Schleifen-Block sollte sichergestellt werden, dass die Schleifen-Bedingung innerhalb einer endlichen Anzahl Iterationen ungültig wird (also zu **False** evaluiert), sodass die Schleife abgebrochen wird

CODE

```
def countdown(n):  
    while n > 0:  
        print n  
  
    print 'LOS!'
```



## Aus einer Schleife ausbrechen – Die `break` Anweisung

- Mit der `break` Anweisung können wir aus einer `while`-Schleife ausbrechen – unabhängig vom Zustand der Schleifen-Bedingung

CODE

```
while True:
    eingabe = raw_input('Gib bitte das Zauberwort ein: ')

    if eingabe == 'Bitte':
        break

    print 'Falsche Eingabe...'

print 'Ende!'
```





## Aufgabe 5.1 • Login Abfrage (Gruppenarbeit)

[Aufgabe]

- Schreibe ein Programm, das dem Benutzer erlaubt sich mittels Benutzername und Passwort für ein Service einzuloggen
- Dabei hat der Benutzer maximal 3 Versuche um das Login richtig einzugeben, danach wird sein Account gesperrt
  - Der Benutzer soll über eine erfolgte Sperrung via Meldung am Bildschirm informiert werden
- Die Login Daten lauten:
  - Benutzername: **user**
  - Passwort: **mypass**
- **Tipp:** Wir brauchen sicher mal eine Variable, um die Anzahl Versuche zu zählen. Wie könnte dann die Schleifen-Bedingung aussehen?

Benutzername: user

BILDSCHIRM AUSGABE

Passwort: mypas

Falsches Login (Fehlversuch 1 von 3)

Benutzername: user

Passwort: mypass

Login erfolgreich!



## Lernziele – Check (mit Farben)

- Nach dieser Einheit wisst ihr...
  - ...was für einen Einfluss mehrere Anweisungen auf dieselbe Variable haben
  - ...wie man repetitive Aufgaben mittels **while**-Schleifen implementieren kann
  - ...warum es wichtig ist, die Schleifen-Bedingung nach einigen Schleifendurchgängen nicht mehr zu erfüllen (Stichwort *unendliche Schleifen*)



**Universität  
Zürich** <sup>UZH</sup>

Zentrale Informatik – IT Fort- und Weiterbildungen

# Datenstrukturen





## Lernziele

- Nach dieser Einheit wisst ihr...
  - ...was der Unterschied zwischen einer *veränderbaren* und *unveränderbaren* Datenstruktur ist
  - ...was ein Index ist und wie man damit auf einzelne Elemente der Datenstrukturen zugreifen kann
  - ...wie man durch die Elemente der einzelnen Datenstrukturen durchiterieren kann



**Universität  
Zürich** <sup>UZH</sup>

Zentrale Informatik – IT Fort- und Weiterbildungen

# Datenstrukturen Teil 1 – Mit Strings Zeichenketten darstellen



## Strings – Eine Sequenz von Zeichen

- Eine String ist eine Sequenz von Zeichen
- Wir können auf jedes Zeichen eines Strings zugreifen:

```
name = 'Marco'  
  
erster_buchstabe = name[0]  
zweiter_buchstabe = name[1]  
  
print erster_buchstabe  
print zweiter_buchstabe
```

CODE

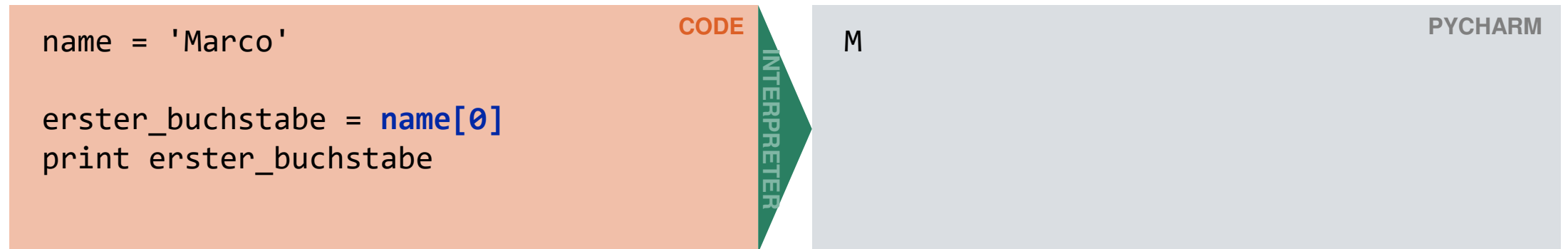
INTERPRETER

M  
a

PYCHARM

## Strings – Auf einzelne Zeichen zugreifen

- Die Zahl im Ausdruck `name[0]` nennt man einen *Index*
- Der Index zeigt an, auf welches Element in der Sequenz wir zugreifen möchten
- **Wichtig:** Der Index des ersten Elementes einer Sequenz hat in Python den Wert 0 (und nicht 1)



- Der Typ des Index muss immer Integer sein
  - `name[1.5]` ist daher kein gültiger Ausdruck



## Strings – Auf einzelne Zeichen zugreifen

Index:	0	1	2	3	4
Zeichen:	M	a	r	c	o





## Strings – Auf einzelne Zeichen zugreifen

Index:	0	1	2	3	4	5
Zeichen:	P	y	t	h	o	n



## Länge eines Strings

- Mit der Funktion `len(variable)` kann man die Anzahl Zeichen in einem String ermitteln:

```
name = 'Giuseppe'  
print len(name)
```

CODE

INTERPRETER

8

PYCHARM



## Letztes Zeichen in einem String

- Versucht bitte folgenden Code auszuführen:

```
sprache = 'Python'  
index_lz = len(sprache)  
letztes_zeichen = sprache[index_lz]
```

CODE



## Letztes Zeichen in einem String

- Weil wir bei Index 0 beginnen zu zählen, befindet sich das letzte Zeichen des Strings auf der Position `len(sprache) - 1`

```
sprache = 'Python'  
index_lz = len(sprache) - 1  
letztes_zeichen = sprache[index_lz]  
  
print letztes_zeichen
```

CODE

INTERPRETER

n

PYCHARM



## Letztes Zeichen in einem String – Negative Indizes

- Wir können auch negative Indizes verwenden. Dabei liefert `sprache[-1]` das letzte Zeichen, `sprache[-2]` das zweitletzte Element, etc.

```
sprache = 'Python'  
letztes_zeichen = sprache[-1]  
  
print letztes_zeichen
```

CODE

INTERPRETER

n

PYCHARM



## Einen String durchqueren – Wir besuchen alle Zeichen einzeln

- Wir möchten jedes Zeichen eines Strings auf einer neuen Zeile ausgeben:

CODE

```
sprache = 'Python'  
  
index = 0  
anz_zeichen = len(sprache)  
  
while index < anz_zeichen:  
    zeichen = sprache[index]  
    print zeichen  
    index = index + 1
```



## Einen String traversieren ("durchqueren")

- Wir können viel eleganter mit einer **for** Schleife jedes Zeichen in einem String traversieren:

```
sprache = 'Python'
```

```
for zeichen in sprache:  
    print zeichen
```

CODE



## String Segmentierung

- Wir können auf bestimmte Segmente in einem String zugreifen
- Der Zugriff geschieht mittels `string[0:5]`, wobei in diesem Fall die Zeichen `string[0]`, `string[1]`, ..., `string[4]` selektiert werden

CODE

```
name = 'Barack Obama'  
  
vorname = name[0:6]  
nachname1 = name[7:12]  
nachname2 = name[7:] # Segment von Index 7 aus bis ans Ende des Strings  
  
print vorname  
print nachname1  
print nachname2
```





## Strings sind unveränderbar

- Wir können einen existierenden String nicht verändern (z.B. in dem wir ein bestimmtes Zeichen ersetzen):

```
sprache = 'Python'  
# Ändere den ersten Buchstabe  
sprache[0] = 'J'
```

CODE

INTERPRETER

```
Traceback (most recent call last):  
  File "...", line 3, in <module>  
    sprache[0] = 'J'  
TypeError: 'str' object does not  
    support item assignment
```

PYCHARM



## Strings sind unveränderbar

- Wir können jedoch einen neuen String erstellen, der eine Variation des originalen Strings ist:

```
sprache = 'Python'  
# Ändere den ersten Buchstabe  
neue_sprache = 'J' + sprache[1:]  
print neue_sprache
```

CODE

INTERPRETER

Jython

PYCHARM



## LC 5.1 • Zeichensuche

{Live Coding}

- Wir schreiben eine Funktion `suche(zeichen, wort)`, welches den Index des ersten Vorkommnis von `zeichen` in `wort` zurückgibt, falls `zeichen` in `wort` vorkommt, und `None` falls nicht
- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
<code>suche("t", "Python")</code>	2
<code>suche("J", "Python")</code>	None



## Aufgabe 6.1 • Vorkommnisse zählen

[Aufgabe]

- Schreibe eine Funktion `anz_vorkommnisse(zeichen, wort)`, welche zurückgibt, wie oft `zeichen` in `wort` vorkommt (auf Grosskleinschreibung achten!)
- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
<code>anz_vorkommnisse("a", "Halleluja")</code>	2
<code>anz_vorkommnisse("e", "Mount Everest")</code>	2
<code>anz_vorkommnisse("k", "Kathedrale")</code>	0



## Kommt den eine Zeichenfolge überhaupt vor? – Der `in` Operator

- Mit dem `in` Operator können wir herausfinden, ob eine bestimmte Zeichenfolge (oder auch nur ein einzelnes Zeichen) in einem String vorkommt
  - Solch eine Zeichenfolge nennt man auch *Substring*

```
name = 'Python'  
  
if 'yth' in name:  
    print '"yth" kommt vor!'
```

CODE

INTERPRETER

```
"yth" kommt vor!
```

PYCHARM



## Strings vergleichen

- Um zu schauen ob zwei Strings gleich sind kann man den `==` Operator verwenden:

```
name = 'Python'  
  
if name == 'Python':  
    print 'Beide Strings sind gleich!'
```

CODE

INTERPRETER

```
Beide Strings sind gleich!
```

PYCHARM



## Strings vergleichen

- Wir können die Vergleichs-Operatoren `<` und `>` dazu verwenden, um die *lexikografische Reihenfolge* zwischen zwei Strings zu überprüfen:

```
name1 = 'Python'  
name2 = 'Jython'  
  
if name1 < name2:  
    print name1, 'kommt vor', name2  
else:  
    print name1, 'kommt nach', name2
```

CODE

INTERPRETER

```
Python kommt nach Jython
```

PYCHARM



**Universität  
Zürich** <sup>UZH</sup>

Zentrale Informatik – IT Fort- und Weiterbildungen

# Datenstrukturen Teil 2 – Listen





## Listen – Eine Sequenz von beliebigen Werten

- In einem String sind die einzelnen Elemente lediglich Zeichen
- In einer Liste können die einzelnen Werte jedoch von einem beliebigen Typ sein

```
liste_von_strings = ["eins", "zwei", "drei"]  
liste_von_ints = [1,2,3,4,5,6]
```

CODE

CODE

## Listen – Eine Sequenz von beliebigen Werten

- Eine Liste ist auch wieder ein Datentyp:

```
liste_von_strings = ["ab", "cd"]
liste_von_ints = [1,2,3,4,5,6]
gemischte_liste = [1,2,"drei"]

print type(liste_von_strings)
print type(liste_von_ints)
print type(gemischte_liste)
```

CODE

INTERPRETER

```
<type 'list'>
<type 'list'>
<type 'list'>
```

PYCHARM

- Wir kennen nun bereits 5 Datentypen: *Ints*, *Floats*, *Strings*, *Bools*, und *Lists*



## Listen – Eine Sequenz von beliebigen Werten

- In einer Liste dürfen die Elemente von verschiedenen Typen sein:

```
liste_gemischt = ["eins", 2, 3, "vier"] # Liste enthält Ints und Strings
```

CODE

- Wir können auch eine leere Liste definieren:

```
leere_liste = []
```

CODE

## Listen – Auf einzelne Elemente zugreifen

- Wir können wieder mittels eines Index auf ein einzelnes Element der Liste zugreifen:
- **Wichtig:** Der Index des ersten Elementes einer Sequenz hat in Python den Wert 0 (und nicht 1)

```
liste = ["eins", 2, 3.0, "vier"]
erstes_element = liste[0]
drittes_element = liste[2]

print erstes_element
print drittes_element
```

CODE

INTERPRETER

```
eins
3.0
```

PYCHARM

- Der Typ des Index muss immer Integer sein
  - `name[1.5]` ist daher kein gültiger Ausdruck



## Listen – Auf einzelne Elemente zugreifen

Index:	0	1	2	3
Elemente:	"eins"	2	3.0	"vier"



## Listen – Auf einzelne Elemente zugreifen

Index:	0	1	2	3
Elemente:	"eins"	2	3.0	"vier"
Typ:	str	int	float	str



## Listen – Eine Sequenz von beliebigen Werten

- In einer Liste dürfen die Elemente von verschiedenen Typen sein:

```
liste_gemischt = ["eins", 2, 3, "vier"] # Liste enthält Ints und Strings
```

CODE

```
print type(liste_gemischt[0])  
print type(liste_gemischt[1])  
print type(liste_gemischt[2])  
print type(liste_gemischt[3])
```



## Listen sind veränderbar

- Im Gegensatz zu Strings sind Listen veränderbar:

```
liste = ["eins", 2, 3, "vier"]  
print "alt:", liste  
  
# Ändere den Wert des ersten Elements  
liste[0] = 1  
  
print "neu:", liste
```

CODE

INTERPRETER

```
alt: ["eins", 2, 3, "vier"]  
neu: [1, 2, 3, "vier"]
```

PYCHARM





## Eine Liste durchqueren – Wir besuchen alle Elemente einzeln

- Wir können viel eleganter mit einer **for** Schleife jedes Element in einer Liste traversieren:

```
liste_gemischt = ["eins", 2, 3, "vier"] # Liste enthält Ints und Strings
```

CODE

```
for element in liste_gemischt:  
    print type(element)
```



## Länge einer Liste

- Mit der Funktion `len(liste)` kann man die Anzahl Elemente in einer Liste ermitteln:

```
liste = [1,2,3,4,"fuenf"]  
  
print len(liste)
```

CODE

INTERPRETER

5

PYCHARM



## Eine Liste durchqueren – Wir besuchen alle Elemente einzeln

- Die for Schleife aus dem vorherigen Slide erlaubt es, die einzelne Elemente zu lesen
- Wollen wir stattdessen bspw. jedes Element verändern, so können wir die **range** Funktion verwenden und mittels Index direkt auf das Element zugreifen::

```
zahlen = [1,2,3,4] # Liste enthält Ints und Strings
anz_zahlen = len(zahlen)

for index in range(anz_zahlen):
    zahlen[index] = zahlen[index] * 2 # Verdopple jedes Element

print zahlen
```

CODE



## Die range Funktion

- `range(n)` gibt eine Liste von Indizes von `0` bis `n-1`, also `[0,1,2,3, ...,n-1]`
- `range(a,b)` gibt eine Liste von Indizes von `a` bis `b` (`a<b`), also `[a,a+1,...,b-1]`

```
range1 = range(5)
range2 = range(3,8)
```

```
print range1
print type(range1)
print range2
print type(range2)
```

CODE

INTERPRETER

```
[0, 1, 2, 3, 4]
<type 'list'>
[3, 4, 5, 6, 7]
<type 'list'>
```

PYCHARM



## Ist ein bestimmtes Element in der Liste vorhanden? – Der `in` Operator

- Mit dem `in` Operator können wir herausfinden, ob ein bestimmtes Element in einer Liste vorhanden ist

```
liste = [1,2,3,4,"fuenf"]  
  
if 'fuenf' in liste:  
    print '"fuenf" kommt vor!'
```

CODE

INTERPRETER

```
"yth" kommt vor!
```

PYCHARM



## Elemente aus Liste entfernen

- Es gibt mehrere Möglichkeiten um ein Element aus einer Liste zu löschen:

CODE

```
liste = [1,2,"drei","vier",5,6,"sieben",8,9.0]
```

```
liste.pop(2) # Entfernt Element beim Index 2, also den String "drei"  
print liste # [1,2,"vier",5,6,"sieben",8,9.0]
```

```
liste.remove("vier") # Entfernt "vier"  
print liste # [1,2,5,6,"sieben",8,9.0]
```

```
del liste[0] # Entfernt Element beim Index 0, also die Zahl 1  
print liste # [2,5,6,"sieben",8,9.0]
```

```
del liste[0:3] # Entfernt die ersten drei Elemente, also die Zahlen 1, 2, und 5  
print liste # ["sieben",8,9.0]
```



## Listen als Argumente einer Funktion

- **Achtung:** Wenn eine Liste als Argument einer Funktion übergeben wird, so kann die Liste von der Funktion verändert werden, sodass die Änderung auch "ausserhalb" sichtbar wird

```
def ersetze_element(liste):  
    liste[0] = "neu"  
  
liste = [1,2,3]  
ersetze_element(liste)  
  
print liste # ["neu", 2, 3]
```

CODE



## Listen verketteten / zusammenknüpfen / verbinden

- Mit dem **+** Operator können zwei Listen verkettet werden:

```
zahlen1 = [1,2,3,4]
```

```
zahlen2 = [5,6,7,8]
```

```
zahlen_gesamt = zahlen1 + zahlen2
```

```
print zahlen_gesamt
```

CODE





## Eine Liste sortieren

```
liste = [5,1,3,2,4]
```

```
liste.sort()
```

```
print liste
```

CODE

INTERPRETER

```
[1, 2, 3, 4, 5]
```

PYCHARM



## Ein Element ans Ende einer Liste anfügen

```
liste = [1,2,3]  
  
liste.append("vier")  
  
print liste
```

CODE

INTERPRETER

```
[1, 2, 3, "vier"]
```

PYCHARM



## LC 5.2 • Beliebig grosse Liste

{Live Coding}

- Wir schreiben ein Programm, mit welchem der Benutzer mittels Eingabe eine beliebige Anzahl Elemente in eine Liste schreiben kann.
- Sobald **exit** eingegeben wird, wird das Programm beendet und die Listen Elemente werden vorher noch ausgegeben (inkl. die Anzahl Elemente in der Liste)



## Aufgabe 6.2 • Mitte

[Aufgabe]

- Schreibt eine Funktion **mitte**, welche eine neue Liste zurückgibt, die alle Elemente enthält, ausser dem ersten und letzten Element
- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
<code>mitte([1,2,3,4])</code>	<code>[2,3]</code>
<code>mitte([1,3,2])</code>	<code>[3]</code>
<code>mitte([3,2,5,20,5,100])</code>	<code>[2,5,20,5]</code>



## Aufgabe 6.3 • Durchschnitt berechnen

[Aufgabe]

- Schreibt eine Funktion **durchschnitt**, welche den Durchschnitt aus allen Elementen in einer Liste berechnet und zurückgibt (Bedingung: Liste darf nur Zahlen enthalten)
- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
<code>durchschnitt([1,2,3,4])</code>	2.5
<code>durchschnitt([4,18,30,-20])</code>	8
<code>durchschnitt([3,3,3,3])</code>	3



## Aufgabe 6.4 • Grösstes und kleinstes Element finden

[Aufgabe]

- Schreibt eine Funktion `min_max`, welche eine Liste bestehend aus zwei Elementen zurückgibt, wobei das erste Element das kleinste Element in der Liste, und das zweite Element das grösste Element in der Liste ist
  - **Tipp:** `min(liste)` gibt das kleinste Element in `liste` zurück, `max(liste)` jeweils das grösste Element
- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
<code>min_max([102, -2, 30, 400])</code>	<code>[-2, 400]</code>
<code>min_max([-123, 430, 5000, -300])</code>	<code>[-300, 5000]</code>



**Universität  
Zürich** <sup>UZH</sup>

Zentrale Informatik – IT Fort- und Weiterbildungen

# Datenstrukturen Teil 3 – Tupel



## Tupel sind unveränderbar

- Wie eine Liste, ist auch ein Tupel eine Sequenz von Werten mit verschiedenen Typen
- Auf die einzelnen Werte können wir auch mit einem ganzzahligen Index zugreifen
- **Wichtiger Unterschied:** Im Vergleich zu Listen sind Tupel *unveränderbar*

```
tupel1 = (1,2,3,4,"fuenf")  
tupel2 = "eins","zwei",3  
tupel3 = ("test") # Achtung!
```

CODE

INTERPRETER

```
type(tupel1)  
type(tupel2)  
type(tupel3)
```

```
<type 'tuple'>  
<type 'tuple'>  
<type 'str'>
```

PYCHARM

- Wir kennen nun bereits 6 Datentypen: *Ints*, *Floats*, *Strings*, *Bools*, *Lists*, und *Tuples*





## Tupel sind unveränderbar

```
t = ('a', 'b', 'c')
```

```
print t[0]  
print t[1:3]
```

```
t[0] = 'A' # Nicht möglich!
```

CODE

INTERPRETER

```
a  
('b', 'c')
```

```
TypeError: 'tuple' object does not  
support item assignment
```

PYCHARM



## Tupel Zuweisung

- Wir können einem Tupel von Variablen ein Tupel von Werten zuweisen:

```
a, b, c = 1, 2, 3  
print a  
print b  
print c
```

CODE

## Tupel als Rückgabewerte

- Strikt gesprochen kann eine Funktion nur einen Wert zurückgeben
- Ist der Rückgabewert jedoch ein Tupel, so ist der Effekt derselbe, als würde man mehrere Werte zurückgeben

```
def summe_prod(a, b):  
    return a+b, a*b  
  
s, p = summe_prod(10,20)  
  
print 's = ', s  
print 'p = ', p
```

CODE

INTERPRETER

```
s = 30  
p = 200
```

PYCHARM



## LC 5.3 • Division mit Rest

{Live Coding}

- Wir schreiben eine Funktion `div_mit_rest(a,b)`, welches als ersten Wert den Quotient zwischen **a** und **b**, und als zweiten Wert den Rest zurückgibt
- Beispiel: Bei der Division **10 / 3** beträgt das Resultat **3 Rest 1**, also beträgt der Quotient **3** und der Rest **1**
- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
<code>div_mit_rest(10, 3)</code>	(3, 1)
<code>div_mit_rest(12, 4)</code>	(3, 0)
<code>div_mit_rest(28, 5)</code>	(5, 3)



**Universität  
Zürich** <sup>UZH</sup>

Zentrale Informatik – IT Fort- und Weiterbildungen

# Datenstrukturen Teil 4 – Dictionaries



## Dictionaries – Die Wörterbücher der Informatik

- Ein *Dictionary* ist eine Datenstruktur, die Schlüssel-Werte Paare (die *Elemente*) enthält
- Mit dem Schlüssel können wir die enthaltenen Elemente adressieren und dabei auf den Wert zugreifen
- Der Schlüssel kann dabei von einem beliebigem Datentyp sein
  - Strings, Listen, und Tupel verwenden ganzzahlige Indizes
  - Ein Dictionary kann z.B. einen String als Datentyp für den Schlüssel verwenden
- Wir kennen nun bereits 7 Datentypen: *Ints*, *Floats*, *Strings*, *Bools*, *Lists*, *Tuples*, und *Dictionaries*



## Ein Dictionary definieren und verwenden

- Syntax um ein Dictionary zu definieren:

```
variable_name = {schluessel1: wert1, schluessel2: wert2, ...}
```

```
kontakte = {  
    "Giuseppe": "g@accaputo.ch",  
    "Peter M.": "peter@muster.ch",  
    "Peter B.": "peter@bernasconi.ch"  
}
```

CODE

```
giuseppes_email = kontakte["Giuseppe"]  
peter_ms_email = kontakte["Peter M."]
```

```
print giuseppes_email  
print peter_ms_email
```



## Ein leeres Dictionary erstellen

- Wir können mittels `{}` ein leeres Dictionary erstellen:

```
fr_woerterbuch = {}
```

```
fr_woerterbuch["programmieren"] = "programmer"
```

```
fr_woerterbuch["woerterbuch"] = "dictionnaire"
```

```
fr_woerterbuch["lernen"] = "apprendre"
```

```
print '"Lernen" auf französisch: ', fr_woerterbuch["lernen"]
```

CODE





## Ein Dictionary traversieren ("durchqueren")

- Mit einer **for**-Schleife und der Dictionary-Funktion **items()** können wir durch ein Dictionary durchiterieren:

CODE

```
preisliste = {  
    "Burger": 20,  
    "Chicken Nuggets": 15,  
}  
  
print "Unsere Preisliste:"  
  
for (name, preis) in preisliste.items():  
    print name, ":", preis
```



## Auf Dictionary Einträge zugreifen

- Mit `dictionary[key]` können wir auf den Wert des Eintrags mit dem Schlüssel `key` im `dictionary` zugreifen:

CODE

```
kontakte = {  
    "Giuseppe": "g@accaputo.ch",  
    "Peter M.": "peter@muster.ch",  
    "Peter B.": "peter@bernasconi.ch"  
}  
  
giuseppes_email = kontakte["Giuseppe"]  
peter_ms_email = kontakte["Peter M."]  
  
print giuseppes_email  
print peter_ms_email
```



## Wert eines Dictionary Eintrages ändern

- In einem Dictionary werden immer Schlüssel-Werte Paare gespeichert
- Wir können mit `dictionary[key] = value` den Werte des Eintrags mit dem Schlüssel `key` im `dictionary` auf `value` setzen
  - Falls ein Eintrag mit dem Schlüssel `key` nicht existiert, so wird dieser gleich zum Dictionary hinzugefügt

CODE

```
kontakte = {  
    "Giuseppe": "g@accaputo.ch",  
    "Peter M.": "peter@muster.ch",  
}  
  
kontakte["Peter M."] = peter@muster2.ch  
kontakte["Jens L."] = jens@luchs.ch  
  
print kontakte
```



## Dictionary Eintrag löschen

- Elemente kann man wie üblich mittels **del** aus einem Dictionary löschen:

CODE

```
kontakte = {  
    "Giuseppe": "g@accaputo.ch",  
    "Peter M.": "peter@muster.ch",  
}
```

```
del kontakte["Peter M."]
```

```
print kontakte
```



## Anzahl Dictionary Einträge

- Die Anzahl Elemente in einem Dictionary findet man mit Hilfe von `len`:

```
kontakte = {  
    "Giuseppe": "g@accaputo.ch",  
    "Peter M.": "peter@muster.ch",  
}  
  
anz_kontakte = len(kontakte)  
  
print anz_kontakte
```

CODE

INTERPRETER

2

PYCHARM



## Einen Schlüssel im Dictionary finden

- Mit `in` können wir herausfinden, ob ein Eintrag mit einem bestimmten Schlüssel im Dictionary befindet:

CODE

```
preisliste = {  
    "Burger": 20,  
    "Chicken Nuggets": 15,  
}  
  
if "Burger" in preisliste:  
    print "Ein Burger kostet", preisliste["Burger"]  
else:  
    print "Leider haben wir keine Burger im Sortiment"
```



## Lernziele – Check (mit Farben)

- Nach dieser Einheit wisst ihr...
  - ...was der Unterschied zwischen einer *veränderbaren* und *unveränderbaren* Datenstruktur ist
  - ...was ein Index ist und wie man damit auf einzelne Elemente der Datenstrukturen zugreifen kann
  - ...wie man durch die Elemente der einzelnen Datenstrukturen durchiterieren kann



## Aufgabe 6.5 • Die Wissensdatenbank

[Aufgabe]

- In den nächsten Aufgaben programmiert ihr eine Wissensdatenbank, welche Stichworte und dazugehörige Beschreibungen enthalten kann
- Folgende Funktionalität soll das Programm anbieten:
  - Hinzufügen eines Stichwortes und dessen Beschreibung zur Wissensdatenbank
  - Löschen eines Stichwortes (inkl. Beschreibung) aus der Wissensdatenbank
  - Nach der Beschreibung eines bestimmten Stichwortes in der Wissensdatenbank suchen
  - Auflistung aller Stichworte und der dazugehörigen Beschreibung die aktuell in der Wissensdatenbank vorhanden sind anzeigen
- Die komplette Aufgabe inklusive Teilaufgaben findet ihr in der Aufgabensammlung zum Tag 2





## **Vielen herzlichen Dank für die tolle Zusammenarbeit**

- Es hat sehr viel Spass gemacht, diesen Kurs mit euch durchzuführen



## Feedback

- Ihr werdet im Anschluss an diesen Kurs von der Kursorganisation gebeten, Feedback zu diesem Kurs zu geben
- Ich bin über jedes einzelne Feedback wirklich froh und vorallem überaus dankbar



## Fragen

- Ihr dürft mich jederzeit sehr gerne nach dem Kurs kontaktieren, falls ihr zu irgendwelchen Themen (Kurs-bezogen oder nicht) fragen habt: [g@accaputo.ch](mailto:g@accaputo.ch)



## Referenzen

- Kursinhalt:
  - Allen B. Downey, "Think Python – How to Think Like a Computer Scientist" (Version 2.0.17), <http://www.thinkpython.com>
- Inspirationen einiger Aufgaben:
  - Michael Kündig, "Grundlagen der Programmierung für Nicht-Informatiker" (FS 2017), <https://bitbucket.org/mkuendig/uzh-python-course>
  - Jason Cannon, "Python Programming for Beginners" (2014)