1 Erste Schritte in Python

Version: 2019-10-28

Univ.-Prof. Dr. Martin Hepp, martin.hepp@unibw.de

1.1 Syntaktische Konventionen

1.1.1 Keine Zeilenummern

Univ.-Prof. Dr. Martin Hepp, martin.hepp@unibw.de

1.1 Syntaktische Konventionen

1.1.1 Keine Zeilenummern

Ohne Zeilennummern:
(Python etc.)

```
a = 1
b = 2
print(a)
```

1.1 Syntaktische Konventionen

1.1.1 Keine Zeilenummern

Ohne Zeilennummern:
(Python etc.)

```
a = 1
b = 2
print(a)
```

Mit Zeilennummern:
(nur in älteren Sprachen)

```
10    a = 1
20    b = 2
30    print(a)
```

1.1.2 Groß-/Kleinschreibung

Groß-/Kleinschreibung muss beachtet werden:

```
a = 10    # die Variable a wird definiert
print(A) # die Variable A gibt es aber nicht
```

1.1.3 Reservierte Namen

Namen, die für Befehle etc. verwendet werden, dürfen nicht als Namen für Werte oder Objekte genutzt werden. Es gibt

- **Schlüsselwörter** für echte Sprachelemente ("keywords") und
- **Namen für vordefinierte Objekte und Methoden** ("Built-ins").

Schlüsselwörter für echte Sprachelemente ("keywords")

In [1]:
```
1 help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

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

Namen für vordefinierte Objekte und Methoden ("Built-ins")

In [2]:

```
import builtins
seq = list(dir(builtins))
seq.sort()
max_len = len(max(seq, key=len))
chunks = [seq[pos:pos + 4] for pos in range(0, len(seq), 4)]
for chunk in chunks:
    print("".join([item.ljust(max_len + 1) for item in chunk]))
```

```
ArithmeticError             AssertionError              AttributeError              B
aseException
BlockingIOError             BrokenPipeError             BufferError                 B
ytesWarning
ChildProcessError           ConnectionAbortedError      ConnectionError             C
onnectionRefusedError
ConnectionResetError        DeprecationWarning          EOFError                    E
llipsis
EnvironmentError            Exception                   False                       F
ileExistsError
FileNotFoundError           FloatingPointError          FutureWarning               G
eneratorExit
IOError                     ImportError                 ImportWarning               I
```

1.1.4 Zuweisungs- und Vergleichsoperator

Die meisten Programmiersprachen unterscheiden zwischen

- **Zuweisung** ("a soll den Wert 5 erhalten") und
- **Vergleich** ("Entspricht a dem Wert 5?") von Werten und Ausdrücken.

1.1.4 Zuweisungs- und Vergleichsoperator

Die meisten Programmiersprachen unterscheiden zwischen

- **Zuweisung** ("a soll den Wert 5 erhalten") und
- **Vergleich** ("Entspricht a dem Wert 5?") von Werten und Ausdrücken.

In [3]:
```
1 # Python
2
3 # Zuweisung
4 a = 5
5 # Vergleich
6 # Entspricht a dem Wert 5?
7 print(a == 5)
```

```
True
```

1.2 Stil und Formatierung

1.2.1 Namen

Namen für Werte (in anderen Programmiersprachen "Variablen") sollten aussagekräftig und ohne Umlaute gewählt werden.

```
dauer = 5
zins = 0.01
```

Wenn der Name aus mehreren Wörtern besteht, werden diese durch einen Unterstrich (_) verbunden:

```
dauer_in_jahren = 5
```

Variablennamen sollten stets in **Kleinbuchstaben** sein.

Wenn der Name aus mehreren Wörtern besteht, werden diese durch einen Unterstrich (_) verbunden:

```
dauer_in_jahren = 5
```

Variablennamen sollten stets in **Kleinbuchstaben** sein.

Für Konstanten verwendet man dagegen Namen in Großbuchstaben:

```
PI = 3.1415
ABSOLUTER_NULLPUNKT = -273.15 # Grad Celsius
```

1.2.2 Leerzeichen

Vor und nach Operanden wie + oder - gehört jeweils ein Leerzeichen:

```
zins = 1 + 0.02
```

Unnötige Einrückungen sind nicht erlaubt:

```
zins = 1 + 0.02
    zinseszins = guthaben * (1 + zins)**4

____^^^^
```

Stilistische Konventionen

- Keine sonstigen unnötigen Leerzeichen, besonders nicht am Zeilenende.
- Unnötige Leerzeilen nur sparsam verwenden.
- Es gibt noch weitere stilistische Konventionen:
  - PEP 8
  - Google Python Styleguide

1.3 Grundlegende Datenstrukturen

- Alles in Python ist genaugenommen ein Objekt - jeder Wert, jedes Unterprogramm etc.

1.3 Grundlegende Datenstrukturen

- Alles in Python ist genaugenommen ein Objekt - jeder Wert, jedes Unterprogramm etc.

- Alle Objekte, also auch Werte liegen irgendwo im Arbeitsspeicher des Computers.

1.3 Grundlegende Datenstrukturen

- Alles in Python ist genaugenommen ein Objekt - jeder Wert, jedes Unterprogramm etc.

- Alle Objekte, also auch Werte liegen irgendwo im Arbeitsspeicher des Computers.

- Die Position nennt man die **Adresse**. Sie entspricht der Nummer der Speicherzelle, an der die Daten abgelegt sind, die das Objekt repräsentieren.

1.3.1 Namen und Objekte

1.3.1.1 Alles in Python ist ein Objekt

- Objekte können, müssen aber keinen Namen haben.

```
print("Hallo Welt")
print(42)
```

- Hier haben die Zeichenfolge "Hallo Welt" und die Zahl 42 keinen Namen, sind aber trotzdem Objekte mit einer Adresse.

Die Adresse eines Objektes im Speicher kann man mit der Funktion `id(name)` zeigen:

Die Adresse eines Objektes im Speicher kann man mit der Funktion `id(name)` zeigen:

In [4]:
```
1 print(type("Hallo Welt"), type(42))
2 print(id("Hallo Welt"), id(42))
```

```
<class 'str'> <class 'int'>
4579809392 4541229456
```

Die Adresse eines Objektes im Speicher kann man mit der Funktion `id(name)` zeigen:

In [4]:
```
1 print(type("Hallo Welt"), type(42))
2 print(id("Hallo Welt"), id(42))
```

```
<class 'str'> <class 'int'>
4579809392 4541229456
```

- `str` und `int` sind die Typen der Objekte
- str/String = Zeichenkette und int/Integer = Ganzzahl
- Die Zahlen darunter sind die Adressen des Objektes.

1.3.1.2 Objekte *können* Namen haben

In [5]:
```
1 mein_text = "Hallo Welt"
2 meine_zahl = 42
```

1.3.1.2 Objekte *können* Namen haben

In [5]:
```
1 mein_text = "Hallo Welt"
2 meine_zahl = 42
```

Diese Namen verweisen auf die **Adresse** des Objektes:

1.3.1.2 Objekte *können* Namen haben

In [5]:
```
1 mein_text = "Hallo Welt"
2 meine_zahl = 42
```

Diese Namen verweisen auf die **Adresse** des Objektes:

In [117]:
```
1 print(id(mein_text))
2 print(id(meine_zahl))
```

```
4579625328
4541229456
```

Das ist ein wesentlicher Unterschied zu anderen Programmiersprachen. In Python führt eine Anweisung wie

```
variable = 1234
```

nicht dazu, dass eine Variable ***erzeugt*** wird, die mit dem Wert 1234 ***initial gefüllt*** wird.

Stattdessen wird geprüft, ob es das Objekt der Zahl 1234 schon gibt. Falls nicht, wird eines im Speicher erzeugt. Dann wird die Adresse dieses Objektes als Verweis dem Namen `variable` zugewiesen, also damit verbunden.

Der Name `variable` wird also mit dem Objekt/Wert verbunden.

[vgl. Fredrik Lundh: Call by Object]

Mehrere Anweisungen wie

```
zahl_1 = 42
zahl_2 = 42
zahl_3 = 42
```

führen daher nicht dazu, dass drei Variablen erzeugt werden, sondern dass drei Namen definiert werden, über die man die Ganzzahl 42 ansprechen kann.

1.3.1.3 Mehrfachzuweisung

Man kann übrigens auch in einer Anweisung mehrere Namen für ein Objekt definieren:

In [7]:
```
1 a = b = c = 3
```

1.3.1.3 Mehrfachzuweisung

Man kann übrigens auch in einer Anweisung mehrere Namen für ein Objekt definieren:

In [7]:
```
1 a = b = c = 3
```

Verständnischeck: Wenn wir nun

```
b = 4
```

ausführen, was passiert?

In [8]:

```
1 a = b = c = 3
2 print(a, b, c)
3 b = 4
4 print(a, b, c)
5
```

```
3 3 3
3 4 3
```

Nur der Wert von `b` ändert sich, weil die Verweise der anderen Namen nicht berührt werden.

1.3.2 Mutable und Immutable Objects

Es gibt in Python Objekte,

- die man verändern kann (**"Mutable Objects"**), und
- solche, die unveränderlich sind (**"Immutable Objects"**).

Zahlen und Zeichenketten sind unveränderlich.

Das heißt aber nicht, dass man den Wert von Variablen dieser Typen nicht ändern könnte:

Das heißt aber nicht, dass man den Wert von Variablen dieser Typen nicht ändern könnte:

In [9]:
```
1 text = "Uni"
2 print(text)
3 text = "FH"
4 print(text)
```

```
Uni
FH
```

Das heißt aber nicht, dass man den Wert von Variablen dieser Typen nicht ändern könnte:

In [9]:
```
1 text = "Uni"
2 print(text)
3 text = "FH"
4 print(text)
```

```
Uni
FH
```

In [10]:
```
1 zahl = 1
2 print(zahl)
3 zahl = zahl + 1
4 print(zahl)
```

```
1
2
```

Das heißt aber nicht, dass man den Wert von Variablen dieser Typen nicht ändern könnte:

In [9]:
```
1 text = "Uni"
2 print(text)
3 text = "FH"
4 print(text)
```

```
Uni
FH
```

In [10]:
```
1 zahl = 1
2 print(zahl)
3 zahl = zahl + 1
4 print(zahl)
```

```
1
2
```

Hier wird jeweils nicht Variable mit einem neuen Wert überschrieben, sondern der neue Wert als neues Objekt erzeugt und der Name mit der Adresse des neuen Objektes verbunden.

1.3.3 Ausgabe mit *print*

Man kann den Wert jeder Variable und jeden mathematischen Ausdruck mit dem Befehl `print(<ausdruck>)` auf dem Bildschirm anzeigen lassen:

1.3.3 Ausgabe mit *print*

Man kann den Wert jeder Variable und jeden mathematischen Ausdruck mit dem Befehl `print(<ausdruck>)` auf dem Bildschirm anzeigen lassen:

In [11]:
```
1 print(1 + 4)
2 print('Hallo')
```

```
5
Hallo
```

In [12]:
```
1 # Mehrere Werte durch Kommata getrennt
2 print(1, 5 * 3, 'Hallo', 10.3)
```

```
1 15 Hallo 10.3
```

1.3.4 Numerische Werte

Numerische Werte, wie

- Zahlen wie 5 oder -1.23
- Mathematische Konstanten wie π oder e
- Unendlich ($\infty / -\infty$) und Not-a-Number sind die häufigsten Arten von Objekten in den meisten Programmen.

1.3.4.1 Ganze Zahlen

Ganze Zahlen werden in Python durch den Datentyp `int` repräsentiert und können beliebig große Werte annehmen (vgl. Numeric Types — int, float, complex).

In [13]:
```
1 a = 15
2 b = -7
3 c = 240
```

Man kann auch eine andere Basis als 10 wählen und dadurch elegant **Binärzahlen** und **Hexadezimalzahlen** erzeugen:

In [14]:
```
1 # Binärzahlen
2 a = 0b00001111
3 # Hexadezimalzahlen
4 c = 0xF0
5 print(a, c)
```

```
15 240
```

1.3.4.2 Gleitkommazahlen

Wenn ein übergebener Wert einen Dezimalpunkt oder einen Exponenten enthält, wird daraus in Python ein Objekt vom Typ `float` erzeugt.

In [118]:
```
1 wert_1 = float(1/3)
2 print(wert_1)
3 print(wert_1 * 3)
```

```
0.3333333333333333
1.0
```

1.3.4.2 Gleitkommazahlen

Wenn ein übergebener Wert einen Dezimalpunkt oder einen Exponenten enthält, wird daraus in Python ein Objekt vom Typ `float` erzeugt.

In [118]:
```
1 wert_1 = float(1/3)
2 print(wert_1)
3 print(wert_1 * 3)
```

```
0.3333333333333333
1.0
```

In [120]:
```
1 # Achtung, Genauigkeitprobleme!
2 wert_2 = wert_1 / 10000
3 print((wert_2 * 10000 * 3))
```

```
0.9999999999999998
```

Bei einem Python-Objekt vom typ `float` handelt es sich auf fast jedem Computersystem um eine Gleitkommazahl mit 64 Bit.

Bei einem Python-Objekt vom typ `float` handelt es sich auf fast jedem Computersystem um eine Gleitkommazahl mit 64 Bit.

Die Genauigkeit und der Wertebereich entsprechen daher dem, was in anderen Programmiersprachen der Typ `double` bietet.

Man muss dazu wissen, das Python in den neueren Versionen versucht, die Beschränkungen von Gleitkommazahlen bei der Ausgabe duch geschickte Rundungsregeln zu verbergern. So wird $1/3$ intern als eine Gleitkommazahl mit einer begrenzten Anzahl an Stellen gespeichert.

Zu den Beschränkungen und Fehlerquellen beim Rechnen mit Gleitkommazahlen vgl. Floating Point Arithmetic: Issues and Limitations.

1.3.4.3 Dezimalzahlen

Wenn es wichtig ist, dass Zahlen genau in der gegebenen Genauigkeit gespeichert und verarbeitet werden, sind Dezimalzahlen mit einer festen Stellenzahl besser geeignet.

Dies betrifft insbesondere Geldbeträge.

Weitere Informationen: https://docs.python.org/3/library/decimal.html

1.3.4.4 Unendlich (∞)

Der Wert unendlich kann in Python auf zwei Wegen verwendet werden:

In [ ]:
```
1 positive_infinity = float('inf')
2 negative_infinity = float('-inf')
```

1.3.4.4 Unendlich (∞)

Der Wert unendlich kann in Python auf zwei Wegen verwendet werden:

In [ ]:
```
positive_infinity = float('inf')
negative_infinity = float('-inf')
```

In [ ]:
```
import math
positive_infinity_2 = math.inf
negative_infinity2 = -math.inf
```

1.3.4.5 Not-a-Number (NaN)

Es gibt Operationen, bei denen sich das Ergebnis nicht als reelle Zahl abspeichern lässt. Ferner kann bei der Verarbeitung eigentlich numerischer Werte durch Datenqualitätsprobleme der Fall eintreten, dass einzelne Werte keine Zahlen sind. Für diesen Fall gibt es einen besonderen Wert, der sich **NaN** für "Not a number" nennt. Beispiele:

- ∞/∞
- Quadratwurzel aus negativen Werten

In [ ]:

```
1 not_a_number = float('NaN')
2 print(100 * not_a_number)
```

In [ ]:

```
1 not_a_number = float('NaN')
2 print(100 * not_a_number)
```

Der wesentliche Nutzen dieses Wertes besteht darin, dass man die Ungültigkeit einer Berechnung erkennen kann.

1.3.5 Mathematische Operationen

1.3.5.1 Arithmetische Operationen

In [17]:
```
a = 1
b = 2
c = 3
```

In [18]:
```
# Grundrechenarten
d = a + b
print(d)
```

```
3
```

In [19]:
```
1 e = c - a
2 print(e)
```

```
2
```

In [19]:
```
1 e = c - a
2 print(e)
```

```
2
```

In [20]:
```
1 f = b * e
2 print(f)
```

```
4
```

In [21]:
```
1 g = f / b
2 print(g)
```

```
2.0
```

In [21]:
```
1 g = f / b
2 print(g)
```

```
2.0
```

Achtung: Seit Python 3.0 ist die Standard-Division eine Gleitkommadivision, 5 / 2 ist also 2.5. In früheren Versionen wurde wurde standardmäßig eine ganzzahlige Division durchgeführt, also 5/2 = 2 (Rest 1).

1.3.5.2 Potenz

x^y in Python als `x**y`

In [22]:
```
# Potenzfunktionen
h = 2**7   # 2^7 = 128
print(h)

```

```
128
```

1.3.5.3 Wurzel

Direkt in Python gibt es keine Funktion für die Quadratwurzel, weil man dies einfach als Potenzfunktion mit einem Bruch als Exponenten ausdrücken kann: $\sqrt{x} = x^{\frac{1}{2}}$

In [23]:
```
1 # Quadratwurzel
2 a = 16
3 print(a**(1/2))
4 print(a**0.5)
```

```
4.0
4.0
```

In [24]:
```
# Es gibt auch ein spezielles Modul math mit zusätzlichen mathematischen Methode
import math
a = 16
print(math.sqrt(a))
```

```
4.0
```

1.3.5.4 Ganzzahlige Division

In [25]:
```
1 a = 5
2 b = 2
3 print(a // b)
```

```
2
```

1.3.5.5 Divisionsrest (modulo)

- **Tip 1:** Nützlich, um zu prüfen, ob eine Zahl gerade ist.
- **Tip 2:** Auch nützlich, wenn man den Wertebereich einer Zahl begrenzen will.

In [26]:
```
1 # Modulo / Divisionsrest
2 print(a % b)
```

```
1
```

1.3.6 Umwandlung des Datentyps numerischer Werte

In [27]:
```
# float als int
# Was passiert?
print(int(3.1))
print(int(3.5))
print(int(3.8))
```

```
3
3
3
```

Univ.-Prof. Dr. Martin Hepp, martin.hepp@unibw.de

1.3.6 Umwandlung des Datentyps numerischer Werte

In [27]:
```
# float als int
# Was passiert?
print(int(3.1))
print(int(3.5))
print(int(3.8))
```

```
3
3
3
```

In [28]:
```
# int als float
print(float(7))
```

```
7.0
```

In [29]:
```
1 # int als Binärzahl
2 print(bin(255))
```

```
0b11111111
```

In [29]:
```
1 # int als Binärzahl
2 print(bin(255))
```

```
0b11111111
```

In [30]:
```
1 # int als Hexadezimalzahl
2 print(hex(255))
```

```
0xff
```

1.3.7 Rundung

In [31]:
```
1 # Runden
2 # round(value[, n])
3 print(round(3.5))
```

```
4
```

1.3.7 Rundung

In [31]:
```
1 # Runden
2 # round(value[, n])
3 print(round(3.5))
```

```
4
```

Der optionale zweite Parameter gibt an, wieviele Nachkommastellen gewünscht werden:

1.3.7 Rundung

In [31]:
```
# Runden
# round(value[, n])
print(round(3.5))
```

```
4
```

Der optionale zweite Parameter gibt an, wieviele Nachkommastellen gewünscht werden:

In [32]:
```
# Wir runden Pi auf drei Stellen nach dem Komma
print(round(3.1415926, 3))
```

```
3.142
```

1.3.8 Wahrheitswerte (Boolesche Werte)

Ähnlich wie wir in der elementaren Algebra mit Zahlen arbeiten, kann man in der sogenannten Booleschen Algebra mit den Wahrheitswerten `wahr(true)` und `unwahr(false)` arbeiten. Als Operatoren stehen uns **AND** (Konjunktion), **OR** (Disjunktion), **XOR** (Kontravalenz) und **NOT** (Negation) zur Verfügung.

Zwei (oder mehr) Boolesche Werte kann man mit den Operatoren AND, OR oder XOR verknüpfen. Mit NOT kann man einen Booleschen Wert invertieren:

a	b	a AND b	a OR b	NOT a	a XOR b
False	False	False	False	True	False
True	False	False	True	False	True
False	True	False	True	True	True
True	True	True	True	False	False

Praktisch relevant ist dies z.B. bei Suchmaschinen

```
"finde alle Bücher, die entweder 'Informatik' oder 'BWL' im Titel enthalten"
```

und bei Bedingungen in Geschäftsprozessen

```
"Kreditkarte_gültig AND Produkt_lieferbar"
```.

1.3.8.1 Boolesche Werte und Operatoren in Python

In [33]:
```
# Wahr und Falsch sind vordefinierte Werte
# Achtung: Schreibweise!
a = True
b = False
```

1.3.8.1 Boolesche Werte und Operatoren in Python

In [33]:
```
# Wahr und Falsch sind vordefinierte Werte
# Achtung: Schreibweise!
a = True
b = False
```

In [34]:
```
# Logische Operatoren
print(a and b)
print(a or b)
print(not a)
print(bool(a) ^ bool(b))
```

```
False
True
False
True
```

1.3.8.2 Boolesche Werte lassen sich in Ganzzahlen umwandeln

In [35]:
```
1 print(int(True))
2 print(int(False))
```

```
1
0
```

1.3.8.2 Boolesche Werte lassen sich in Ganzzahlen umwandeln

In [35]:
```
print(int(True))
print(int(False))
```

```
1
0
```

In [36]:
```
# Ziemlich nützlich bei Berechnungen
versandkostenpflichtig = True
versandkosten = 5.95
nettobetrag = 135.00
bruttobetrag = nettobetrag + versandkosten * versandkostenpflichtig
print(bruttobetrag)
```

```
140.95
```

1.3.9 Vergleichsoperatoren

In einem Programm muss man oft den Wert von Objekten vergleichen, z.B. den Lagerbestand mit einer Mindestmenge. Dazu gibt es sogenannte **Vergleichsoperatoren**. Das Ergebnis ist immer ein Boolescher Wert, also `True` oder `False`.

In [37]:

```
1 a = 90
2 b = 60
3 c = 60
```

In [37]:
```
1 a = 90
2 b = 60
3 c = 60
```

In [38]:
```
1 print(a < b)
```

```
False
```

In [39]:
```
1 print(a > b)
```

```
True
```

In [39]:
```
1 print(a > b)
```

```
True
```

In [40]:
```
1 print(a < a)
```

```
False
```

In [41]:
```
1 print(a <= a)
```

```
True
```

In [41]:
```
1 print(a <= a)
```

```
True
```

In [42]:
```
1 print(a >= a)
```

```
True
```

1.3.10 Wertevergleich oder Identitätsvergleich?

Wenn man Ausdrücke oder Objekte vergleicht, muss man sich überlegen, ob man

1. den **Wert** der Ausdrücke vergleichen will, oder
2. ob geprüft werden soll, ob es sich um **dasselbe Objekt** handelt.

Wertevergleich mit `a == b`

Identitäsvergleich mit `a is b`

In [43]:
```
1 # Bei numerischen Ausdrücken gibt es keinen Unterschied:
2 print(3 * 5 == 15)
3 print(3 * 5 is 15)
```

```
True
```

In [44]:

```
# Dito bei Strings
a = "Text"
b = "Text"
print(a == b)
print(a is b)
# Warum?
```

```
True
True
```

Bei änderbaren Objekten (Mutables) sieht es aber anders aus:

In [45]:
```
1 a = [1, 2]
2 b = [1, 2]
3 print(a == b)
4 print(a is b)
```

```
True
False
```

Bei änderbaren Objekten (Mutables) sieht es aber anders aus:

In [45]:
```
1 a = [1, 2]
2 b = [1, 2]
3 print(a == b)
4 print(a is b)
```

```
True
False
```

Das liegt daran, dass änderbare Objekten im Speicher eigene Plätze einnehmen, weil der Computer ja nicht wissen kann, ob sie immer identisch bleiben.

Beim Wertevergleich mit == wird automatisch eine Typumwandlung versucht:

In [46]:
```
1 print(5 == 5.0)
```

```
True
```

Beim Identitätsvergleich sind verschiedene Datentypen verschiedene Objekte, selbst wenn sich ihre Werte umwandeln ließen:

Beim Identitätsvergleich sind verschiedene Datentypen verschiedene Objekte, selbst wenn sich ihre Werte umwandeln ließen:

In [47]:

```
1 print(5 is 5.0)
2 print(True is 1)
```

```
False
False
```

Beim Identitätsvergleich sind verschiedene Datentypen verschiedene Objekte, selbst wenn sich ihre Werte umwandeln ließen:

In [47]:
```
1 print(5 is 5.0)
2 print(True is 1)
```

```
False
False
```

In [48]:
```
1 # Aber:
2 print(5 is int(5.0))
3 print(int(True) is 1)
4 print(True is bool(1))
```

```
True
True
True
```

1.3.11 Trigonometrische Funktionen

Siehe auch https://docs.python.org/3/library/math.html.

In [49]:
```
import math

# Pi
print(math.pi)
# Eulersche Zahl
print(math.e)
```

```
3.141592653589793
2.718281828459045
```

In [50]:

```
# Quadratwurzel
print(math.sqrt(16))
# Sinus
print(math.sin(90))
# Cosinus
print(math.cos(math.pi))
# Tangens
print(math.tan(math.pi))
# Log2
print(math.log2(256))
```

```
4.0
0.8939966636005579
-1.0
-1.2246467991473532e-16
8.0
```

1.3.12 Komplexe Datentypen

Als komplexe Datentypen bezeichnet man solche, die eine adressierbare Struktur an Unterelementen haben.

- Zeichenketten

0	1	2
W	O	W

- Listen
- Dictionaries
- Tuples uvm.

1.3.12.1 Zeichenketten

In [51]:
```
# Zeichenkette
my_string_1 = 'UniBwM'
my_string_2 = "UniBwM"
```

In [52]:
```
# Die Wahl zwischen einfachen und doppelten Anführungszeichen erlaubt es elegant
# die jeweils andere Form innerhalb der Zeichenkette zu verwenden:
my_string_3 = "Die Abkürzung für unsere Universität lautet 'UniBwM'."
my_string_3 = 'Die Abkürzung für unsere Universität lautet "UniBwM".'
```

In [53]:

```
# Mehrzeilige Zeichenketten erfordern DREI Anführungszeichen:
my_long_string_1 = """Herr von Ribbeck auf Ribbeck im Havelland,
Ein Birnbaum in seinem Garten stand,
Und kam die goldene Herbsteszeit,
Und die Birnen leuchteten weit und breit,
Da stopfte, wenn’s Mittag vom Thurme scholl,
Der von Ribbeck sich beide Taschen voll,
Und kam in Pantinen ein Junge daher,
So rief er: 'Junge, wist’ ne Beer?'
Und kam ein Mädel, so rief er: 'Lütt Dirn'
Kumm man röwer, ick hebb’ ne Birn."""
```

In [54]:

```
my_long_string_2 = '''Herr von Ribbeck auf Ribbeck im Havelland,
Ein Birnbaum in seinem Garten stand,
Und kam die goldene Herbsteszeit,
Und die Birnen leuchteten weit und breit,
Da stopfte, wenn’s Mittag vom Thurme scholl,
Der von Ribbeck sich beide Taschen voll,
Und kam in Pantinen ein Junge daher,
So rief er: "Junge, wist’ ne Beer?"
Und kam ein Mädel, so rief er: "Lütt Dirn"
Kumm man röwer, ick hebb’ ne Birn.'''
```

Addition von Zeichenketten

In [55]:
```
1 my_string_1 = "UniBwM"
2 print('Ich studiere an der ' + my_string_1)
```

```
Ich studiere an der UniBwM
```

Multiplikation von Zeichenketten

In [56]:
```
1 print('ABCD' * 3)
```

```
ABCDABCDABCD
```

Univ.-Prof. Dr. Martin Hepp, martin.hepp@unibw.de

Multiplikation von Zeichenketten

In [56]:
```
1 print('ABCD' * 3)
```

```
ABCDABCDABCD
```

In [57]:
```
1 # Nützlich z.B. für
2 print('=' * 60)
3 print('Progamm ABCD Version 1.0')
4 print('=' * 60)
```

```
============================================================
Progamm ABCD Version 1.0
============================================================
```

Aber man kann keine Zeichenketten *miteinander* multiplizieren:

In [58]:

```
1 my_string_test = '11'
2 second_string = '2'
3 print(my_string_test * second_string)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-1f4b1a3739c9> in <module>
      1 my_string_test = '11'
      2 second_string = '2'
----> 3 print(my_string_test * second_string)

TypeError: can't multiply sequence by non-int of type 'str'
```

Länge ermitteln

In [60]:
```
1 my_string = "LOTTO"
2 print(len(my_string))
```

```
5
```

Weitere Hilfsfunktionen für Strings

In [123]:
```
# https://docs.python.org/3/library/stdtypes.html
text = "UniBwM ist toll"
print(text.lower())
print(text.upper())
print(text.split(" "))
```

```
unibwm ist toll
UNIBWM IST TOLL
['UniBwM', 'ist', 'toll']
```

1.3.12.2 Listen

Listen sind komplexe Variablen aus mehreren Unterelementen beliebigen Typs. Die Unterelemente können einzeln adressiert und auch geändert werden.

In [61]:
```
# Liste
my_list = [1, 2, 5, 9]
my_list_mixed = [1, True, 'UniBwM']
print(my_list_mixed)
```

```
[1, True, 'UniBwM']
```

In [62]:
```
# Listenelemente können einzeln adressiert werden.
# Das erste Element hat den Index 0:
print(my_list[0])
print(my_list[1])
print(my_list[2])
```

```
1
2
5
```

In [62]:
```
# Listenelemente können einzeln adressiert werden.
# Das erste Element hat den Index 0:
print(my_list[0])
print(my_list[1])
print(my_list[2])
```

```
1
2
5
```

In [63]:
```
# Sie können auch einzeln geändert werden:
my_list_mixed[2] = 'LMU München'
print(my_list_mixed)
```

```
[1, True, 'LMU München']
```

In [64]:

```
# Man kann auch Bereiche adressieren:
my_list = ['one', 'two', 'three', 'four', 'five']

# Alle ab dem zweiten Element
print(my_list[1:])
# Alle bis zum zweiten Element
print(my_list[:2])
# Alle vom zweiten bis zum dritten Element
print(my_list[1:3])
# Alle ohne die letzten beiden
print(my_list[:-2])

```

```
['two', 'three', 'four', 'five']
['one', 'two']
['two', 'three']
['one', 'two', 'three']
```

In [65]:
```
1 # Man kann auch Bereiche einer Liste ändern oder die Liste dadurch verkürzen ode
2 my_list[1:3] = ['zwei', 'drei']
3 print(my_list)
```

```
['one', 'zwei', 'drei', 'four', 'five']
```

In [65]:
```
# Man kann auch Bereiche einer Liste ändern oder die Liste dadurch verkürzen ode
my_list[1:3] = ['zwei', 'drei']
print(my_list)
```

```
['one', 'zwei', 'drei', 'four', 'five']
```

In [66]:
```
my_list = ['one', 'two', 'three', 'four', 'five']
my_list[0:2] = ['one_and_two']
print(my_list)
```

```
['one_and_two', 'three', 'four', 'five']
```

In [67]:

```
# Achtung: Wenn man einen ListenBEREICH ändert, muss man eine Liste übergeben.
my_list = ['one', 'two', 'three', 'four', 'five']
my_list[0:2] = ['one_and_two']
print(my_list)
```

```
['one_and_two', 'three', 'four', 'five']
```

In [68]:

```
# Sonst Versucht python, den Wert als Liste seiner Unterelemente zu verstehen.
my_list = ['one', 'two', 'three', 'four', 'five']
my_list[0:2] = ['ABC']
print(my_list)
```

```
['ABC', 'three', 'four', 'five']
```

In [69]:
```
# Wenn man ein Listenelement ändert, muss man ein Element übergeben
my_list = ['one', 'two', 'three', 'four', 'five']
my_list[2] = 'drei'
print(my_list)
```

```
['one', 'two', 'drei', 'four', 'five']
```

In [70]:

```
# Wenn man an einer Position MEHRE neue Elemente einfügen will,
# muss man diese Position als Bereich der Länge 1 adressieren.

my_list = ['one', 'two', 'three', 'four', 'five']
my_list[2:3] = ['drei_a', 'drei_b']
print(my_list)
```

```
['one', 'two', 'drei_a', 'drei_b', 'four', 'five']
```

In [71]:
```
# Sonst würde an dieser Stelle eine Liste als Element eingefügt:
my_list = ['one', 'two', 'three', 'four', 'five']
my_list[2] = ['drei_a', 'drei_b']
print(my_list)
```

```
['one', 'two', ['drei_a', 'drei_b'], 'four', 'five']
```

In [72]:

```
1 # Listen: append()
2 my_list = ['one', 'two']
3 my_list.append('three')
4 print(my_list)
```

```
['one', 'two', 'three']
```

In [73]:
```
1 # Listen: extend()
2 my_list = ['one', 'two']
3 my_list.extend(['three', 'four'])
4 print(my_list)
```

```
['one', 'two', 'three', 'four']
```

In [74]:
```
# Kontrollfrage:
# Was passiert, wenn Sie der Methode 'append' eine LISTE als Parameter übergeben
my_list = ['one', 'two']
my_list.append(['three', 'four'])
print(my_list)
```

```
['one', 'two', ['three', 'four']]
```

In [75]:
```
# Kontrollfrage:
# Was passiert, wenn Sie der Methode 'extend' einen einzelnen Wert als Parameter
my_list = ['one', 'two']
my_list.extend('three')
print(my_list)
```

```
['one', 'two', 't', 'h', 'r', 'e', 'e']
```

In [76]:

```
1 # Wenn eine atomare Variable übergeben wird, gibt es eine Fehlermeldung:
2 my_list = ['one', 'two']
3 my_list.extend(1)
4 print(my_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-76-bca24c3ec178> in <module>
      1 # Wenn eine atomare Variable übergeben wird, gibt es eine Fehlermeldun
g:
      2 my_list = ['one', 'two']
----> 3 my_list.extend(1)
      4 print(my_list)

TypeError: 'int' object is not iterable
```

In [78]:
```
# Man kann Listen einfach sortieren:
my_list = [1, 6, 5, 3, 2, 4]
my_list.sort()
print(my_list)
```

```
[1, 2, 3, 4, 5, 6]
```

In [79]:
```
# Das geht auch mit inverser Sortierfolge:
my_list = [1, 6, 5, 3, 2, 4]
my_list.sort(reverse=True)
print(my_list)
```

```
[6, 5, 4, 3, 2, 1]
```

In [80]:
```
1 # Prüfen, ob Element in Liste enthalten
2 my_liste_3 = [1, 4, 9, 7]
3 print(2 in my_liste_3)
```

```
False
```

In [80]:
```
# Prüfen, ob Element in Liste enthalten
my_liste_3 = [1, 4, 9, 7]
print(2 in my_liste_3)
```

```
False
```

In [81]:
```
# Suchen zu Fuß
my_liste_4 = ['Hepp', 'Mueller', 'Meier']
positionszaehler = 0
for element in my_liste_4:
    if element == 'Meier':
        print('Täter gefunden!')
        print('Position:', positionszaehler)
    positionszaehler = positionszaehler + 1
```

```
Täter gefunden!
Position: 2
```

In [82]:
```
1 if 'Mueller' in my_liste_4:
2     print('Täter gefunden!')
```

```
Täter gefunden!
```

In [82]:
```
1 if 'Mueller' in my_liste_4:
2     print('Täter gefunden!')
```

```
Täter gefunden!
```

In [121]:
```
1 if 'Mueller' in my_liste_4:
2     print(my_liste_4.index('Mueller'))
```

```
1
```

1.3.12.3 Tuples

In [83]:
```
# Tuple
my_tuple = (1, 3, 9)
my_tuple_mixed = (1, True, 'UniBwM')

latitude = 48.0803
longitude = 11.6382
geo_position = (latitude, longitude)
```

In [84]:
```
1 # Entpacken
2 lat, lon = geo_position
3 print(lat)
```

```
48.0803
```

In [84]:
```
# Entpacken
lat, lon = geo_position
print(lat)
```

```
48.0803
```

In [85]:
```
# Auch die Elemente eines Tuples können einzeln adressiert werden:
print(geo_position[0])
```

```
48.0803
```

In [86]:
```
1 # Die Unterelemente eines Tuples können aber nicht geändert werden:
2 geo_position[0] = 44.123
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-623b6a40f573> in <module>
      1 # Die Unterelemente eines Tuples können aber nicht geändert werden:
----> 2 geo_position[0] = 44.123

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

1.3.12.4 Dictionaries

In [87]:
```
my_dict = {'name' : 'Martin Hepp',
           'fakultaet' : 'WOW',
           'geburtsjahr' : 1971}
print(my_dict['name'])
print(my_dict['fakultaet'])

```

```
Martin Hepp
WOW
```

In [88]:
```
# Elemente können geändert und hinzugefügt werden
my_dict['fakultaet'] = 'INF'
my_dict['lieblingsvorlesung'] = 'Programmierung in Python'
```

In [88]:
```
# Elemente können geändert und hinzugefügt werden
my_dict['fakultaet'] = 'INF'
my_dict['lieblingsvorlesung'] = 'Programmierung in Python'
```

In [89]:
```
# Wenn es den Schlüssel ('key') nicht gibt, wird eine Fehlermeldung produziert:
print(my_dict['einkommen'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-89-909ad30b7b4b> in <module>
      1 # Wenn es den Schlüssel ('key') nicht gibt, wird eine Fehlermeldung pro
duziert:
----> 2 print(my_dict['einkommen'])

KeyError: 'einkommen'
```

In [90]:
```
1 # Das kann man mit der Methode get vermeiden:
2 print(my_dict.get('einkommen'))
```

```
None
```

In [90]:
```
# Das kann man mit der Methode get vermeiden:
print(my_dict.get('einkommen'))
```

```
None
```

In [91]:
```
# Man kann auch einen Default-Wert vorgeben (normalerweise None):
print(my_dict.get('einkommen', 'Unbekannt'))
print(my_dict.get('einkommen', 0))
```

```
Unbekannt
0
```

In [92]:
```
1 adresse = {}
2 print(adresse)
```

```
{}
```

In [92]:
```
1 adresse = {}
2 print(adresse)
```

```
{}
```

In [93]:
```
1 adresse = {}
2 adresse['plz'] = '85577'
3 print(adresse['plz'])
4 adresse['sonderfeld'] = 'Bemerkungen bitte hier'
5 print(adresse)
```

```
85577
{'plz': '85577', 'sonderfeld': 'Bemerkungen bitte hier'}
```

In [94]:

```
gast_1 = {'name' : 'Frank Farian'}
gast_2 = {'name' : 'Lady Gaga'}
gast_3 = {'name' : 'John Lennon'}

gaesteliste = []
gaesteliste.append(gast_1)
gaesteliste.append(gast_2)
gaesteliste.append(gast_3)
gast_2['bemerkung'] = 'Supercool!'
print(gaesteliste)

```

```
[{'name': 'Frank Farian'}, {'name': 'Lady Gaga', 'bemerkung': 'Supercool!'},
{'name': 'John Lennon'}]
```

In [95]:
```
1 for gast in gaesteliste:
2     print(gast['name'], gast.get('bemerkung', ''))
```

```
Frank Farian
Lady Gaga Supercool!
John Lennon
```

In [95]:
```
for gast in gaesteliste:
    print(gast['name'], gast.get('bemerkung', ''))
```

```
Frank Farian
Lady Gaga Supercool!
John Lennon
```

In [96]:
```
gast_2['bemerkung'] = 'Supercool!'
print(gaesteliste)
```

```
[{'name': 'Frank Farian'}, {'name': 'Lady Gaga', 'bemerkung': 'Supercool!'},
{'name': 'John Lennon'}]
```

1.3.12.5 Sets (Mengen)

In [97]:
```
1 a = set(['rot', 'gruen', 'blau', 'gelb'])
2 print(a)
```

```
{'gelb', 'rot', 'blau', 'gruen'}
```

In [98]:
```
1 a = 'Dies ist eine Zeichenkette.'
2 # Nun schauen wir, welche Buchstaben hierin vorkommen.
3 zeichenvorrat = set(a)
4 print(zeichenvorrat)
```

```
{' ', '.', 'i', 'h', 'e', 'Z', 'k', 's', 'n', 'c', 'D', 't'}
```

1.3.13 Benutzereingabe mit `input()`

In [99]:
```
1 # Benutzereingabe mit input([text])
2 # Ergebnis ist Zeichenkette (s.u.)
3 eingabe = input('Ihr Name?')
```

```
Ihr Name?
```

1.3.14 Typumwandlung (Type Cast)

In [100]:
```
1 # Zeichenkette in Ganzzahl (int)
2 zahl_als_text = "7"
3 zahl_als_int = int(zahl_als_text)
```

1.3.14 Typumwandlung (Type Cast)

In [100]:
```
1 # Zeichenkette in Ganzzahl (int)
2 zahl_als_text = "7"
3 zahl_als_int = int(zahl_als_text)
```

In [101]:
```
1 # Zeichenkette als Gleitkommazahl (float)
2 float_als_text = "3.1415"
3 float_als_zahl = float(float_als_text)
```

In [102]:

```
1 # Zahl als Zeichenkette (String)
2 zahl_als_text = str(7)
3 float_als_text = (str(3.1415))
4
```

In [103]:

```
# Umwandlung einer Zahl in eine Zeichenkette
a = 42
a_string = str(a)

# Was ist hier der Unterschied?
print(a * 2)
print(a_string * 2)
```

```
84
4242
```

1.4 Übungsaufgaben

Univ.-Prof. Dr. Martin Hepp, martin.hepp@unibw.de

1.4.1 Übung: Satz des Pythagoras

Schreiben Sie diese Formel als Python-Ausdruck

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

In [105]:
```
1 a = 5
2 b = 7
```

In [106]:
```
1 c = ?
```

In [107]:
```
1 print(c)
```

```
8.602325267042627
```

1.4.2 Übung: Formel für Kreisfläche

Schreiben Sie die Formel für die Kreisfläche als Python-Ausdruck

$$flaeche = \pi * r^2$$

1.4.2 Übung: Formel für Kreisfläche

Schreiben Sie die Formel für die Kreisfläche als Python-Ausdruck

$$flaeche = \pi * r^2$$

In [108]:
```
1 r = 3
```

In [ ]:
```
1 flaeche = ?
```

In [ ]:
```
1 print(flaeche)
```

1.5 Erweiterungen

Univ.-Prof. Dr. Martin Hepp, martin.hepp@unibw.de

1.5.1 Python 3.6 Type Annotations, PEP 526

In [112]:

```
# http://mypy.readthedocs.io/en/latest/cheat_sheet_py3.html
# https://www.python.org/dev/peps/pep-0526/
a: int = 1
b: int
b = 2
print(a)
print(b)
```

```
1
2
```

In [113]:

```python
from typing import List, Set, Dict, Tuple, Text, Optional, AnyStr
my_list: List[int] = []
my_list.append(1)
print(my_list)

def test(number: int) -> float:
    return float(number)

print(test(1))
print(type(test(1)))
```

```
[1]
1.0
<class 'float'>
```

Vielen Dank!

http://www.ebusiness-unibw.org/wiki/Teaching/PIP

```
In [ ]: 1
```

Univ.-Prof. Dr. Martin Hepp, martin.hepp@unibw.de