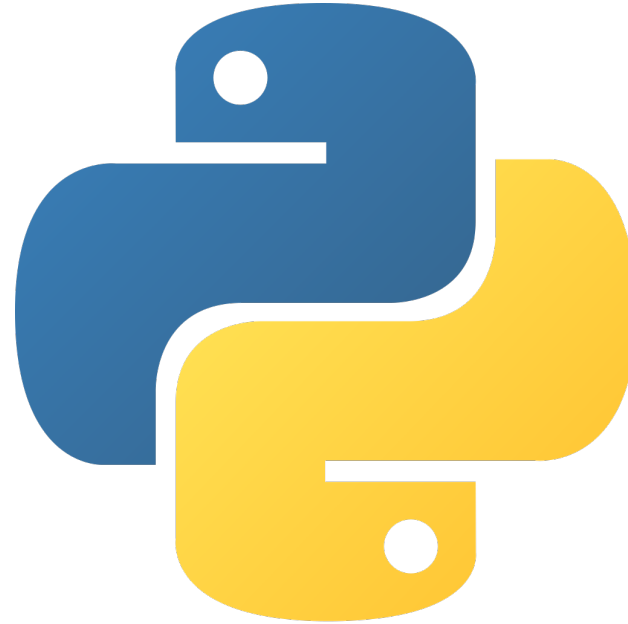
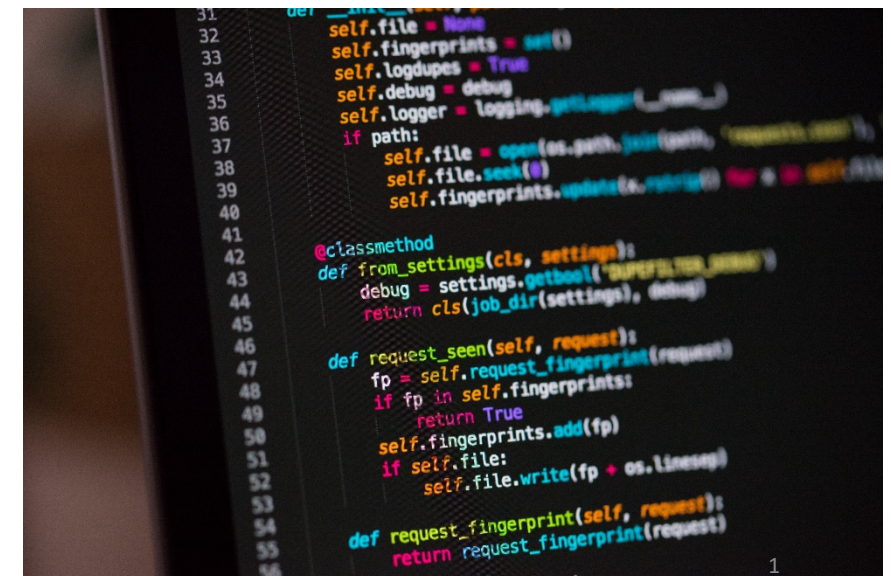


# Einführung in Python



Prof. Dr.-Ing. Prof. h.c. Detlef Jensen  
Dekan Fachbereich Technik

Tel.: +49 (0) 481 / 85 55 - 300  
mobil: +49(0)177 / 87 31 62 1  
Fax: +49 (0) 481 / 85 55 - 301  
Email: [jensen@fh-westkueste.de](mailto:jensen@fh-westkueste.de)



# Von der Idee zum Programm

- **Problemanalyse**

- Was soll das Programm leisten? Z.B. eine Nullstelle finden, Ströme simulieren
- Was sind Nebenbedingungen? Z.B. ist die Funktion reell wertig? Wieviele Atome?

- **Methodenwahl**

- Schrittweises Zerlegen in Teilprobleme (Top-Down-Analyse) Z.B. Propagation, Kraftberechnung, Ausgabe
- Wahl von Datentypen und –strukturen Z.B. Listen oder Tupel? Wörterbuch?
- Wahl der Rechenstrukturen (Algorithmen) Z.B. Newton-Verfahren, Regula falsi
- Wahl der Programmiersprache

# Von der Idee zum Programm

- **Implementation und Dokumentation**

- Programmieren und gleichzeitig dokumentieren
- Kommentare und externe Dokumentation (z.B. Formeln)

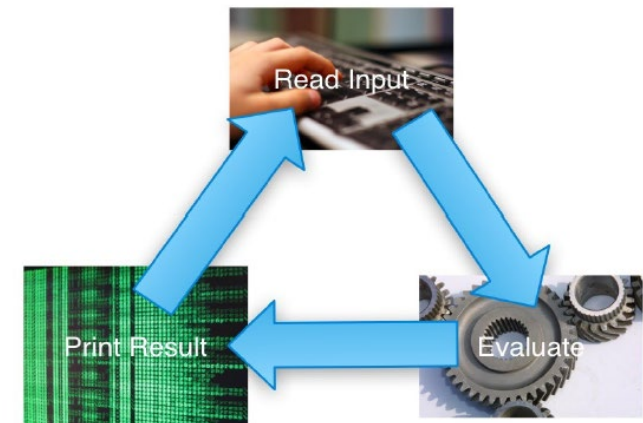
- **Testen auf Korrektheit**

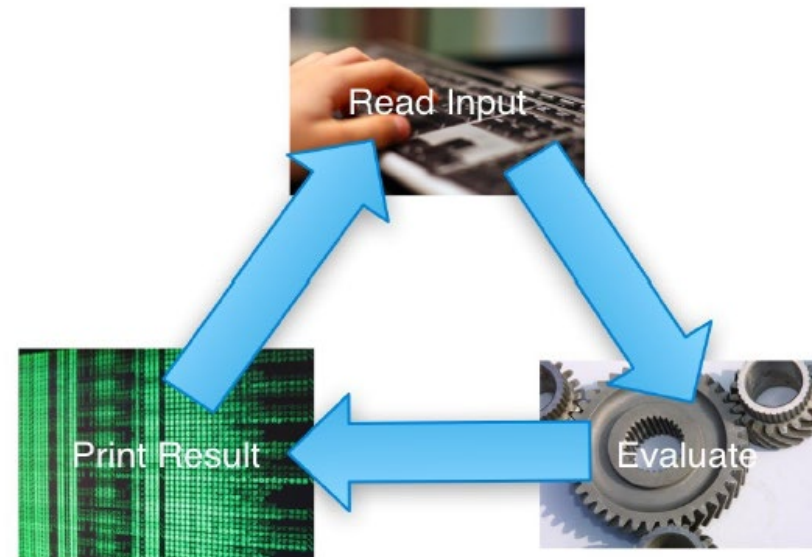
- Funktioniert das Programm bei erwünschter Eingabe? Z.B. findet es eine bekannte Lösung?
- Gibt es aussagekräftige Fehler bei falscher Eingabe? Z.B. vergessene Parameter, zu große Werte

- **Testen auf Effizienz**

- Wie lange braucht das Programm bei beliebigen Eingaben?
- Wieviel Speicher braucht es?

- **Meist wieder zurück zur Problemanalyse, weil man etwas vergessen hat .**







# python

- **Ursprünglich entwickelt von Guido van Rossum im Rahmen eines Forschungsprojekts am "Centrum voor Wiskunde en Informatica" in Amsterdam.**
- **Entwickelt seit 1989, erste öffentliche Version 1991.**
- **Meilensteine: Versionen 1.0.0 (1994), 1.5 (1998), 2.0 (2000), 3.0 (2008), zur Zeit 3.7.x**
- **Mittlerweile wird Python als Open-Source-Projekt von der Allgemeinheit weiterentwickelt, wobei ein innerer Kern die meiste Arbeit übernimmt. Guido van Rossum hat als "BDFL" (benevolent dictator for life, gütiger Diktator auf Lebenszeit) das letzte Wort.**

# Was ist Python

**Kurz: Python ist eine objektorientierte **Skriptsprache**.**

**Ausführlicher: Python ist eine. . .**

- objektorientierte,
- dynamisch getypte,
- **interpretierte** und
- interaktive
- High-Level-Programmiersprache.

# Zum Namen, Aussprache

Python ist nicht nach einem Reptil benannt, sondern nach **Monty Python's Flying Circus**, einer (hoffentlich!) bekannten englischen Komikertruppe aus den 1970ern.

Daher auch viele Namen von Tools rund um Python:

- IDLE
- Eric
- Bicycle Repair Man
- Grail

Wo andere Programmiersprachen die Variablen `foo` und `bar` verwenden, wählt man in Python gerne `spam` und `egg`.

# Python vs. C, C++, Java

**Python hat gegen über der C-Familie einen deutlich höheren Abstraktionsgrad („weiter weg von der Maschine“):**

- Automatische Speicherverwaltung
- Unbeschränkte Ganzzahlarithmetik
- Eingebaute komplexe Datentypen: list, dict, tuple
- Funktionen höherer Ordnung: map, filter, reduce
- Alles ist ein Objekt
- Alles ist dynamisch: Metaklassen und Metaprogrammierung

**Im Vergleich zu Sprachen aus der C-Familie sind Python-Programme:**

- kürzer
- lesbarer
- portabler
- **langsamer (!)**





# python

- **schnell zu erlernende, moderne Programmiersprache**
  - tut, was man erwartet
- **viele Standardfunktionen („all batteries included“)**
- **Bibliotheken für alle anderen Zwecke**
- **freie Software mit aktiver Gemeinde**
- **portabel, gibt es für fast jedes Betriebssystem**
- **entwickelt von Guido van Rossum, CWI, Amsterdam**

# Python

- **Informationen zu Python**
  - Aktuelle Versionen 3.7.2
  - 2.x ist noch weiter verbreitet
  - Diese Vorlesung behandelt daher noch 3.x
  - Aber längst nicht alles, was Python kann
- **Hilfe zu Python**
  - offizielle Homepage  
<http://www.python.org>
  - Einsteigerkurs „A Byte of Python“
    - <http://swaroopch.com/notes/Python> (englisch)
    - <http://abop-german.berlios.de> (deutsch)
  - mit Programmiererfahrung „Dive into Python“  
<http://diveintopython.net>

# Online-Dokumentation von Python

- **Einstiegspunkt:** <http://docs.python.org/>
- **Besonders wichtig/interessant:**
  - am Anfang das Tutorial(<http://docs.python.org/tutorial/index.html>)
  - im Programmieralltag die Library Reference(<http://docs.python.org/library/index.html>)

# IDEs für Python

- **IDLE ist die bei Python mitgelieferte IDE.**
- **Boa Constructor und PythonWin (nur Windows) sind weitere freie IDEs.**
- **Black Adder, Komodo und Wing IDE sind populäre kommerzielle Python-IDEs.**
- **Für Visual Studio (nur Windows) gibt es ein Python-Plugin. Für Eclipse ebenfalls (Pydev).**
- **Pycharm**

# Offizielle Internet-Ressourcen zu Python

## Offizielle Website:

- <http://www.python.org/>
- interessant dort zum Beispiel: Dokumentation, Python FAQs, Python Wiki, PEPs, Python Package Index

## Newsgroups:

- comp.lang.python.l
- comp.lang.python.announce

## Mailingliste:

- python-dev: siehe  
<http://mail.python.org/mailman/listinfo/python-dev>
- Newsgroup-Interface über <http://www.gmane.org/>

# Online-Bücher zu Python

- **Zwei kostenlose Online-Bücher zu Python:**
  - Mark Pilgrim: **Dive into Python**
    - <http://diveintopython.org/>
- **Allen Downey, Jeff Elkner und Chris Meyers:**  
**How to Think Like a Computer Scientist**
  - <http://www.greenteapress.com/thinkpython/thinkCSpy/>
  - Für Programmieranfänger.
  - Spätere Auflagen käuflich zu erwerben.

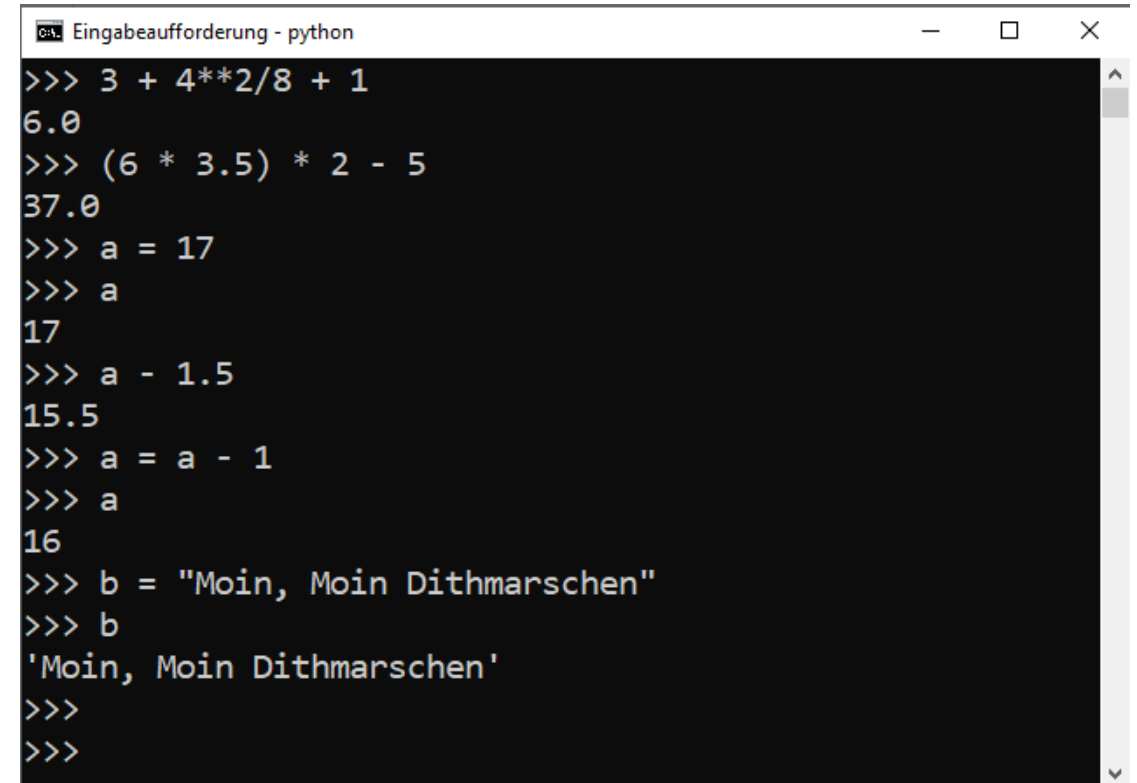


# Programmieren in Python

## Variablen, Zuweisungen und Ausdrücke

# Variablen, Zuweisungen und Ausdrücke

- Zuweisung: Variablenname = Ausdruck
- Nicht zu verwechseln mit mathematischem =
- Ausgabe des Ergebnisses nur interaktiv
- **print ()** gibt eine String-Repräsentation von Objekten aus
- Leerzeichen im Ausdruck egal (Aber nicht VOR dem Ausdruck)



```
Eingabeaufforderung - python
>>> 3 + 4*2/8 + 1
6.0
>>> (6 * 3.5) * 2 - 5
37.0
>>> a = 17
>>> a
17
>>> a - 1.5
15.5
>>> a = a - 1
>>> a
16
>>> b = "Moin, Moin Dithmarschen"
>>> b
'Moin, Moin Dithmarschen'
>>>
>>>
```



# Variablennamen

- a-z, A-Z, 0-9,
- Erstes Zeichen nicht 0-9 ist erlaubt, hat aber spezielle Bedeutung
- Variablennamen sollten aussagekräftig sein
- Bei mathematischen Ausdrücken entsprechend den mathematischen Variablen
- Ansonsten deskriptiv
- Einige Worte sind reserviert:  

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in , is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, with, while, yield
- Variablen in Python sind dynamisch typisiert



# Programmieren in Python

## Ausgaben und Zahlen

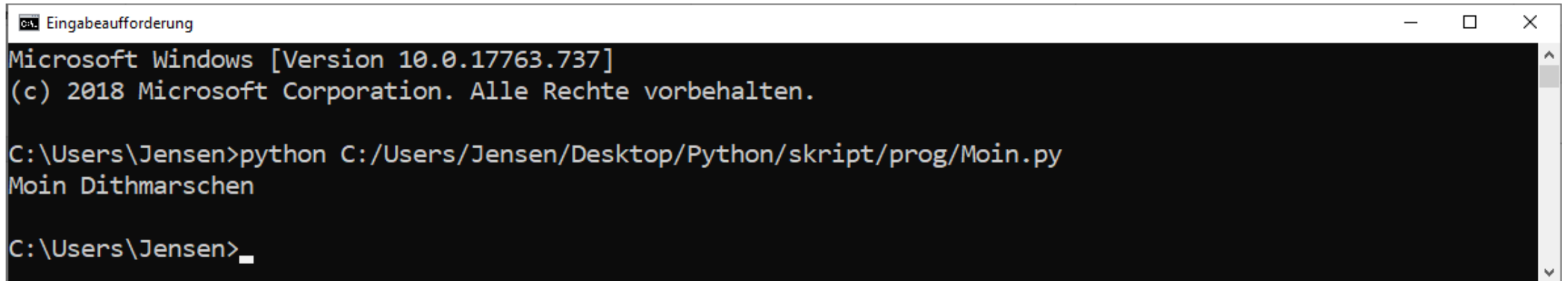
# Ausgaben und Zahlen

**In dieser Lektion geht es darum, ein erstes Gefühl für Python zu bekommen. Wir beschränken uns auf zwei einfache Bereiche:**

- **Ausgaben: print**
- **Eingaben: input**
- **Zahlen: int, long, float, complex**

# Interpreter & Runtime

Das Programm python (bzw. python.exe) kann sowohl verwendet werden, um **Python-Programme auszuführen**, als auch als interaktiver Interpreter benutzt werden:



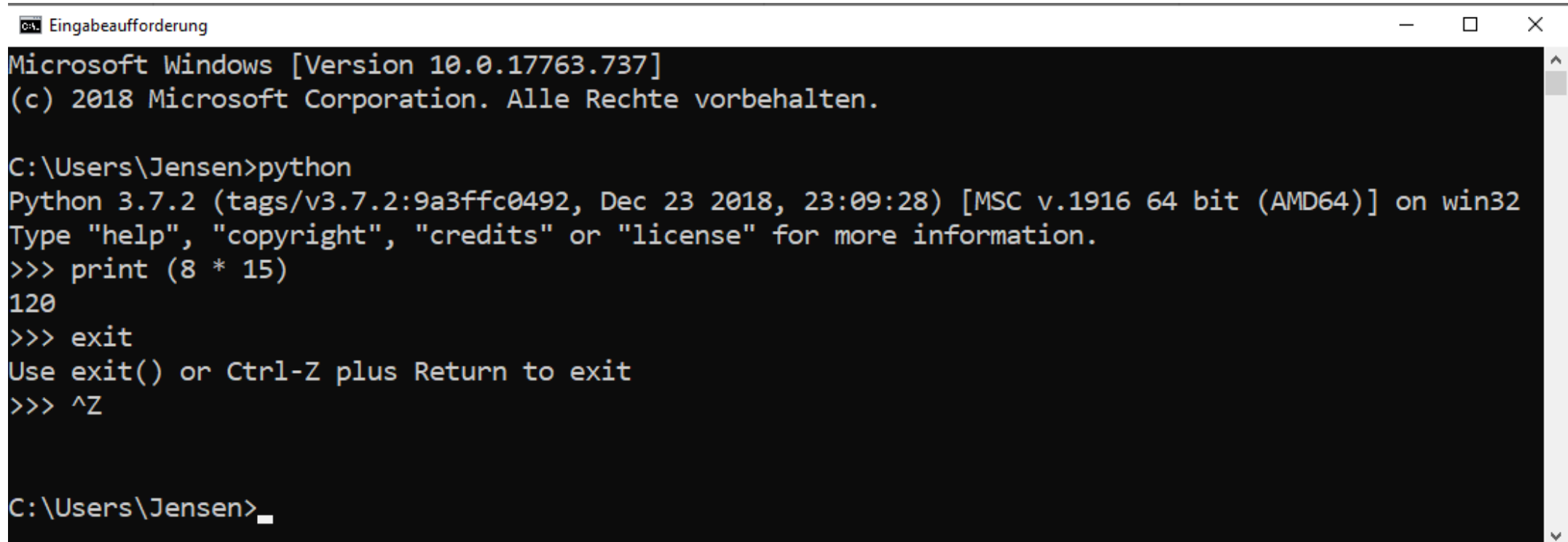
```
Microsoft Windows [Version 10.0.17763.737]
(c) 2018 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Jensen>python C:/Users/Jensen/Desktop/Python/skript/prog/Moin.py
Moin Dithmarschen

C:\Users\Jensen>_
```

# Interpreter & Runtime

Das Programm **python** (bzw. **python.exe**) kann sowohl verwendet werden, um Python-Programme auszuführen, als auch als **interaktiver Interpreter** benutzt werden:



```
C:\> Eingabeaufforderung

Microsoft Windows [Version 10.0.17763.737]
(c) 2018 Microsoft Corporation. Alle Rechte vorbehalten.

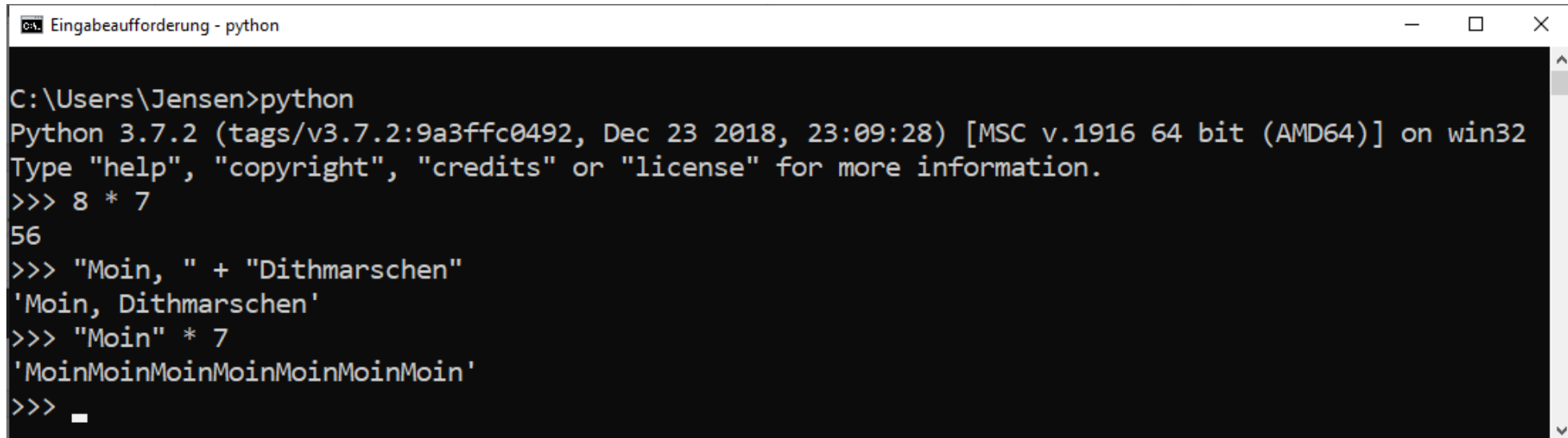
C:\Users\Jensen>python
Python 3.7.2 (tags/v3.7.2:9a3fffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print (8 * 15)
120
>>> exit
Use exit() or Ctrl-Z plus Return to exit
>>> ^Z

C:\Users\Jensen>_
```

# Ausgaben des Interpreters

Da die Beispiele in diesem Kapitel noch recht elementar sind, können wir sie allesamt im Interpreter behandeln.

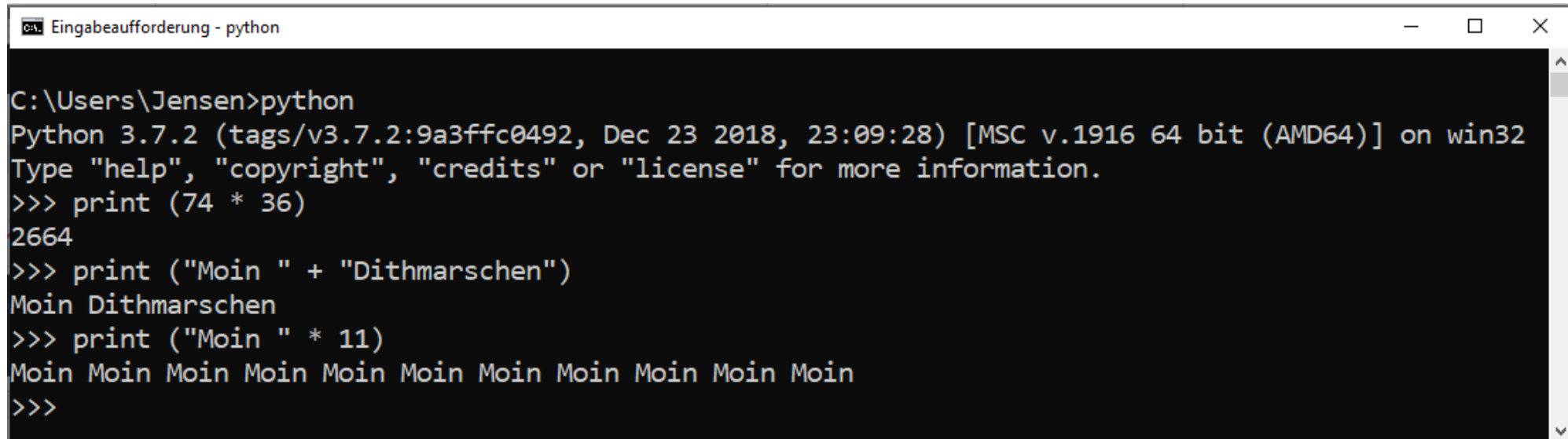
Um dem Interpreter eine Ausgabe zu entlocken, gibt es zwei Methoden. Zum einen kann man einfach einen Ausdruck eingeben, woraufhin der Interpreter dann den Ausdruck auswertet und das Ergebnis ausgibt:



```
C:\Users\Jensen>python
Python 3.7.2 (tags/v3.7.2:9a3fffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 8 * 7
56
>>> "Moin, " + "Dithmarschen"
'Moin, Dithmarschen'
>>> "Moin" * 7
'MoinMoinMoinMoinMoinMoinMoin'
>>> _
```

# Ausgaben des Interpreters (2)

Zum anderen kann man die `print`-Anweisung verwenden, um einen Ausdruck auszugeben:



```
C:\Users\Jensen>python
Python 3.7.2 (tags/v3.7.2:9a3fffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print (74 * 36)
2664
>>> print ("Moin " + "Dithmarschen")
Moin Dithmarschen
>>> print ("Moin " * 11)
Moin Moin Moin Moin Moin Moin Moin Moin Moin Moin Moin
>>>
```

**`print ()`** ist der übliche Weg, Ausgaben zu erzeugen und funktioniert daher auch in „richtigen“ Programmen, d.h. außerhalb des Interpreters, wo nackte Ausdrücke nur wegen ihrer Seiteneffekte verwendet werden.

# Ausgaben des Interpreters (3)

Es besteht ein kleiner Unterschied zwischen Ausdrücken und print-Anweisungen:

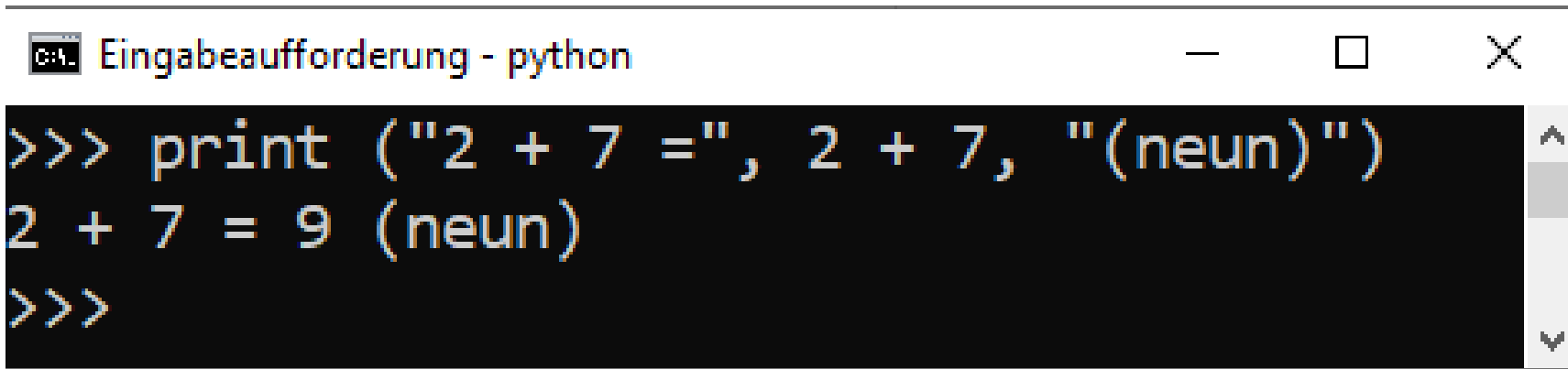
```
Eingabeaufforderung - python
>>> print ("Moin, Dithmarschen")
Moin, Dithmarschen
>>> print ("oben \n unten")
oben
    unten
>>> print (None)
None
>>> _
```

```
Eingabeaufforderung - python
>>> "Moin, Dithmarschen"
'Moin, Dithmarschen'
>>> "oben \n unten"
'oben \n unten'
>>> None
>>> _
```



# Etwas mehr zu print

Wir werden die Möglichkeiten von **print** später noch ausführlicher behandeln. Ein Detail soll aber schon jetzt erwähnt werden, da es für die Übungen wichtig ist:



```
>>> print ("2 + 7 =", 2 + 7, "(neun)")
2 + 7 = 9 (neun)
>>>
```

Man kann **print** mehrere Ausdrücke übergeben, indem man sie mit **Kommas** trennt.

Die Ausdrücke werden dann in derselben Zeile ausgegeben, und zwar durch Leerzeichen getrennt.

# Ausgaben und Zahlen

**In dieser Lektion geht es darum, ein erstes Gefühl für Python zu bekommen. Wir beschränken uns auf zwei einfache Bereiche:**

- **Ausgaben: print**
- **Eingaben: input**
- **Zahlen: int, long, float, complex, bool**

# Lesen von stdin

- **Ganze Zeile als String einlesen:**

```
x = raw_input("Prompt:")
```

- Das Prompt ist optional

- **Einzelne Zahl einlesen:**

```
x = input("prompt")
```

- Klappt nur, wenn der eingegebene String eine einzelne Zahl ist
- Genauer: Eingabe muß ein einzelner gültiger Python-Ausdruck sein

# Komplexere Eingaben lesen

- Mehrere Zahlen in einer Zeile einlesen:

**v1, v2, v3 = input("Geben Sie den Vektor ein: ")**

- Eingabe muß 1, 2, 3 sein!

- Zeilen lesen bis Input leer:

```
import sys

for line in sys.stdin :
    # do something with line
```

- **Von Tastatur einlesen:**

```
>>> ./program  
Geben Sie den Vektor ein: □
```

- **Mit I/O-Redirection aus File lesen:**

```
>>> ./program < vector.txt Geben Sie den Vektor ein:  
>>>
```

**Achtung: falls mit Schleife gelesen wird, muß man manchmal 2x Ctrl-D drücken zum Beenden der Eingabe (Bug in Python?)**

# Kommando-Zeilen-Argumente

- Bei Aufruf der Form

**% ./program arg1 arg2 arg3**

werden Kommandozeilenargumente im sog. Environment des neuen Prozesses gespeichert

- Zugriff über die argv-Variable:

```
import sys
for arg in sys.argv:
    print (arg)
```

- Oder:

**print (argv[0], argv[1])**

- argv[0] enthält Name des Skriptes

# Ausgaben und Zahlen

**In dieser Lektion geht es darum, ein erstes Gefühl für Python zu bekommen. Wir beschränken uns auf zwei einfache Bereiche:**

- **Ausgaben: print**
- **Eingaben: input**
- **Zahlen: int, long, float, complex**

Python kennt verschiedene Datentypen:

- **int** für ganze Zahlen im Bereich -2147483648 ... 2147483647 (mehr auf 64-Bit-Rechnern). **int** entspricht dem gleichnamigen Datentyp in C und Java.
- **long** für ganze Zahlen beliebiger Größe.
- **float** für Fließkommazahlen (entspricht double in C und Java).
- **complex** für komplexe (Fließkomma-) Zahlen.
- **bool** für
- **string** für
- **listen** für



# int und long (1)

- Rechnen, wie wir es gewohnt sind ( $12*34+567$ )
- In Python ist der Zahlenbereich beliebig groß

```
Eingabeaufforderung - python
>>> 12345**67
1348662605521577141133186026956371847027468714202501683631598110392597320303310238047815062074408337911
7936761486799988195646885805346351682922720871767924284000007972943491013427059666942168974390838351638
975049411082012506431283848686020210294600474298931658267974853515625
>>> type(12345)
<class 'int'>
>>> type(12345**67)
<class 'int'>
>>>
```

# int und long (2)

**int**-Konstanten schreibt man, wie man es erwartet:

10, -20, 300

**long**-Konstanten kann man analog zu C und Java schreiben:

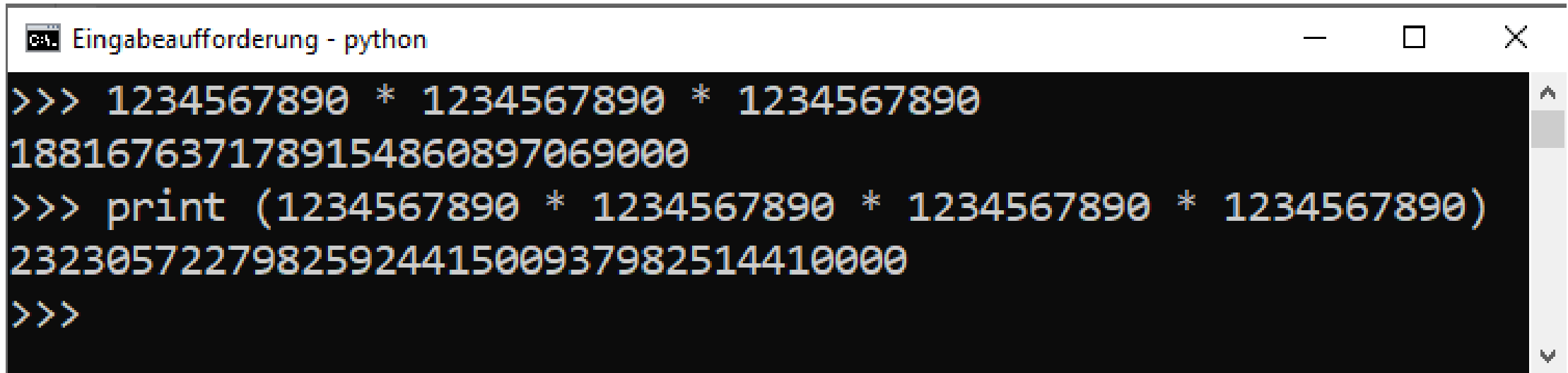
10L, -30L, 9876543210L

**Wie in C und Java werden Hexadezimal- und Oktalzahlen notiert:**

0x1ae00, 0xff, 0377

# int und long (3)

Man kann den Unterschied zwischen **int** und **long** normalerweise vergessen, da Python automatisch nach **long** umwandelt, wenn ein Rechenergebnis nicht mehr in ein **int** passt:



```
Eingabeaufforderung - python

>>> 1234567890 * 1234567890 * 1234567890
1881676371789154860897069000
>>> print (1234567890 * 1234567890 * 1234567890 * 1234567890)
2323057227982592441500937982514410000
>>>
```

# Rechnen mit int und long

**Python benutzt für Arithmetik weitgehend die von C und Java bekannten Symbole:**

- Grundrechenarten: +, -, \*, /
- Modulo: %
- Potenz (nicht in C/Java): \*\*
- Boole'sche Bitoperatoren: &, |, ^, ~
- Bit-Shift: <<, >>

# Beispiele Vorrang

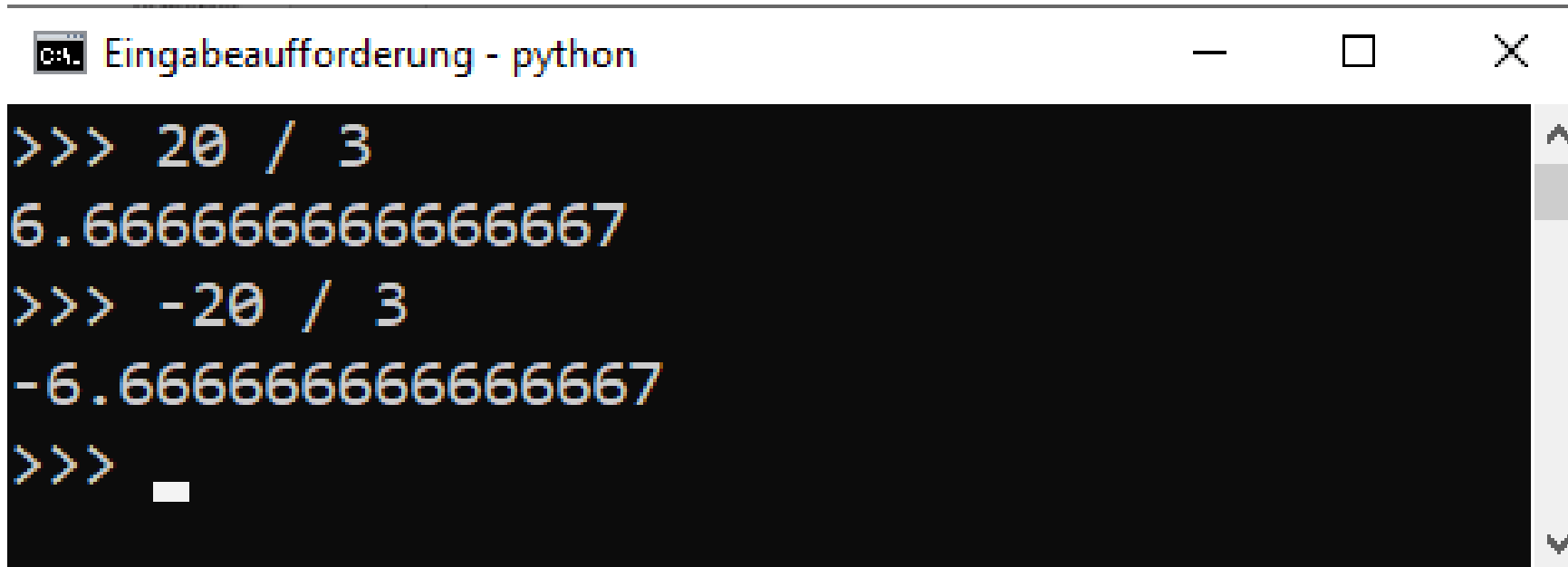
- **$5 * 3 + 1$**
- **$1 + 3 * 5$**
- **$1 + 16 \% 4$**
- **$1 + 17 \% 4$**
- **$2 * 16 \% 4$**
- **$15 // 3 / 2$**
- **$15 / 3 // 2$**

# Rechnen mit int und long: Beispiele

```
Eingabeaufforderung - python
>>> 27 * 16 + 63
495
>>> 13 % 8
5
>>> -2 % 8
6
>>> 11 ** 11
285311670611
>>> 1 << 40
1099511627776
>>> 1000 & 0xff00ff
232
>>> 21256 ^ (21256 & (21256 - 1))
8
>>> 
```

# Integer-Division: Mit oder ohne Rest? (1)


Bei der Division wird eine **float** Variable ausgegeben (Gerundet wie bei C++ wurde in früheren Versionen):



```
>>> 20 / 3
6.666666666666667
>>> -20 / 3
-6.666666666666667
>>> 
```

# Integer-Division: Mit oder ohne Rest? (2)

Für die Division mit Abrunden verwendet man den Operator **//**:



```
Eingabeaufforderung - python
>>> 20 // 3
6
>>> -20 // 3
-7
>>>
```



# Fließkommazahlen und komplexe Zahlen

**float**-Konstanten schreibt man wie in C und Java:

2.44, 1.0, 5., 1e+100

**complex**-Konstanten schreibt man als Summe von (optionalem) Realteil und Imaginärteil mit imaginärer Einheit **j**:

4+2j, 2.3+1j, 2j, 5.1+0j

**float** und **complex** unterstützen dieselben arithmetischen Operatoren wie die ganzzahligen Typen, aber (sinnvollerweise) keine Bitoperationen. Wir haben also:

Grundrechenarten: +, -, \*, /

Modulo: %

Potenz: \*\*

# Rechnen mit float

```

>>> 1.23 * 4.56
5.6088
>>> 17 / 2.0
8.5
>>> 23.1 % 2.7
1.5
>>> 1.5 ** 100
4.065611775352152e+17
>>> 10 ** 0.5
3.1622776601683795
>>> 4.23 ** 3.11
88.69896302282595
>>>

```

# Rechnen mit complex

```
Eingabeaufforderung - python

>>> 2+3j + 4-1j
(6+2j)
>>> 1+2j * 100
(1+200j)
>>> -1 ** 0.5
-1.0
>>> (-1) ** 0.5
(6.123233995736766e-17+1j)
>>> print ((-1) ** 0.5)
(6.123233995736766e-17+1j)
>>>
```

# Rechnen mit complex (2)

- Imaginärteil hat Zusatz j
- Basisoperationen normal anwendbar
- Für komplexere Operationen (z.B. sin) spezielle Bibliotheken
- Alles in Python ist ein Objekt
- Zugriff auf Objektmethoden: <Objektname>.<Methodenname>

```
Eingabeaufforderung - python
>>> 1 + 5j
(1+5j)
>>> complex (4,7)
(4+7j)
>>> d = 22 * ( _ ** 3 - 4j)
>>> d
(-11528-242j)
>>> d.real
-11528.0
>>> d.imag
-242.0
>>> _
```

# Numerische Typen - Vergleichsoperatoren

- `a == b` gibt `True` bei Gleichheit (= dient Zuweisung)

Eingabeaufforderung - python

```
>>> 1 == 2
False
>>> 1 == 1.0
True
>>> 1 == "1"
False
```

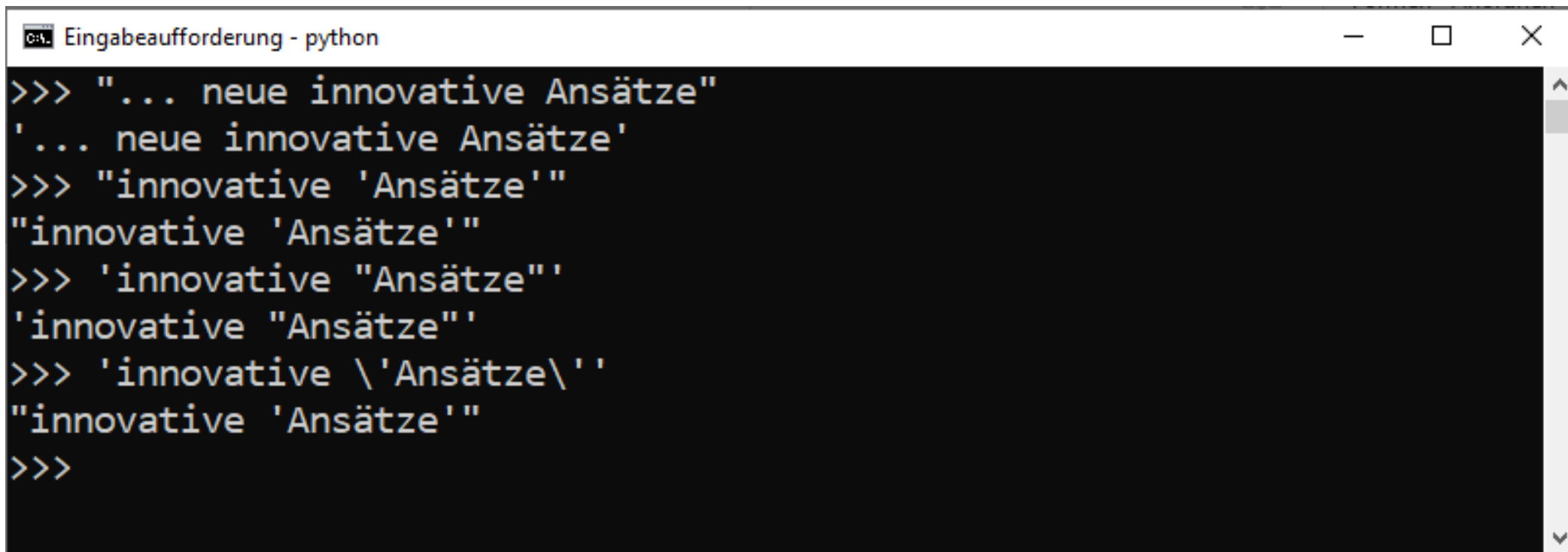
- weitere Operatoren: `!=`, `<`, `>`, `<=`, `>=`
- Verknüpfung mit `and` und `or`, Negation mit `not`
- Shifting: `<<`, `>>`
- Bitweise Operatoren: `&`, `|`, `^`, `~`

Eingabeaufforderung - python

```
>>> not (1 == 2 or 3 + 1 == 2 << 1)
False
>>> _
```

# Sequenzen – Strings(1)

- Sequenz von Zeichen (character)
- einzelne oder doppelte Anführungszeichen
- dreifache Anführungszeichen für mehrzeiligen String



```
Eingabeaufforderung - python
>>> "... neue innovative Ansätze"
'... neue innovative Ansätze'
>>> "innovative 'Ansätze'"
"innovative 'Ansätze'"
>>> 'innovative "Ansätze"'
'innovative "Ansätze"'
>>> 'innovative \'Ansätze\''
"innovative 'Ansätze'"
>>>
```

# Sequenzen – Strings(2)

- Verknüpfung von Strings

```
Auswählen Eingabeaufforderung - python

>>> s1 = "Wir brauchen "
>>> s2 = "neue innovative "
>>> s3 = "Ansätze"
>>> s = s1 + ' ' + s2 + ' ' + s3
>>> print(s)
Wir brauchen  neue innovative  Ansätze
>>>
```

# Sequenzen – Strings (3)

- Indizierung und Abschnitte

```
Eingabeaufforderung - python
>>> s = "1234567890"
>>> s[3], s[2:4], s[-2]
('4', '34', '9')
>>> s[6:], s[:3]
('7890', '123')
>>> s[0:-1:2]
'13579'
>>> s[2] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> len(s)
10
>>> _
```



# Sequenzen – Strings (4)

- **Replikation**



```
C:\> Eingabeaufforderung - python

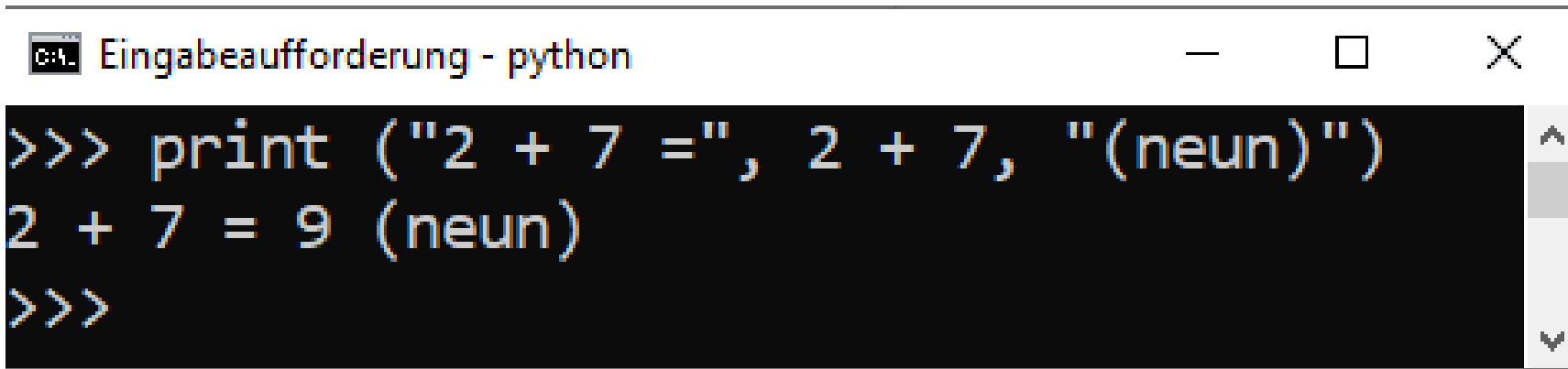
>>>
>>>
>>> "Moin! " * 3
'Moin! Moin! Moin! '
>>>
```

# Ausgewählte String-Methoden

```
s = „Wir brauchen neue innovative Ansätze“  
s. islower ()  
s. isupper ()  
s. startswith ("Wir")          # s. startswith ("Wir", 2)  
s. endswith ("Ansätze")  
s = s. strip ()  
s. upper ()  
s = s. lower ()  
s = s. center ( len (s )+4)      # zentrieren  
s. lstrip ()  
s. rstrip (" ")  
s = s. strip ()  
s. find ("brauchen")  
s. rfind ("innovative")
```

# Noch etwas mehr zu print(1)

Wir werden die Möglichkeiten von **print** später noch ausführlicher behandeln. Ein Detail soll aber schon jetzt erwähnt werden, da es für die Übungen wichtig ist:



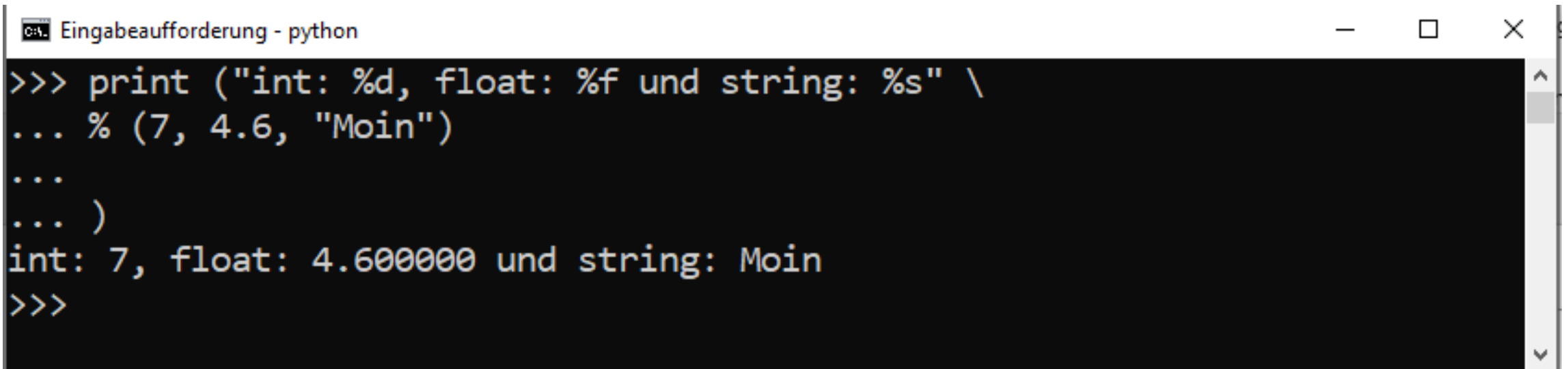
```
>>> print ("2 + 7 =", 2 + 7, "(neun)")
2 + 7 = 9 (neun)
>>>
```

Man kann **print** mehrere Ausdrücke übergeben, indem man sie mit **Kommas** trennt.

Die Ausdrücke werden dann in derselben Zeile ausgegeben, und zwar durch Leerzeichen getrennt.

# Noch etwas mehr zu print(2)

- Stringinterpolation gibt hohe Flexibilität
- Platzhalter mit % in einem String
- Nach dem String folgen ein % und ein Tupel mit Werten.



```
Eingabeaufforderung - python
>>> print ("int: %d, float: %f und string: %s" \
... % (7, 4.6, "Moin")
...
... )
int: 7, float: 4.600000 und string: Moin
>>>
```

# Konvertierungszeichen zur Stringinterpolation

- **%d** Dezimal Integer
- **%f** Gleitpunkt (float)
- **%e,%E** Gleitpunkt wissenschaftlich (m.ddde+xx)
- **%g,%G** kompakteste Gleitpunktdarstellung
- **%s** String
- **%c** einzelnes Zeichen
- **%o** Oktaldarstellung eines Integers
- **%x, %X** Hexadezimaldarstellung eines Integers
- **%%** Das Zeichen %

```
Eingabeaufforderung - python

>>> a = 7.357 ; b = 47
>>> print ("%f, %.2f, %8.1f" % (a, a, a))
7.357000, 7.36,      7.4
>>> print ("%o, %x" % (b, b))
57, 2f
>>> 
```

# Sequenzen - Tupel

- Sequenzen beliebiger Objekte
- Unveränderlich
- Indizierung und Ausschnitte wie bei Strings

```
>>> t = (1 , " zwei " , (3 ,4 ,5))
>>> t [0]
1
>>> t [ -1]
(3 ,4 ,5)
>>> t [1] = 2 # error
>>> len (t)
3
>>> t + (6 ,7)
(1 , ' zwei ' , (3 , 4, 5) , 6, 7)
```

- Minimum und Maximum mit **min(t)** und **max(t)**

# Sequenzen - Listen

- **Sequenzen beliebiger Objekte**
- **Veränderlich**
- **Indizierung und Ausschnitte wie bei Strings**

```
>>> l = [1, "zwei", (3,4,5)]  
>>> l[0]  
1  
>>> l[1] = 2; l  
[1, 2, (3,4,5)]  
>>> l + [6,7]  
[1, 2, (3, 4, 5), 6, 7]
```

- **Operationen auf Listen**

```
l = [0,1,2,3,4,5,6,7,8,9]  
l[2:4] = [10, 11]  
l[1:7:2] = [-1, -2, -3]  
del l[::2]
```

# Sequenzen - Listen (2)

- **Listen sind Objekte mit Methoden:**

```
>>> l = [0 ,1 ,2]
>>> l.append ("x")           # [0 , 1, 2, 'x ']
>>> l.extend ([5 ,6])        # [0 , 1, 2, 'x ', 5, 6]
>>> l.insert (2 , "x")       # [0 , 1, 'x ', 2, 'x ', 5, 6]
>>> l.count ("x")            # 2
>>> l.sort ()                # [0 , 1, 2, 5, 6, 'x ', 'x ']
>>> l.reverse ()             # [6, 5, 2, 1, 0, 'x ', 'x ']
>>> l.remove ("x")           # ['x ', 6, 5, 2, 1, 0]
>>> l.pop ()                 # ['x ', 6, 5, 2, 1]
0
```

Integerlisten erzeugen: range Funktion

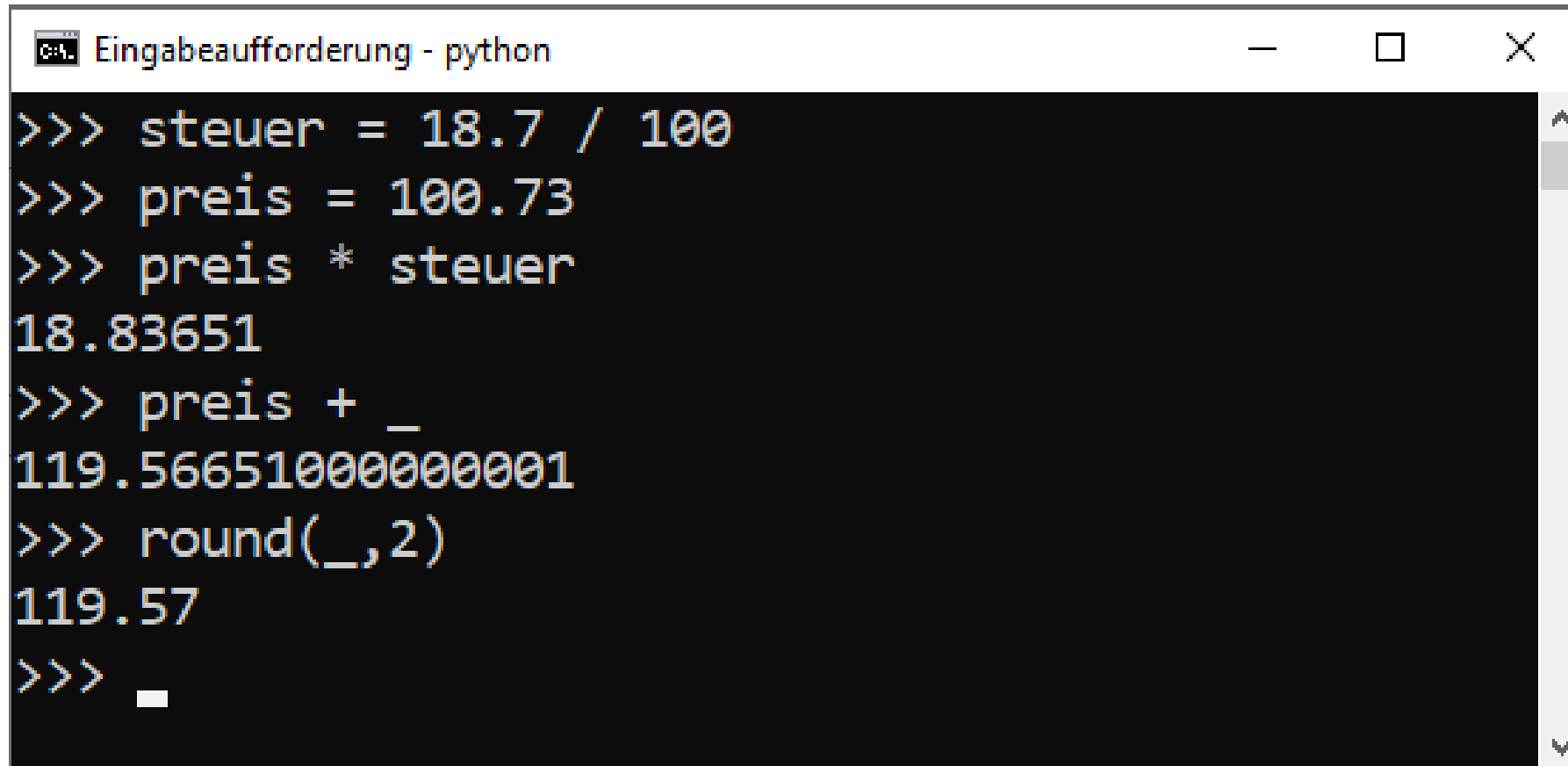
- **Integerlisten erzeugen: range Funktion**

```
>>> range (5)
[0 , 1, 2, 3, 4]
>>> range ( -10 , 5, 3)
[-10 , -7, -4, -1, 2]
```



# Spezielle Variablen

- Spezielle Variable „\_“ (nur im interaktiven Modus):



```
>>> steuer = 18.7 / 100
>>> preis = 100.73
>>> preis * steuer
18.83651
>>> preis + _
119.56651000000001
>>> round(_,2)
119.57
>>> _
```

**Ausdrücke mit gemischten Typen wie  $100 * (1+2j)$  verhalten sich so, wie man es erwarten würde. Die folgenden Bedingungen werden der Reihe nach geprüft, die erste zutreffende Regel gewinnt:**

- Ist einer der Operanden ein complex, so ist das Ergebnis ein complex.
- Ist einer der Operanden ein float, so ist das Ergebnis ein float.
- Bei der Division aus der Zukunft ist das Ergebnis ein float.
- Ist einer der Operanden ein long oder passt das Ergebnis der Operation nicht in ein int, so ist das Ergebnis ein long.
- Ansonsten ist das Ergebnis ein int.