KAPITEL 1 PYTHON

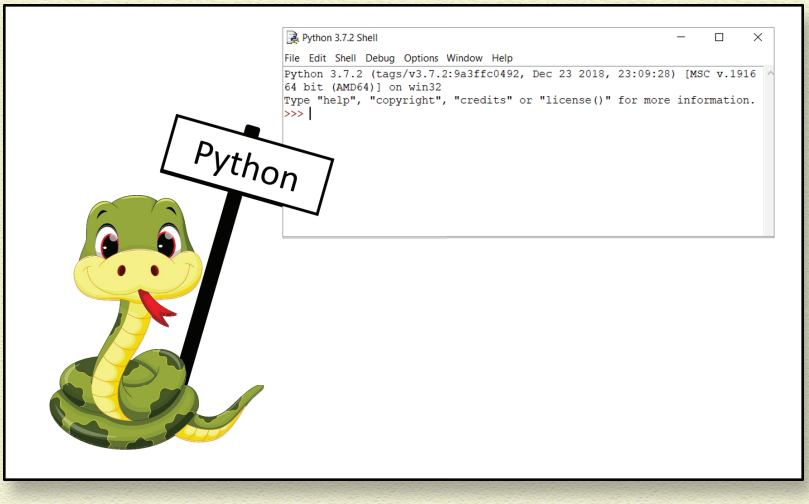


Illustration: © irwanjos – stock.adobe.com

WAS HABEN SCHLANGEN MIT COMPUTERN ZU TUN? Zu Beginn unserer KI-Reise wollen wir dir zeigen, was die Programmiersprache Python alles kann. Leider können wir dir nicht ausführlich zeigen, wie man in Python programmiert. Das Buch würde sonst aus allen Nähten platzen.

Daher konzentrieren wir uns auf die KI-Programme und gehen davon aus, dass du schon ein bisschen Erfahrung mit Python hast. Vielleicht hast du ja *Erste Schritte mit Python für Dummies Junior* oder *Python programmieren lernen für Dummies* gelesen. Oder du hast ein Python-Tutorial im Internet durchgearbeitet (zum Beispiel https://cscircles.cemc.uwaterloo.ca/using-website-de/).

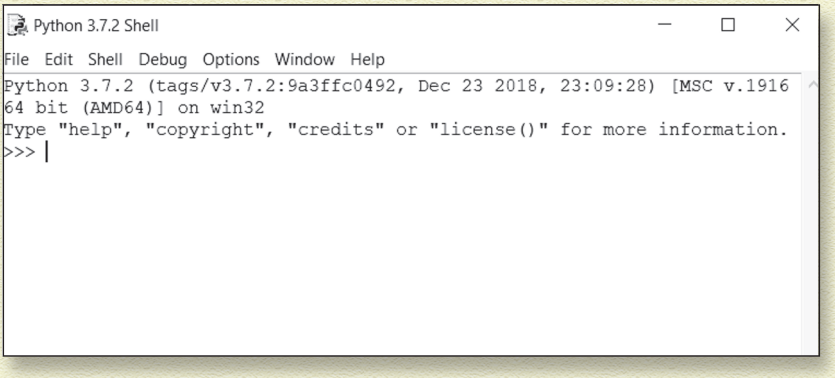
In diesem Kapitel werden wir wichtige Python-Grundlagen zusammenfassen, die wir in diesem Buch brauchen. Die meisten der Themen solltest du schon kennen. Wenn nicht, auch kein Problem. Dann lernst du in diesem Kapitel viel Neues.

DER PYTHON-EDITOR IDLE

Die neueste Python-Version erhältst du immer auf der offiziellen Internetseite von Python: https://python.org. Dort kannst du die neueste Version auswählen und für deinen Computer herunterladen.

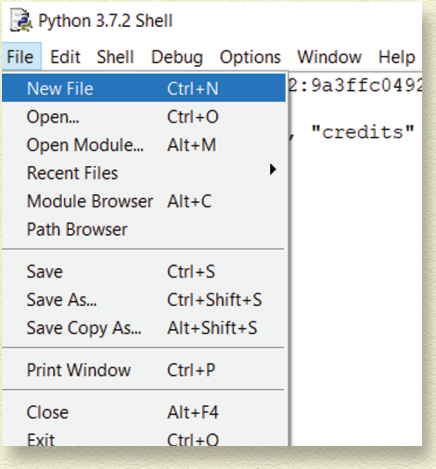
Der Download enthält immer auch die Python-Entwicklungsumgebung IDLE. Alle Beschreibungen in diesem Buch gehen davon aus, dass du IDLE verwendest. Wir wollen dir IDLE kurz vorstellen.

Wenn du IDLE startest, öffnet sich die Python-Shell. Die ersten Zeilen im Fenster enthalten Informationen zur verwendeten Python-Version und wie du verschiedene Hilfetexte aufrufen kannst.

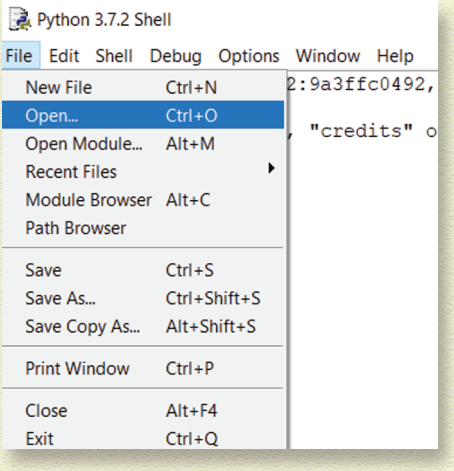


Wie du siehst, verwenden wir die Python-Version 3.7.2. Wichtig sind die drei Pfeile >>>. Sie zeigen an, dass du Befehle eingeben kannst, die dann sofort ausgeführt werden.

Ein neues, leeres Python-Dokument legst du am besten in IDLE an. Klicke in der Menüzeile auf File und wähle dann New File.

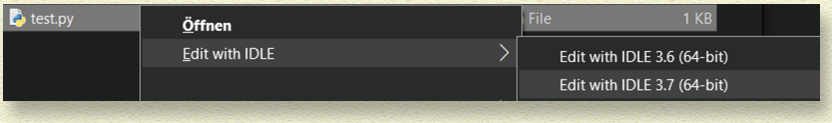


Wenn du ein Python-Dokument öffnen möchtest, klickst du in der Menüzeile auf File und wählst dann Open.



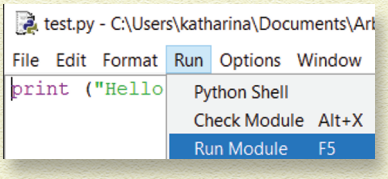
Ein neues Fenster öffnet sich, in dem du dann die Datei auswählen kannst. Schließlich klickst du auf Öffnen, um die Datei zu öffnen.

Alternativ kannst du die Datei auch in deinem Dateimanager (zum Beispiel Windows Explorer) suchen. Dort klickst du dann mit der rechten Maustaste auf den Dateinamen und auf Edit with IDLE. Dann wählst du Edit with IDLE 3.7.



Wenn du ein Python-Dokument speichern möchtest – egal, ob du ein geöffnetes Dokument bearbeitet hast oder ein neues Dokument angelegt hast –, klickst du in der Menüzeile auf File und dann auf Save (unter bisherigem Namen speichern) oder auf Save As (unter neuem Namen speichern).

Natürlich kannst du dein Python-Programm auch gleich in der Shell ausführen. Dazu klickst du in der Menüzeile auf Run und dann auf Run Module. Dann wird der Inhalt deines Dokuments ausgeführt.



Normalerweise wird in deinem Programm oder in der Shell jede Anweisung ausgeführt. Manchmal möchtest du aber etwas in dein Programm schreiben, das nicht ausgeführt wird, beispielsweise eine Erklärung der verwendeten Befehle. Man nennt das im Allgemeinen Kommentar.

Um in Python einen Kommentar zu schreiben, benutzt du das Raute-Zeichen #. Schreibe es einfach vor deine Anweisung und alles bis zum Ende dieser Zeile wird ignoriert.

```
>>> # Mein Kommentar

>>> a = a + 1 # a um eins erhöhen
```

EINFACHE DATENTYPEN UND VARIABLEN

Du kannst in Python ganz leicht rechnen: Gib einfach einen Rechenausdruck in der Shell (oder in deinem Programm) ein.

```
>>> 13 - (2/25 - 6) / 16
13.37
```

Python kann mit ganzen Zahlen (integer oder kurz `int`) und Kommazahlen (`float`) umgehen, beherrscht die Grundrechenarten +, -, *, /, beachtet die »Punkt- vor Strichrechnung«-Regel und versteht, was Klammern in Rechenausdrücken bedeuten.

Die Programmiersprache Python wird auf der ganzen Welt verwendet. Deshalb sind die Anweisungen in Englisch. Wir schreiben dir in diesem Kapitel immer die englischen Bezeichnungen in Klammern dazu, damit du leichter Informationen dazu im Internet finden kannst.

Eine andere Art von Daten, mit der Python umgehen kann, sind Zeichenketten aus Buchstaben, Ziffern und Sonderzeichen (string oder kurz `str`). Diese brauchst du oft, um Informationen auf dem Bildschirm auszugeben, zum Beispiel mit einer `print`-Anweisung.

```
>>> print("Willkommen! ")
Willkommen!
```

Zeichenketten schreibt man in Python innerhalb von Anführungszeichen. Man kann in Python auch ein bisschen mit Zeichenketten »rechnen«: Zwei Zeichenketten können addiert, also hintereinander gefügt werden, oder eine Zeichenkette kann multipliziert, also wiederholt werden.

```
>>> print("Herzlich " + "Willkommen! ")
Herzlich Willkommen!
>>> print(5 * "Hallo")
HalloHalloHalloHalloHallo
```

Schließlich gibt es in Python noch Wahrheitswerte (`bool`). Mit ihnen kann man ausdrücken, dass etwas zutrifft (`True`) oder nicht zutrifft (`False`).

Oftmals möchte man in Python einen Wert speichern, um ihn an anderer Stelle wieder zu verwenden. Man legt eine sogenannte *Variable* an. Um einer Variablen einen Wert zuzuweisen, verwenden wir das Gleichheitszeichen (=). Wenn der Wert der Variablen verwendet werden soll, schreiben wir einfach den Namen der Variablen.

```
>>> meine_Zahl = 3
>>> meine_Zeichenkette = "Hallo"
>>> 5*meine_Zahl
15
>>> print(meine_Zeichenkette)
Hallo
```

Variablennamen dürfen Buchstaben, Zahlen und Unterstriche enthalten, müssen aber mit einem Buchstaben oder einem Unterstrich beginnen. Leerzeichen sind in einem Variablennamen nicht erlaubt. Außerdem wird zwischen Groß- und Kleinschreibung unterschieden, `zahl` und `Zahl` sind zwei verschiedene Variablen.

Manche Computer haben Probleme mit Umlauten ä, ö, ü und auch mit ß. Daher sollte man diese Zeichen nicht in Variablennamen verwenden. Auch nicht als Großbuchstaben. Schreibe besser ae, oe, ue und ss.

LISTEN UND TUPEL

Gehören zwei Werte zusammen, kann man dafür in Python ein sogenanntes Paar verwenden. Beide Werte werden durch Klammern zusammengefasst und durch ein Komma getrennt.

```
>>> (1, 2)
(1, 2)
```

Es können auch mehr als zwei Werte zusammengefasst werden. Dazu werden alle Werte durch Kommas getrennt zwischen zwei Klammern notiert. Man spricht dann von einem Tupel.

Wenn man auf die Werte in einem Tupel zugreifen will, gibt es zwei Möglichkeiten:

Zum einen kann man die einzelnen Werte Variablen zuweisen. Anders als bei einer normalen Variablenzuweisung schreibt man dazu mehrere Variablennamen durch Kommas getrennt vor ein Gleichheitszeichen. Hinter dem Gleichheitszeichen steht dann das Tupel. Es müssen genau so viele Variablennamen vor dem Gleichheitszeichen stehen, wie Werte im Tupel sind. Du musst also wissen, wie viele Werte dein Tupel enthält.

```
>>> werte = ("a", "b", "c")
>>> wert1, wert2, wert3 = werte
>>> print(wert2)
b
```

Die zweite Möglichkeit, auf Werte in einem Tupel zuzugreifen, ist über die Position im Tupel. Dazu schreibst du einfach die Nummer der Position in eckigen Klammern direkt hinter das Tupel beziehungsweise die Variable, die das Tupel enthält. Aber Achtung: Python fängt bei 0 an zu zählen. Der erste Wert in einem Tupel hat also die Positionsnummer 0. Der zweite Wert die Positionsnummer 1 und so weiter.

```
>>> (1, 2, 3)[0]
1
```

Eine andere Art, Werte zusammenzufassen, ist in einer Liste (`list`). Hier werden alle Werte durch Kommas getrennt in eckigen Klammern notiert. Die Werte in einer Liste nennt man auch die Elemente der Liste.

Auf die Elemente der Liste kannst du genauso zugreifen wie auf die Werte in einem Tupel. Entweder weist du die Elemente der richtigen Anzahl Variablen zu oder du greifst über die Positionsnummer auf ein Element zu. Auch bei Listen beginnt die Positionsnummer bei 0.

```
>>> werte = ["a", "b", "c"]
>>> wert1, wert2, wert3 = werte
>>> print(wert2)
b
>>> print(werte[1])
b
```

Aus einer Liste kannst du auch eine Teilliste erzeugen. Dazu schreibst du hinter die Liste in eckigen Klammern die Positionsnummer des ersten Elements, das in der Teilliste sein soll, einen Doppelpunkt und das erste Element, das nicht mehr in der Teilliste sein soll. Wir zeigen dir das mal an einem einfachen Beispiel:

```
>>> zahlen = [1,2,3,4,5]
>>> zahlen[2:4]
[3, 4]
```

Was ist passiert? Zuerst erzeugen wir eine Liste mit den Zahlen 1 bis 5 und weisen sie der Variablen zu. Dann erzeugen wir daraus eine Teilliste mit den Zahlen 3 und 4. Das dritte Element der Liste (Positionsnummer 2) wird zum ersten Element der Teilliste. Danach kommt das vierte Element (Positionsnummer 3). Das fünfte und letzte Element (Positionsnummer 4) kommt nicht mehr in die Teilliste.

Eine nützliche Anweisung für Listen ist `len` (kurz für Englisch *length*). Die Anweisung gibt die Länge der Liste, also die Anzahl ihrer Elemente zurück.

```
>>> len(["a", "b", "c"])
3
```

Bisher haben wir nur Listen und Tupel gesehen, die entweder nur Zahlen- oder nur Zeichenketten-Werte enthalten haben. Listen und Tupel können aber beliebige Werte enthalten – auch unterschiedliche Arten gleichzeitig. Listen können also beispielsweise auch Tupel als Elemente haben. Das wird dir in diesem Buch noch mehrfach begegnen.

```
>>> liste1 = ["Hallo", 2, 3, True]
>>> liste2 = [ [1,2,3], ["a", "b", "c"], (5, 6) ]
```

BEDINGTE ANWEISUNGEN

Beim Programmieren ist es häufig notwendig, eine Anweisung nur dann auszuführen, wenn eine bestimmte Bedingung zutrifft. Man nennt dies eine *bedingte Anweisung*. In Python heißt diese Anweisung `if`.

```
if Bedingung:
    Dann-Teil
```

Wenn die Bedingung im Wenn-Teil (englisch `if`) zutrifft, führt Python die Anweisung im Dann-Teil aus. Du musst den Dann-Teil immer einrücken. Gib dafür einfach mehrere Leerzeichen ein. Der Dann-Teil kann auch aus mehreren Zeilen bestehen. Die musst du dann gleich weit einrücken. Nur so weiß Python, welche Zeilen alle zum Dann-Teil gehören sollen.

Die Bedingung besteht meist aus einem Vergleich. Python bietet hierfür die Vergleichsoperatoren `<` (kleiner als), `>` (größer als), `==` (gleich), `<=` (kleiner oder gleich) und `>=` (größer oder gleich). Für die Gleichheit verwenden wir in Python zwei Gleichheitszeichen, um den Vergleich von einer Zuweisung zu einer Variablen zu unterscheiden.

Das folgende Beispiel gibt die Zahl 1 in Buchstaben aus, wenn die Variable `zahl` den Wert 1 enthält.

```
>>> if zahl == 1:
        print("eins")
```

Die Bedingung kann auch komplizierter sein. Sie kann aus verschiedenen Teilbedingungen bestehen. Zwei Teilbedingungen müssen entweder beide zutreffen (`and`) oder mindestens eine von beiden muss zutreffen (`or`).

Wenn du mehr als zwei Teilbedingungen verwenden möchtest, solltest du sie durch Klammern gruppieren. Es ist auch möglich, eine Anweisung auszuführen, wenn eine (Teil-)Bedingung nicht erfüllt ist (`not`).

Die folgende Anweisung gibt beispielsweise `ja` aus, wenn der Wert von `zahl` zwischen 5 und 10 liegt oder `andere_zahl` nicht den gleichen Wert wie `zahl` enthält.

```
>>> if (zahl >= 5 and zahl <= 10) or \
        not andere_zahl == zahl:
        print("ja")
```

Anstelle eines Vergleichs kann eine (Teil-)Bedingung auch einen Wahrheitswert, also `True` oder `False`, enthalten.

Die `if`-Anweisung kann durch einen Sonst-Teil (englisch `else`) erweitert werden. Der Teil wird nur ausgeführt, wenn die Bedingung nicht zutrifft.

```
if Bedingung:
    Dann-Teil
else:
    Sonst-Teil
```

Zusätzlich kann eine `if`-Anweisung noch einen oder mehrere andernfalls-Teile enthalten (`elif`, eine Kombination aus `else` und `if`). Diese haben jeweils ihre eigene Bedingung und werden nur dann ausgeführt, wenn bisher keine andere Bedingung zugetroffen hat.

```
elif AndereBedingung:
    Andernfalls-Teil
```

Das folgende Beispiel gibt den Wert der Variablen `zahl` in Buchstaben aus. Zumindest falls der Wert zwischen 1 und 3 liegt.

```
>>> if zahl == 1:
        print("eins")
    elif zahl == 2:
        print("zwei")
    elif zahl == 3:
        print("drei")
    else:
        print("mehr als drei")
```

ANWEISUNGEN ÜBER MEHRERE ZEILEN

Du kannst in Python Anweisungen nicht einfach so über mehrere Zeilen verteilen. Python nimmt nämlich nur bei einer geöffneten Klammer automatisch an, dass eine Anweisung auf der nächsten Zeile weitergehen soll.

Folgende Varianten sind in Ordnung (wenn auch nicht sehr übersichtlich):

```
>>> werte = [1,2,
        3,4]

>>> tupel = (1,2,3,
        4,5)

>>> print("Eine Nachricht",
        "mit mehreren",
        "Teilen.")
```

Die folgenden Zeilen hingegen führen zu einer Fehlermeldung (`SyntaxError`):

```
>>> wert = 1 +
        2*5
        + 4
```

Python weiß einfach nicht, dass deine Anweisung über drei Zeilen verteilt ist. Das kannst du Python aber sagen. Wenn eine Zeile mit einem Backslash-Zeichen (\) endet, nimmt Python die nächste Zeile noch mit zu deiner Anweisung hinzu.

```
>>> wert = 1 + \
        2*5 \
        + 4
```

Jetzt ist in der Variablen `wert` die Zahl 15 gespeichert (1 + 2 * 5 + 4).

SCHLEIFEN

Python bietet die Möglichkeit, Anweisungen mehrfach hintereinander auszuführen. Hierzu verwendet man eine sogenannte Schleife. Eine Schleifenart ist die sogenannte `while`-Schleife. Sie ist der bedingten Anweisung sehr ähnlich, weshalb sie auch *bedingte Schleife* genannt wird.

```
while Bedingung:
    Dann-Teil
```

Anders als bei der bedingten Anweisung wird der Dann-Teil nicht nur einmalig ausgeführt, wenn die Bedingung erfüllt ist, sondern immer wieder, solange die Bedingung erfüllt ist. Auch bei der `while`-Schleife musst du den Dann-Teil durch mehrere Leerzeichen einrücken, damit Python weiß, welche Zeilen alle zum Dann-Teil gehören.

```
>>> x = 2
>>> while x < 100:
        print(x)
        x = 3 * x + 1

2
7
22
67
```

Die `while`-Schleife führt den Dann-Teil so lange aus, wie die Bedingung erfüllt ist. Wenn das Ende des Dann-Teils erreicht ist, wird erneut überprüft, ob die Bedingung erfüllt ist. Falls ja, wird der Dann-Teil noch einmal ausgeführt.

Es ist also wichtig, dass im Dann-Teil eine Variable, die in der Bedingung verwendet wird, geändert wird oder – durch eine bedingte Anweisung – geändert werden könnte. Ansonsten wird die Bedingung immer zutreffen und der Dann-Teil wird immer wieder ausgeführt. Du musst also endlos warten, bis dein Programm fertig ist.

Neben der `while`-Schleife gibt es in Python noch eine zweite Schleifenart, die `for`-Schleife.

```
>>> for meine_zahl in range(3):
        print("Aktuell: ", meine_zahl)

Aktuell:  0
Aktuell:  1
Aktuell:  2
```

Denke daran, den Code in der `for`-Schleife gleichmäßig einzurücken, damit Python erkennt, was alles innerhalb der Schleife passieren soll.

Mithilfe einer `for`-Schleife werden Anweisungen mehrfach ausgeführt. Du legst mit einer Variablen und einem sogenannten Wertebereich fest, wie oft die Anweisungen ausgeführt werden sollen. Der Variablen werden nacheinander alle Werte aus dem Wertebereich zugewiesen. Die Variable nennt man daher auch Zähler oder Zählervariable.

In unserem Beispiel heißt der Zähler `meine_zahl`, du kannst diesen Namen aber beliebig ändern. Die Anweisung `range(3)` legt fest, dass der Zähler mit ganzen Zahlen belegt werden soll. Und zwar beginnend bei der Null aufsteigend, solange die Zahl kleiner 3 ist. Man kann als Wertebereich auch eine Liste verwenden.

```
>>> for name in ["Thomas", "Martina", "Luisa"]:
        print("Hallo", name)

Hallo Thomas
Hallo Martina
Hallo Luisa
```

Du kannst auch eine `for`-Schleife mit mehreren Zählervariablen verwenden, die gleichzeitig verändert werden. Als Wertebereich musst du dann eine Liste von Tupeln verwenden. Achte darauf, dass alle Tupel die gleiche Anzahl an Werten haben.

```
>>> for x, y in [(2,3), (4,1), (5,2)]:
        print(x*y)

6
4
10
```

Diese Art von `for`-Schleife greift auf die Elemente eines Tupels so zu, wie du es weiter oben gelernt hast. Unser Beispiel ist deshalb nur eine verkürzte Variante von:

```
>>> for element in [(2,3), (4,1), (5,2)]:
        x, y = element
        print(x*y)
```

FUNKTIONEN

Du hast nun gelernt, wie man den gleichen Programmteil mehrfach hintereinander ausführen kann. Was macht man aber, wenn man denselben Programmteil an mehreren Stellen ausführen möchte? Man nutzt Funktionen. Eine *Funktion* gibt einem Programmteil einen Namen. Diesen kann man dann später verwenden, um den Programmteil auszuführen. Man nennt dies »die Funktion aufrufen«. Bevor du eine Funktion aufrufen kannst, musst du sie definieren (define oder kurz `def`):

```
>>> def rechne():
        print(5/3 + 2*4)
```

Wir haben uns hier eine Funktion mit dem Namen `rechne` definiert. Immer wenn wir jetzt `rechne()` ausführen, wird die Zahl 9.666666666666666 ausgegeben – das Ergebnis von `5/3 + 2*4`. Für die Namen von Funktionen gelten dieselben Regeln wie für die Namen von Variablen.

Unsere Beispielfunktion ist bisher wenig nützlich – es wird ja immer die gleiche Zahl ausgegeben. Viel nützlicher wäre die Funktion, wenn man ihr sagen könnte, mit welchen Zahlen sie rechnen soll. Das kann man mit sogenannten Parametern beim Funktionsaufruf machen. *Parameter* sind Werte, die in die Klammern hinter dem Funktionsnamen geschrieben werden. Damit die Funktion dann auch weiß, was sie mit den Parametern machen soll, musst du schon bei der Definition der Funktion entsprechende Variablen vorsehen.

```
>>> def rechne_neu(x,y):
        print(x/3 + 2*y)
```

Wir haben uns also eine neue Funktion `rechne_neu` definiert, die zwei Parameter erwartet. Immer wenn wir die Funktion aufrufen, müssen wir genau zwei Werte in die Klammern nach dem Funktionsnamen schreiben. Mit diesen Werten werden dann die Variablen `x` und `y` belegt.

```
>>> rechne_neu(5,4)
9.666666666666666
>>> rechne_neu(2,3)
6.666666666666667
>>> rechne_neu(9,1)
5.0
```

Unsere Funktion berechnet immer ein Ergebnis und schreibt dies in die Shell. Wir können mit dem Ergebnis aber nicht weiter rechnen! Wenn wir `rechne_neu(9,1) - 1` in der Shell eingeben, erhalten wir eine Fehlermeldung, nicht das erhoffte Ergebnis 4.0.

Um mit dem Ergebnis einer Funktion weiter arbeiten zu können, müssen wir das Ergebnis der Funktion zurückgeben. Hierfür gibt es die `return`-Anweisung.

```
>>> def plus_2(x):
        return x+2
```

Die Funktion `plus_2` addiert immer 2 zu ihrem Parameter und gibt das Ergebnis zurück. Wir können den Funktionsaufruf daher wie eine Variable verwenden:

```
>>> print(plus_2(4))
6
>>> print(plus_2(4) + 2)
8
>>> print(plus_2(plus_2(4)))
8
```

Die `return`-Anweisung arbeitet nicht nur mit Zahlen, sondern mit beliebigen Arten von Daten, zum Beispiel mit Zeichenketten.

```
>>> def hallo(name):
        return "Hallo " + name + "!"

>>> print(hallo("Julia"))
Hallo Julia!
```

Eine `return`-Anweisung beendet immer die aktuelle Funktion. Egal, was danach noch an Anweisungen steht oder ob die Anweisung innerhalb einer Schleife ist, die Funktion ist nach Ausführung der `return`-Anweisung zu Ende.

```
>>> def so_nicht(x):
        return 2*x
        print("Parameter:",x)

>>> print(so_nicht(3))
6
```

MODULE

Du weißt jetzt, wie du Funktionen schreiben kannst. Aber du musst nicht jedes Mal das Rad neu erfinden. Für viele einfache Aufgaben hat bereits jemand eine Funktion geschrieben. Deine

Python-Installation enthält schon einige Sammlungen mit solchen nützlichen Funktionen. So eine Sammlung nennt man *Modul*.

Wenn du die Funktionen in einem Modul nutzen willst, musst du das Modul zuerst in deinem Programm oder in der Shell bekannt machen. Man sagt dazu auch, »das Modul importieren« (`import`).

Beispielsweise könntest du das Modul `turtle` in der Shell importieren. Mithilfe dieses Moduls kannst du auf deinem Bildschirm zeichnen.

```
>>> import turtle
```

Wenn du nach dem Importieren die Funktionen aus dem Modul verwenden willst, schreibst du den Namen des Moduls, gefolgt von einem Punkt, gefolgt von dem Namen der Funktion.

```
>>> turtle.forward(50)
>>> turtle.left(120)
>>> turtle.forward(50)
>>> turtle.left(120)
>>> turtle.forward(50)
```

Du hast gerade ein kleines Dreieck auf dem Bildschirm gezeichnet.

Manchmal ist es zu viel Schreibarbeit, immer den Namen des Moduls zu wiederholen. Es gibt deshalb auch die Möglichkeit, direkt auf die Funktionen in einem Modul zuzugreifen.

Wir zeigen dir das am Beispiel des Moduls `random`, das Funktionen für Zufallszahlen enthält.

```
>>> from random import *
>>> randint(1,10)
1
>>> randint(1,10)
8
```

Wir haben zwei zufällige ganze Zahlen (englisch *random integer* oder kurz `randint`) zwischen 1 und 10 erzeugt. Bei dir werden vermutlich andere Zahlen herauskommen.

Wir werden in diesem Buch immer über den Namen des Moduls auf Funktionen in Modulen zugreifen. Dann kann man nämlich immer sehen, woher die Funktion kommt.

KLASSEN

Ähnlich wie Module Funktionen zusammenfassen, fassen *Klassen* Funktionen und Variablen zusammen. Zum Beispiel könntest du eine Klasse `Person` schreiben, die Speicherplatz für den Namen und das Alter einer Person vorsieht. Wenn du dann die Daten für eine konkrete Person speichern willst, erzeugst du ein *Objekt* der Klasse.

Das Objekt hat sogenannte *Attribute* (Name und Alter) und *Methoden*, das sind die Funktionen der Klasse. Wir können dir hier keine umfassende Einführung zu Klassen und Objekten – der sogenannten objektorientierten Programmierung – geben. Wir geben dir hier nur einen kurzen Überblick, wie wir Klassen in diesem Buch verwenden.

Wenn du mehr über Klassen wissen möchtest, könnte das Buch Python. Der Sprachkurs für Einsteiger und Individualisten *etwas für dich sein.*

Um ein Objekt einer Klasse zu erzeugen, verwendest du den Namen der Klasse wie einen Funktionsaufruf. Je nach Klasse erwartet die Funktion möglicherweise Parameter.

Nehmen wir an, wir möchten ein Objekt der Klasse `Person` erzeugen, wobei wir den Namen und das Alter der Person als Parameter übergeben müssen.

```
>>> jemand = Person("Sarah", 13)
```

Wir haben also ein Objekt der Klasse `Person` erzeugt und in der Variablen `jemand` gespeichert. Um nun auf ein Attribut oder eine Methode des Objekts zuzugreifen, nutzt du – wie bei Modulen – den Punkt als Namenstrenner.

```
>>> print(jemand.alter)
13
>>> jemand.hallo()
Hallo Sarah
```

Wir haben dir hier die Idee von Klassen und Objekten nur sehr kurz vorgestellt. In Python sind eigentlich alle Daten Objekte verschiedener Klassen. So gibt es die Klasse `list`*, die Listen darstellt, und [1,2,3] ist eigentlich ein Objekt dieser Klasse. Diese Klasse hat natürlich auch Methoden. So kann man beispielsweise mittels* `[1,2,3].append(4)` *die Zahl 4 als neues Element am Ende der Liste einfügen.*