Auszug aus: https://py-tutorial-de.readthedocs.io/de/python-3.3/introduction.html#
Sie steht unter der Apache License 2.0
Dies ist die Übersetzung des offiziellen Python-Tutorials der Version 3.3.
Hier finden Sie das Python-Tutorial der aktuellen Version:
https://docs.python.org/3/tutorial/index.html

3. Eine informelle Einführung in Python

Zum Ausprobieren der folgenden Beispiele muss alles eingetippt werden, was auf die Eingabeaufforderung (>>> oder . . .) folgt. Zeilen die nicht mit einer solchen Eingabeaufforderung beginnen, sind Ausgaben des Interpreters. Steht die sekundäre Eingabeaufforderung . . . allein in einer Zeile, dann muss eine Leerzeile eingegeben werden. Dadurch werden mehrzeilige Befehle abgeschlossen.

Viele Beispiele in diesem Tutorial enthalten Kommentare - auch solche, die im interaktiven Modus eingegeben werden. Kommentare beginnen in Python mit einer Raute # und gelten bis zum Ende der physikalischen Zeile. Ein Kommentar kann am Anfang einer Zeile beginnen oder im weiteren Verlauf einer Zeile, allerdings nicht innerhalb eines Zeichenkettenliterals. Eine Raute innerhalb eines Zeichenkettenliterals ist einfach nur eine Raute. Da Kommentare dazu dienen, Code zu erklären und von Python nicht interpretiert werden, können sie beim Abtippen weggelassen werden.

Ein paar Beispiele:

```
# Das ist der erste Kommentar
spam = 1                    # und dies ist der zweite Kommentar
                            # ... und jetzt ein dritter!
string = "# Dies ist kein Kommentar."
```

3.1. Benutzung von Python als Taschenrechner

Wir wollen ein paar einfache Python-Befehle ausprobieren: Starte den Interpreter und warte auf die primäre Eingabeaufforderung, >>>.

3.1.1. Zahlen

Der Interpreter kann wie ein Taschenrechner eingesetzt werden: Man kann einen Ausdruck eingeben und der Interpreter berechnet das Ergebnis. Die Syntax für solche Ausdrücke ist einfach: Die Operatoren +, -, * und / wirken genauso wie in den meisten anderen Sprachen (beispielsweise Pascal oder C); Klammern können zur Gruppierung benutzt werden. Zum Beispiel:

```
>>> 2 + 2
4
>>> # Dies ist ein Kommentar
... 2 + 2
4
>>> 2 + 2  # und dies ist ein Kommentar in derselben Zeile wie Code
4
>>> (50 - 5 * 6) / 4
5.0
>>> 8 / 5 # Brüche gehen nicht verloren, wenn man Ganzzahlen teilt
1.6
```

Anmerkung: Möglicherweise liefert die letzte Berechnung bei dir nicht genau das gleiche Ergebnis, weil sich Ergebnisse von Fließkommaberechnungen von Computer zu Computer unterscheiden können. Im weiteren Verlauf wird noch darauf eingegangen, wie man die Darstellung bei der Ausgabe von Fließkommazahlen festlegen kann. Siehe Fließkomma-Arithmetik: Probleme und Einschränkungen für eine ausführliche Diskussion von einigen Feinheiten von Fließkommazahlen und deren Repräsentation.

Um eine Ganzzahldivision auszuführen, die ein ganzzahliges Ergebnis liefert und den Bruchteil des Ergebnisses vernachlässigt, gibt es den Operator `//`:

```
>>> # Ganzzahldivision gibt ein abgerundetes Ergebnis zurück:
... 7 // 3
2
>>> 7 // -3
-3
```

Das Gleichheitszeichen (`'='`) wird benutzt um einer Variablen einen Wert zuzuweisen. Danach wird kein Ergebnis vor der nächsten interaktiven Eingabeaufforderung angezeigt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Ein Wert kann mehreren Variablen gleichzeitig zugewiesen werden:

```
>>> x = y = z = 0  # Null für x, y und z
>>> x
0
>>> y
0
>>> z
0
```

Variablen müssen “definiert” sein, bevor sie benutzt werden können, sonst tritt ein Fehler auf. Diese Definition geschieht durch eine Zuweisung:

```
>>> # Versuche eine undefinierte Variable abzurufen
... n
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python bietet volle Unterstützung für Fließkommazahlen. Werden Operanden verschiedener Zahlentypen durch einen Operator verknüpft, dann werden ganzzahlige Operanden in Fließkommazahlen umgewandelt:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Auch komplexe Zahlen werden unterstützt. Der Imaginärteil wird mit dem Suffix `j` oder `J` angegeben. Komplexe Zahlen mit einem Realanteil, der von Null verschieden ist, werden als `(real+imagj)` geschrieben oder können mit der Funktion `complex(real, imag)` erzeugt werden.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3 + 1j * 3
(3+3j)
>>> (3 + 1j) * 3
(9+3j)
>>> (1 + 2j) / (1 + 1j)
(1.5+0.5j)
```

Komplexe Zahlen werden immer durch zwei Fließkommazahlen repräsentiert, dem Realteil und dem Imaginärteil. Um diese Anteile einer komplexen Zahl *z* auszuwählen, stehen `z.real` und `z.imag` zur Verfügung.

```
>>> a = 1.5 + 0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Die Konvertierungsfunktionen in Fließkommazahlen und Ganzzahlen (`float()`, `int()`) stehen für komplexe Zahlen nicht zur Verfügung. Man kann `abs(z)` verwenden, um den Betrag einer komplexen Zahl (als Fließkommazahl) zu berechnen, oder `z.real`, um den Realteil zu erhalten:

```
>>> a = 3.0 + 4.0j
>>> float(a)
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

Im interaktiven Modus wird der zuletzt ausgegebene Ausdruck der Variablen _ zugewiesen. Die ist besonders hilfreich, wenn man den Python-Interpreter als Taschenrechner einsetzt

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Die Variable _ sollte man so behandeln, als wäre sie schreibgeschützt und ihr nicht explizit einen Wert zuweisen. Dadurch würde eine unabhängige lokale Variable mit demselben Namen erzeugt, die die eingebaute Variable _ mit ihrem speziellen Verhalten verdeckt.

3.1.2. Zeichenketten (Strings)

Außer mit Zahlen kann Python auch mit Zeichenketten umgehen, die auf unterschiedliche Weise darstellbar sind. Sie können in einfache oder doppelte Anführungszeichen eingeschlossen werden:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Ja," hat er gesagt.'
'"Ja," hat er gesagt.'
>>> "\"Ja,\" hat er gesagt."
'"Ja," hat er gesagt.'
>>> '"Isses nich\'," sagte sie.'
'"Isses nich\'," sagte sie.
```

Der Interpreter gibt das Ergebnis von Zeichenketten-Operationen auf die gleiche Weise aus, wie sie eingegeben werden: Innerhalb von Anführungszeichen und mit durch Backslashes maskierten Anführungszeichen oder anderen seltsamen Zeichen, um den exakten Wert wiederzugeben. Die Zeichenkette wird von doppelten Anführungszeichen eingeschlossen, wenn sie ein einfaches Anführungszeichen, aber keine doppelten enthält, sonst wird sie von einfachen Anführungszeichen eingeschlossen. Die Funktion `print()` produziert eine lesbarere Ausgabe.

Es gibt mehrere Möglichkeiten, mehrzeilige Zeichenkettenliterale zu erzeugen, zum Beispiel durch Fortsetzungszeilen, die mit einem Backslash am Ende der physikalischen Zeile anzeigen, dass die nächste Zeile die logische Fortsetzung der aktuellen ist:

```
hello = "Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben
würde.\n\
    Achtung: Leerzeichen am Anfang haben eine Bedeutung\
für die Darstellung."

print(hello)
```

Zu beachten ist, dass Zeilenumbrüche immer noch in den Zeichenkette mit Hilfe von `\n` eingebettet werden müssen. Der auf den Backslash folgende Zeilenumbruch gehört allerdings nicht mit zur Zeichenkette. Die vom Beispiel erzeugte Ausgabe sieht so aus :

```
Dies ist eine ziemlich lange Zeichenkette,
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde.
    Achtung: Leerzeichen am Anfang haben eine Bedeutung für die
Darstellung.
```

Zeichenketten können auch mit einem Paar von dreifachen Anführungszeichen umgeben werden: `"""` oder `'''`. Zeilenenden müssen nicht hierbei escaped werden, sondern werden in die Zeichenkette übernommen. Deshalb wird im folgende Beispiel das erste Zeilenende escaped, um die unerwünschte führende Leerzeile zu vermeiden:

```
print("""\

Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

Das erzeugt folgende Ausgabe:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Wenn wir den Zeichenkettenliteral zu einem “raw”-String machen, wird `\n` nicht in einen Zeilenumbruch umgewandelt; auch der Backslash am Ende und das Zeilenumbruch-Zeichen im Quellcode sind Teil der Zeichenkette. Das Beispiel:

```
hello = r"Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde."

print(hello)
```

führt zu folgender Ausgabe:

```
Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde.
```

Zeichenketten können mit dem `+`-Operator verkettet und mit `*` wiederholt werden:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Zwei Zeichenkettenliterale nebeneinander werden automatisch miteinander verknüpft. Die erste Zeile im obigen Beispiel hätte also auch `word = 'Help' 'A'` lauten können. Das funktioniert allerdings nur mit zwei Literalen, nicht mit beliebigen String-Ausdrücken:

```
>>> 'str' 'ing'             #Das ist ok
'string'
>>> 'str'.strip() + 'ing'   #Das ist ok
'string'
>>> 'str'.strip() 'ing'     #Das ist ungültig
 File "<stdin>", line 1, in ?
   'str'.strip() 'ing'
                 ^
SyntaxError: invalid syntax
```

4.8. Intermezzo: Schreibstil

Jetzt da Du längere, komplexere Stücke in Python schreibst, ist es an der Zeit einmal über den Schreibstil (*coding style*) zu sprechen. Viele Sprachen können in verschiedenen Stilen geschrieben (präziser: *formatiert*) werden; davon sind manche lesbarer als andere. Es anderen leichter zu machen Deinen Code zu lesen ist immer eine gute Idee und sich einen schönen Schreibstil anzugewöhnen hilft dabei ungemein.

Für Python hat sich **PEP 8** als der Styleguide herauskristallisiert, dem die meisten Projekte folgen. Es fördert einen sehr lesbaren Schreibstil, der angenehm zu lesen ist. Jeder Pythonentwickler sollte ihn irgendwann einmal lesen, hier jedoch die wichtigsten Punkte:

- Benutze eine Einrückung von 4 Leerzeichen, keine Tabs.
  4 Leerzeichen sind ein guter Kompromiss zwischen geringer Einrückung, die eine größere Verschachtelungstiefe ermöglicht, und größerer Einrückung, die den Code leichter lesbar macht. Tabs führen zu Unordnung und sollten deshalb vermieden werden.
- Breche Zeilen so um, dass sie nicht über 79 Zeichen hinausgehen.
  Das ist hilfreich für Benutzer mit kleinen Bildschirmen und macht es auf größeren möglich mehrere Dateien nebeneinander zu betrachten.
- Benutze Leerzeilen, um Funktion und Klassen, sowie größere Codeblöcke innerhalb von Funktionen zu trennen.
- Verwende eine eigene Zeile für Kommentare, sofern das möglich ist.
- Schreibe Docstrings.
- Benutze Leerzeichen um Operatoren herum und nach Kommas, jedoch nicht direkt innerhalb von Klammerkonstrukten: `a = f(1, 2) + g(3, 4).`
- Benenne Deine Klassen und Funktionen konsistent: Die Konvention schlägt `CamelCase` für Klassen und `klein_geschrieben_mit_unterstrichen` für Funktionen und Methoden vor. Benutze immer `self` als Namen für das erste Methoden Argument (mehr zu Klassen und Methoden, siehe Eine erste Betrachtung von Klassen).
- Benutze keine ausgefallenen Dateikodierungen, wenn Dein Code für ein internationales Publikum vorgesehen ist. Pythons Standardkodierung – UTF-8 – oder sogar einfaches ASCII ist in jedem Fall am Besten.
- Benutze auch keine nicht-ASCII-Zeichen in Bezeichnern, wenn es auch nur den Hauch einer Chance gibt, dass der Code von Menschen gelesen oder gewartet wird, die eine andere Sprache sprechen.

Fußnoten

[1] Eigentlich wäre *call by object reference* eine bessere Beschreibung, denn wird ein veränderbares Objekt übergeben, sieht der Aufrufende jegliche Veränderungen, die der Aufgerufene am Objekt vornimmt (beispielsweise Elemente in eine Liste einfügt)

8. Fehler und Ausnahmen

Bis jetzt wurden Fehlermeldungen nur am Rande erwähnt, aber wenn Du die Beispiele ausprobiert hast, hast Du sicherlich schon einige gesehen. Es gibt (mindestens) zwei verschiedene Arten von Fehlern: *Syntaxfehler* (engl.: syntax errors) und *Ausnahmen* (engl.: exceptions).

8.1. Syntaxfehler

Syntaxfehler, auch Parser-Fehler genannt, sind vielleicht die häufigsten Fehlermeldungen, die Du bekommt, wenn Du Python lernst:

```
>>> while True print('Hallo Welt')
  File "<stdin>", line 1, in ?
    while True print('Hallo Welt')
                     ^
SyntaxError: invalid syntax
```

Der Parser wiederholt die störende Zeile und zeigt mit einem kleinen 'Pfeil' auf die Stelle, an der der Fehler entdeckt wurde. Der Fehler ist an dem Token aufgetreten (oder wurde zumindest dort entdeckt), welches *vor* dem Pfeil steht: In dem Beispiel wurde der Fehler bei der `print()`-Funktion entdeckt, da ein Doppelpunkt (`':'`) vor der Funktion fehlt. Des weiteren werden der Dateiname und die Zeilennummer ausgegeben, sodass Du weißt, wo Du suchen musst, falls die Eingabe aus einem Skript kam.

8.2. Ausnahmen

Selbst wenn eine Anweisung oder ein Ausdruck syntaktisch korrekt ist, kann es bei der Ausführung zu Fehlern kommen. Fehler, die bei der Ausführung auftreten, werden *Ausnahmen* (engl: exceptions) genannt und sind nicht notwendigerweise schwerwiegend: Du wirst gleich lernen, wie Du in Python-Programmen mit ihnen umgehst. Die meisten Ausnahmen werden von Programmen aber nicht behandelt, und erzeugen Fehlermeldungen, wie dieses Beispiel zeigt:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: int division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

Die letzte Zeile der Fehlermeldung gibt an, was passiert ist. Es gibt verschiedene Typen von Ausnahmen, und der Typ der Ausnahme wird als Teil der Meldung ausgegeben: Die Typen in diesem Beispiel sind `ZeroDivisionError`, `NameError` und `TypeError`. Die Zeichenkette, die als Ausnahmetyp ausgegeben wird, ist der Name der eingebauten Ausnahme, die aufgetreten ist. Dies gilt für alle eingebauten Ausnahmen, muss aber für benutzerdefinierte Ausnahmen

nicht zutreffen (es ist aber eine nützliche Konvention). Standard-Ausnahmen sind eingebaute Bezeichner (keine reservierten Schlüsselwörter).

Der Rest der Zeile gibt Details an, die auf dem Ausnahmetyp und darauf, was die Ausnahme ausgelöst hat, basieren.

Der vorangehende Teil der Fehlermeldung zeigt den Zusammenhang, in dem die Ausnahme auftrat, in Form eines Traceback. Im Allgemeinen wird dort der Programmverlauf mittels der entsprechenden Zeilen des Quellcodes aufgelistet; es werden jedoch keine Zeilen ausgegeben, die von der Standardeingabe gelesen wurden.

Built-in Exceptions listet alle eingebauten Ausnahmen und ihre Bedeutung auf.