Python-Einführung

René Schwaiger

Contents

1 Grundlagen

Wie bei jedem Handwerk gibt es auch beim Programmieren gewisse Werkzeuge die man benötigt um eine Applikation zu erstellen. Das erste Werkzeug, dass ein angehender Programmierer verwendet ist wahrscheinlich ein **Text-Editor**. In diesem Programm schreibt man dann wie schon der Name andeutet Text, so-genannten **Code**, in einer bestimmten **Programmiersprache**. Je nach Programmiersprache beschreibt der Code entweder

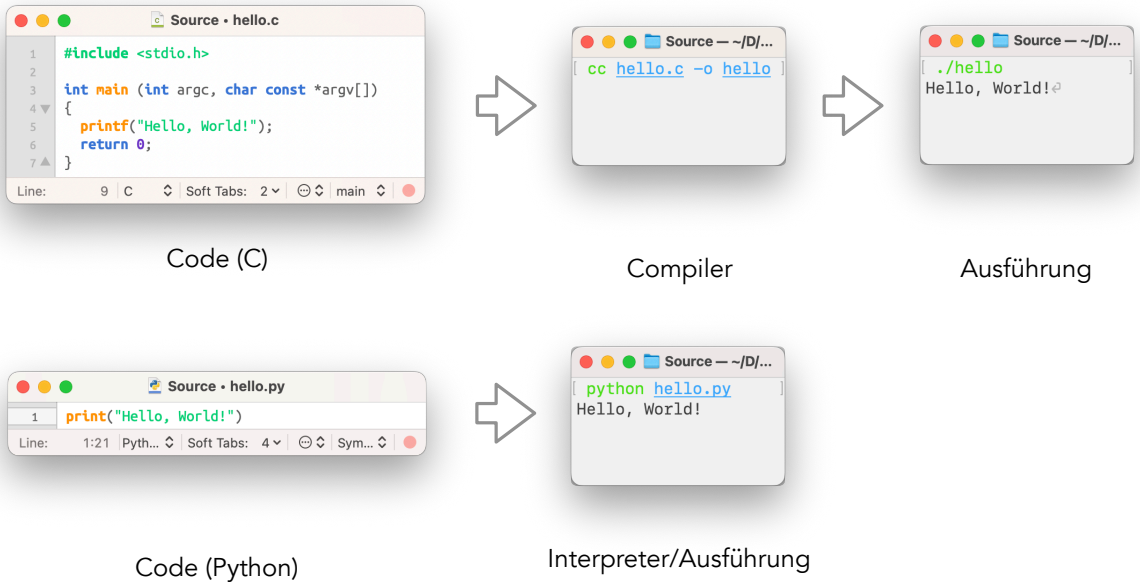
1. welche Schritte ein Computer ausführen soll (imperativ), oder
2. das Problem selber (deklarativ).

> Ein Programmiersprache ist wesentlich einfacher aufgebaut als eine menschliche Sprache. Das mag zu dem Gedanken verleiten, dass es einfacher ist einem Computer als einer Person beizubringen, wie ein bestimmtes Problem aussieht (deklarative Programmierung) oder wie es zu lösen ist (imperative Programmierung). Das ist aber ein Trugschluss. Im Endeffekt benötigt der Computer (durch die Einfachheit der Programmiersprache) eine wesentlich genauere Beschreibung als ein Mensch, führt diese aber dafür (im Normalfall) fehlerfrei und äußerst schnell durch.

Der „fertige“ Code wird entweder

1. von einem **Compiler** in Maschinen-Befehle übersetzt (**kompiliert**) und dann vom Computer ausgeführt, oder
2. direkt vom einem **Interpreter** ausgeführt (**interpretiert**).

Üblicherweise findet, zur Optimierung der Geschwindigkeit eines Programms, auch bei der Interpreter-Variant eine Übersetzung in maschinen-nahen Code statt. Als Programmierer muss man sich dabei aber – im Gegensatz zur Compiler-Variante – üblicherweise keine Gedanken machen.



1.1 Die Werkzeuge

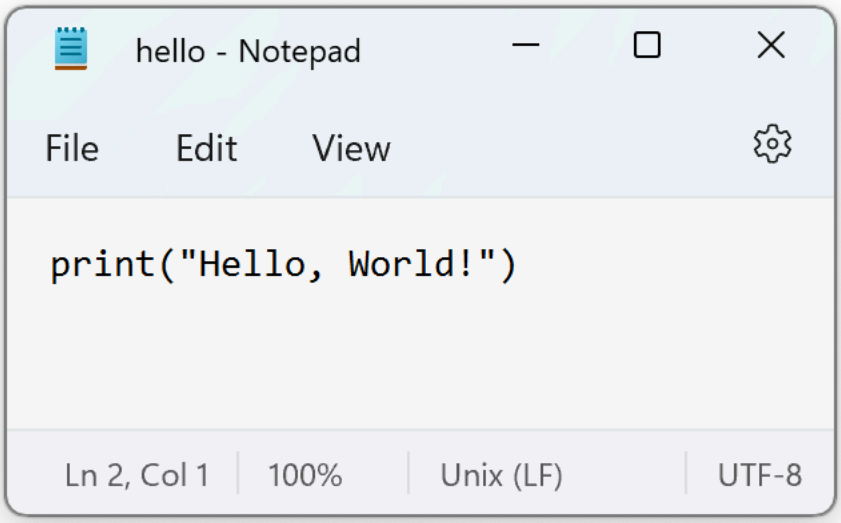
1.1.1 Editor

Wie schon in einführenden Teil beschrieben ist ein Text-Editor oder auch kurz Editor eines der wesentlichen Werkzeuge eines Programmierers. Von einer Textverarbeitung (wie z.B. “Word”, “LibreOffice” oder “Google Docs”) unterscheidet sich ein Texteditor, dadurch, dass man damit wirklich nur Text schreiben kann. Eine Formatierung oder das Einfügen von Bildern ist nicht möglich.

Als essentielles Werkzeug ist ein Texteditor Teil der Standard-Installation vieler Betriebssystemen:

- Linux: gedit, Kate, Vim, Emacs, ed
- macOS: TextEdit, Vim, Emacs, ed
- Windows Notepad

Theoretisch wäre es also möglich z.B. einfach Notepad zu verwenden und darin zukünftige Programmier-Projekte zu verwirklichen.



In der Praxis verwendet man aber üblicherweise einen speziell für die Programmierung angepassten Texteditor (manchmal auch Code-Editor genannt).

1.1.2 Command Line

Während man heutzutage als Anwender meist eine grafische Benutzeroberfläche (**GUI: Graphical User Interface**) verwendet, ist es als Programmierer üblich für viele Aufgaben in einer text-basierten Oberfläche (**CLI: Command Line Interface**) zu erledigen. Diese hat den Vorteil, dass man hier (wesentlich leichter als in einem GUI) Schritte, wiederum mittels Programmierung, automatisieren kann. Das heißt auch bei den **Text** den man bei einer textbasierten Oberfläche eingibt handelt es sich um **Code in einer bestimmten Programmiersprache**.

Um eine kleine Einführung in eine text-basierte Oberfläche zu geben schauen wir uns hier an wie man unter Windows Software mittels CLI installiert. Dazu installieren wir als erstes „Windows Terminal“ in der Powershell. Dabei handel es sich um einem Command-Line-Interpreter für die **Programmiersprache PowerShell**.

1. Das Programm PowerShell öffnen

    1. „Windows-Taste“ drücken
    2. „PowerShell“ eingeben
    3. Mit „Return“ ( ) bestätigen

2. Den folgenden Text (Code) in das PowerShell-Fenster einfügen

    ```
    winget install --id=Microsoft.WindowsTerminal -e
    ```

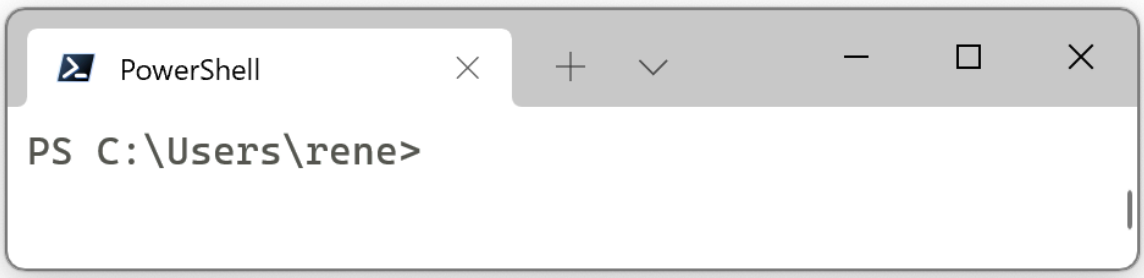
    und dann mittels (Return) ausführen.

    - Bei `winget` handelt es sich hierbei um ein Programm (**Befehl**) zum Suchen und Installieren von Software, die in diesem Zusammenhang auch als **Paket** bezeichnet wird.
    - Mit dem **(Sub-)Befehl** (Argument) `install` teilt man `winget` mit, dass es ein Paket installieren soll.

- Der Name des Pakets wird mit der (Long-)**Option** (`--`) `id` mit dem **Argument** `Microsoft.WindowsTerminal` spezifiziert
- Die (Short-)**Option** (`-`) `e` wiederum gibt an, dass das das genau (exakt) das Paket `Microsoft.WindowsTerminal` installiert werden soll

3. Zeit zum auf die Schulter klopfen, nachdem du im 2. Schritt (wahrscheinlich) dein erstes PowerShell-„Programm“ geschrieben hast

Um zu sehen ob die Installation von „Windows Terminal“ funktioniert hat suchen wird nach dem Programm Windows Terminal (einem mehr oder weniger modernerer Ersatz des Programms PowerShell) und führen es aus.



Nun wollen wir in einem zweiten Schritt einen Code-Editor installieren. Ein beliebtest Programm ist dabei Visual Studio Code von Microsoft.

1. Da wir nicht genau wissen wie das Paket für Visual Studio heißt suchen wir mit dem Befehl Kommando/Argument `search` nach dem Paket

   ```
   winget search 'Visual Studio Code'
   ```

   Die Anführungszeichen (Single Quotes) sorgen dabei dafür, dass der Text als als Ganzes interpretiert wird und nicht als die 3 Argumente `Visual`, `Studio` und `Code`.

2. Die Ausgabe des obigen Befehls/Code sollte nun die gefunden Pakete/Software anzeigen

   ```
   Name                                  Id                                      Version Source
   ------------------------------------------------------------------------------------------
   Visual Studio Code                    XP9KHM4BK9FZ7Q                          Unknown msstore
   Visual Studio Code - Insiders         XP8LFCZM790F6B                          Unknown msstore
   Microsoft Visual Studio Code          Microsoft.VisualStudioCode              1.65.2  winget
   Microsoft Visual Studio Code Insiders Microsoft.VisualStudioCode.Insiders     1.66.0  winget
   ```

3. Wir entscheiden uns für das Paket mit der Id `Microsoft.VisualStudioCode` und installieren dieses mit dem Befehl

   ```
   winget install -e --id Microsoft.VisualStudioCode
   ```

   Am obigen Befehl sehen wir, dass die Reihenfolge von Option (`e` und `id`) normalerweise keine Reihenfolge spielt. Weiters können wir feststellen, dass wir statt des Gleichheitszeichen zur Trennung der Option `id` und des dazugehörigen Arguments (`Microsoft.VisualStudioCode`) auch einfach Leerzeichen verwendet werden können.

Bevor wir nun Visual Studio Code öffnen und unser erstes Python-Programm erstellen sollten wir noch den Python-Interpreter installieren.

1. Eine Suche mittels

   ```
   winget search Python
   ```

   zeigt und, dass die richtige Id des Programms wohl `Python.Python.3` ist

2. Wir installieren Python 3 mittels:

   ```
   winget install Python.Python.3 -e
   ```

   Der obige Befehl zeigt uns, dass man die Option `id` auch weglassen kann und `winget` in diesem Fall annimmt, dass es sich beim Argument `Python.Python.3` um den Namen des Pakets handelt.

Zum Schluss unser ersten Ausflug in die Programmiersprache PowerShell wollen wir noch das Programm „Hello, World!“ schreiben. Dabei handelt es sich um ein ein typische erstes Programm, dass den Text (String) „Hello, World!“ auf dem Bildschirm ausgibt. Im Wikipedia-Artikel zur Programmiersprache PowerShell sehen wir, dass der Befehl `Write-Output` verwendet werden kann um einen Text auf dem Bildschirm auszugeben. Eine typische Implementierung (Realisierung eines Problems in Code) von „Hello, World!“ in PowerShell-Code könnte also z.B. so aussehen:

```
Write-Output "Hello, World!"
```

Die doppelten Anführungszeichen erfüllen dabei den gleichen Zweck wie die einfachen Anführungszeichen beim Befehl `winget search 'Visual Studio Code'`. Der Text `Hello, World!` wird als einzelnes Argument interpretiert und nicht als die zwei Argumente `Hello,` und `World!`.

1.1.3 Interpreter

Nachdem wir „Hello, World!“ schon in PowerShell geschrieben haben, wollen wir das gleiche Programm nun in der **Programmiersprache Python** implementieren. Der Code dafür kann z.B. so aussehen

```
print("Hello, World!")
```

Wir sehen, dass der übliche Befehl (Funktion) zum Ausgeben eines Texts (**String**) in Python `print` heißt. Das Argument des Befehls `"Hello, World!"` wird hier, wie auch in der PowerShell, unter doppelte (oder einfache) Anführungszeichen gesetzt. Damit wird in Python angezeigt, dass es sich um einen Text (String/Zeichenkette) und nicht um einen Befehl (wie z.B. bei der Funktion `print`) handelt.

Wir haben num verschiedene Möglichkeiten unser Programm auszuprobieren. Eine der Möglichkeiten ist die Ausführung direkt im **Python-Interpreter**.

1. Python-Interpreter öffnen

   - Möglichkeit 1: PowerShell öffnen, den Befehl `python` eingeben und mittels (Return) bestätigen
   - Möglichkeit 2: Mit „Windows-Taste“ die Suche öffnen, den Text “Python” eingeben um nach den Interpreter zu suchen und diesen öffnen

2. Den Code von oben in den Interpreter kopieren und dann mittels (Return) ausführen

3. Der Text „Hello, World!“ wird unter dem Code ausgegeben

```
>>> print("Hello, World!")
Hello, World!
```

Bisher haben wir Code nur direkt im Interpreter ausgeführt ohne diesen vorher in einer Text-Datei zu speichern. Angesichts dessen, dass der bisherige Code nur eine Zeile lang war – dieser Code wird auch oft als "one liner" bezeichnet – war das auch nicht wirklich nötig.

Nun wollen wir den Code unseres „Hello, World!"-Programms mittels Visual Studio Code speichern und ausführen.

1. Visual Studio Code öffnen
2. Eine neue Datei erstellen: Ctrl + N
3. Den Code `print("Hello, World!")` einfügen
4. Die Datei mittels Ctrl + S unter dem Namen `hello.py` speichern
5. Mittels „Run" → „Run Without Debugging" ausführen

1.1.4 IDE

Bei einem "Integrated Development Environment" handelt es sich um ein Programm, dass den Alltag eines Programmierers erleichtern soll. Dazu bündelt dieses üblicherweise Werkzeuge wie z.B.

- Code-Editor,
- Command-Line-Interface,
- Debugger (Programm um Fehler in Programmen zu finden und zu beheben), und
- Versionsverwaltung (zum Verwalten von Code-Änderungen)

in einer Oberfläche. Der Übergang von Texteditor mit Zusatz-Fähigkeiten wie z.B. das Ausführen des Codes in einem Schritt zu einer IDE ist dabei relativ fließend. Manche Leute würden z.B. Visual Studio Code auch schon als IDE bezeichnen.

Beliebte IDEs für die Entwicklung von Python-Code sind z.B.

- Spyder oder
- PyCharm.

1.2 Wiederholungsfragen

1. Installiere das Programm "PowerToys" mittels `winget`
2. Deinstalliere "PowerToys" mittels `winget uninstall`
3. Finde raus wie du mittels `winget` Software auf deinem Computer auf den neuesten Stand bringen kannst
4. Nenne zumindest zwei Programmiersprachen
5. Was unterscheidet einen Interpreter von einem Compiler?
6. Schreibe ein Programm das den Text "Goodby Cruel World" ausgibt in
    1. PowerShell
    2. Python

2 Datentypen & Ausdrücke

2.1 REPL

Bisher haben wir den meisten Code direkt im Powershell- oder Python-**Interpreter** (`python`) geschrieben. Dabei geben wir eine oder mehrere Zeilen von Code ein und führen diese – üblicherweise nach der Eingabe mittels Return – aus. Diese Art der Eingabe-Oberfläche wird oft auch als sogenannte **REPL** (Read-Eval-Print Loop) bezeichnet.

Wie bei vielen anderen interpretierten Sprachen existieren auch bei Python alternative REPLs, die das Arbeiten mit Code nochmals vereinfachen können. Eine solche REPL mit dem Namen `ptpython` wollen wir nun installieren. Dazu verwenden wir `pip`, einen Package-Manager für Python. Ähnlich wie `winget` (und viele andere Package-Manager) kann man das Sub-Kommando `install` dazu verwenden Software zu installieren:

```
pip install ptpython
```

Nach der Installation können wir `ptpython` mittels Eingabe des Befehls

```
ptpython
```

verwenden und uns mit der Bedienung davon ein wenig vertraut machen. Wie wir im Verlauf noch sehen werden bietet `ptpython` in Vergleich zur Standard-REPL (`python`) Funktionen wie **Autovervollständigung** und **Syntax-Highlighting**.

2.2 Ausdrücke

Bei einem Ausdruck (`Expression`) handelt es sich um ein (üblicherweise relativ kurzes) Stück Code, dass ausgewertet werden kann und dann ein bestimmtes Ergebnis eines bestimmten Datentyps liefert.

2.2.1 Konstanten

Einer der einfachsten Varianten von Ausdrücken sind Konstanten. Diese kennt man eventuell auch schon aus dem Mathematik-Unterricht. Hier mal ein paar Beispiele:

Konstante/Ausdruck	Datentyp
`True`	`bool` (Boolscher Wert)
`1234`	`int` (Ganzzahl)
`12.34`	`float` (Gleitkommazahl)
`"1234"`	`str` (Zeichenkette)
`'1234'`	`str` (Zeichenkette)

Um heraus zu finden welchen Typ ein bestimmter Ausdruck hat kann man die Funktion type verwenden:

```
type(True)
```

```
<class 'bool'>
```

```
type(1234)
```

```
<class 'int'>
```

```
type(12.34)
```

```
<class 'float'>
```

```
type("1234")
```

```
<class 'str'>
```

```
type('1234')
```

```
<class 'str'>
```

2.2.2 Funktionen und Operatoren

Diverse Funtkionen wie z.B. `abs`, `min` und Operatoren wie z.B. `+`, `-` , `**` (Potzenz) können ebenfalls Teil eines Ausdrucks sein.

```
1+2
```

```
3
```

```
"1" + "2"
```

```
'12'
```

```
2**3
```

```
8
```

```
abs(-123)
```

```
123
```

```
min(12.4, 12.3)
```

```
12.3
```

```
max(12.4, 12.3)
```

```
12.4
```

```
min(max(1,2,3), max(-1,-2,-3))
```

```
-1
```

Wie man bei der Funktion `min` sieht werden die Argumente einer Funktion in Python durch Beistriche getrennt.

- Warum unterscheidet sich das Ergebnis des ersten und des zweiten Ausdrucks?
- Um welche mathematische Operation handelt es sich bei `**`?
- Welchen Wert ermitteln die Funktionen `min`, `max` und `abs`?
- Welche Typen haben die obigen Ausdrücke?

Operationen wie `+` oder `**` können ebenfalls als Funktion angesehen werden, die

- einen speziellen Namen (wie z.B. `+`, `-` , `*`) besitzen und
- meist **In**fix-Notation (Funktion/Operator zwischen Argumenten) statt **Post**fix-Notation (Argumente nach Operation/Funktion) verwenden.

So gibt es neben dem Operator `+` z.B. auch eine Funktion mit mehr oder weniger gleicher Funktionalität mit dem Namen `add`:

```
from operator import add # Funktion add importieren

add(1, 2) # Argumente nach Funktion (Postfix-Notation)
# 3
1 + 2 # Argumente vor und nach Funktion/Operator (Infix-Notation)
# 3
```

Neben den von Haus aus sichtbaren Funktionen/Operatoren, wie z.B. `abs` und `-` gibt es noch viele andere Funktionen/Operatoren, die erst importiert werden müssen bevor man sie verwenden kann. Ein Beispiel dafür sehen wir in der ersten Zeile des obigen Codeblocks in dem die Funktion `add` aus dem Modul `operator` importiert wird.

Ein weitere Neuerung im obrigen Text sind Kommentare. Diese beginnen in Python mit dem Zeichen `#`. Der Text hinter diesem Zeichen (bis zum Zeilenende) wird vom Interpreter ignoriert. Kommentare dienen z.B. dazu zu **dokumentieren**

- **warum** ein bestimmte Löung für ein Problem gewählt wurde
- welche **Probleme** auftreten könnten oder auch
- **wie** ein bestimmtes Problem gelöst wurde.

Dabei sollte man darauf achten **keine trivialen Kommentare** zu schreiben. Ein Kommentar wie z.B.

```
1 + 2 # Die Ganzzahlen 1 und 2 werden addiert
```

ist höchstens für absolute Programmierneulinge interessant. Das soll natürlich keine Aufforderung keine Kommentare zu schreiben. **Kommentare** und andere Form der **Dokumentation** sind ein **essentieller Teil guter Software**.

2.2.2.1 Argumente (Input) und Rückgabewerte (Output)

Jede Funktion in Python übernimmt eine **bestimmte Anzahl von Argumenten** und returniert **einen Rückgabewert**. Dabei ist zu beachten, dass die Anzahl von Argumenten auch 0 sein kann. Eine Funktion kann auch `None` retunieren. Dieser Wert vom Typ `NoneType` steht im Endeffekt für „keinen Wert". Schauen wir uns die Eingabewerte von Funktionen und Operatoren an Hand von ein paar Beispielen an.

```
1 + 2
```

```
3
```

Die Operation `+` übernimmt hier **zwei Argumente** vom Typ `int` und gibt die Summe dieser Werte `3` (Typ `int`) zurück. Die Operation `+` ist auch auf andere Datentypen wie z.B. `float` und `str` definiert. Bei nachfolgendem Ausdruck:

```
1 + 4.5
```

```
5.5
```

handelt es sich beim ersten Argumenten wiederum um eine **Ganzzahl** (`int`) und beim zweiten Argument um eine **Gleitkommazahl** (`float`). Beim Rückgabewert von `5.5` handelt es sich wiederum um eine **Gleitkommazahl**. Die Operation `+` ist auch auf Strings definiert, wobei hierbei die String aneinandergefügt werden:

```
"Hello, " + "World"
```

```
'Hello, World'
```

Beim Rückgabewert handelt es sich wiederum um einen String:

```
type("Hello, " + "World")
```

```
<class 'str'>
```

Manche Funktionen können auch eine beliebige Anzahl von Argumenten übernehmen. Eine dieser Funktionen ist `min`:

```
min(1, -10)
```

```
-10
```

```
min(1, -1, 0.5, -2)
```

```
-2
```

Zum Schluss wollen wir uns noch die Funktion `print` ansehen, die wir schon verwendet haben um die Zeichenkette "Hello, World!" auf dem Bildschirm auszugeben. Diese kann wiederum eine beliebige Anzahl von Argumenten übernehmen und gibt diese auf dem "Standard Output" (`stdout`), also üblicherweise dem Bildschirm, getrennt durch Leerzeichen aus. Die Funktion gibt dabei aber keinen Wert zurück (`None`).

```
print("Hello,", "World")
```

```
Hello, World
```

```
type(print("Hello,", "World"))
```

```
Hello, World
<class 'NoneType'>
```

- Warum gibt der Interpreter beim Aufruf des zweiten Ausdrucks sowohl den Typ des Ausdrucks (`<class 'NoneType'>`) als auch den Text "Hello, World" aus?

Die Funktion `print` kann auch ohne Argumente aufgerufen werden. Das sorgt dafür, dass ein leere Zeile (ein Zeilenvorschub) auf `stdout` ausgegeben wird.

```
print("One")
print()
print("Two")
```

```
One

Two
```

- Wie lautet das Ergebnis der untenstehenden Ausdrücke?
- Welchen Type haben die Argumente und der Rückgabewert der Ausdrücke?

Versuche die Aufgabe als erstes **im Kopf** zu lösen und gib diese danach im Python-Interpreter ein um dein Ergebnis zu überprüfen.

Welche Aufgaben dabei die noch nicht besprochenen Funktionen (wie z.B. `float`) übernehmen kannst du dabei feststellen indem du diese inklusive erster Klammer – also z.B. `float(` – in `ptpython` eingibst. Eine andere Möglichkeit ist in der Python-Dokumentation nach der jeweiligen Funktion zu suchen.

```
1
1.0
float(1)
int("1")
1+10 * 3.3
1.1
print(1+3)
str(1) + "10"
True
int(True)
int(False)
bool(1337)
```

2.2.3 Vergleichsoperatoren

Eine, wie wir später noch sehen werden, in Programmiersprachen sehr wichtige Art Aufgabe übernehmen Vergleichsoperatoren. Diese Arten von Operatoren

- vergleichen **zwei Argumente** und
- geben ein **Ergebnis vom Typ `bool`**, also entweder `True` oder `False` zurück.

Der häufigste Vergleichsoperator ist wohl `==` der seine zwei Argumente auf **Gleichheit** überprüft:

```
1 == 2
```

```
False
```

```
"Hello" == 'Hello'
```

```
True
```

```
"hello" == "Hello"
```

```
False
```

```
None == None
```

```
True
```

Der genau entgegengesetzte Operator zu `==` ist `!=` und überprüft ob die Ausdrücken links und rechts **nicht gleich** (unterschiedlich) sind.

```
1 != 1
```

```
False
```

```
1337 != 4
```

```
True
```

```
"Hello" != 'hello'
```

```
True
```

```
None != None
```

```
False
```

- Wie lautet das Ergebnis der untenstehenden Ausdrücke?

```
"1" == '1'
1 != 1
True == False
True != False
2 != 2.2
```

Neben Gleichheit kann man bestimmte Ausdrücke, wie z.b. Zahlen, auch mit

- `<` (kleiner) ,
- `<=` (kleiner gleich) ,
- `>` (größer), oder
- `>=` (größer gleich)

vergleichen.

```
5 < 6
```

```
True
```

```
6 < 6
```

```
False
```

```
6 <= 6
```

```
True
```

```
2 >= 2
```

```
True
```

```
2.1 > 2.01
```

```
True
```

- Wie lautet das Ergebnis der untenstehenden Ausdrücke?

```
20 > 21
20 >= 21
1 >= 1
1 < 1
```

2.2.4 Boolsche Operatoren

Boolsche Ausdrücke, also Ausdrücke die entweder den Wert `True` oder `False` annehmen sind ein wichtiger Bestandteil praktisch jedes Programms. Um Boolsche Ausdrücke miteinander zu kombinieren kann man boolsche Operatoren verwenden. Hier wollen wir kurz auf die wichtigsten dieser Operatoren:

- `not`,
- `and`, und
- `or`

eingehen.

2.2.4.1 `not` Der Operator `not` (nicht) wandelt **einen boolschen Ausdruck** in das **Gegenteil** um. Das heißt

- aus `not True` wird `False` und
- aus `not False` wird `True`.

Hierzu ein paar Beispiele:

```
not False
```

```
True
```

```
not (1 >= 2)
```

```
True
```

```
not not not True
```

```
False
```

Die Klammern `()` im zweiten Ausdruck dienen dazu sicherzustellen, dass `1 >= 2` als erstes ausgewertet wird. Da `not` **stärker** wie Vergleichsoperatoren, wie `>=`, **bindet** und **Ausdrücke von links nach rechts** ausgewertet werden würde beim Ausdruck

```
not 1 >= 2
```

als erstes `not 1` ausgewertet. Da jede Zahl außer `0` als wahr (`True`) interpretiert wird kann dieser Ausdruck auch als `not True` geschrieben werden. Das Ergebnis des Teilausdrucks ist also `False`:

```
not 1
```

```
False
```

Dadurch ergibt sich der Ausdruck:

```
False >= 2
```

Hierbei wird **implizit** `False` in `0` (und `True` in `1`) umgewandelt. Daduch ergibt sich der Ausdruck:

```
0 >= 2
```

der den Wert `False` zurückliefert. Wir können unser Ergebnis nochmals überprüfen indem wir den Ausdruck ohne Klammern in den Interpreter eingeben:

```
not 1 >= 2
```

```
True
```

Zu der impliziten Umwandlung von Zahlen in boolsche Ausdrücke ist noch zu sagen, dass man sich auf diese nicht verlassen sollte. Wie wir oben gesehen haben ist diese Auswertung nicht unbedingt intuitiv. Es macht Sinn

- **Ausdrücke entweder** so zu **klammern**, dass keine Zahlen mit boolschen Ausdrücken verglichen werden, oder
- Ausdrücke vorher **explizit** mittels den Funktionen`bool` in einen boolschen Ausruck oder mittels `int` in eine Zahl **umzuwandeln**.

2.2.4.2 `and` Mittels des Operators `and` kann man zwei boolsche Ausdrücke miteinander verbinden. Dabei wird der kombinierte Ausdruck genau dann wahr (`True`) wenn der Ausdruck links und rechts (von `and`) wahr sind:

```
False and False
```

```
False
```

```
True and False
```

```
False
```

```
False and True
```

```
False
```

```
True and True
```

```
True
```

```
(1 < 2) and (1 <= 2)
```

```
True
```