Einführung in Python - Teil 1

Lernziel

Lernziel

```python
def biggest(a: int, b: int, c: int) -> int:
    current_biggest: int = a

    if b > a:
        current_biggest = b

    if c > current_biggest:
        current_biggest = c

    return current_biggest


print(biggest(4, 2, 8))
```

Übersicht Python

Vorstellung Python

- Universelle Programmiersprache
- Üblicherweise interpretiert
- Betont eher Lesbarkeit und Ausdrucksstärke des Codes
- Einrückungen (siehe später) sind Teil der Syntax!

Python - Versionen

Python existiert seit 2008 in den Versionen 2 und 3. Es gibt leider Unterschiede zwischen den beiden Versionen, die auch Beginner betreffen (Division, Konsolenausgabe, ...)

Welche Version soll genutzt werden?

Kurz gesagt:

Python 2.x ist Vergangenheit, 3.x ist Gegenwart und Zukunft

- *`https://wiki.python.org/moin/Python2orPython3`*

Außerdem erreicht Python 2 den Status **End-of-life** am 01.01.2020.

Daher: Verwendung von Python 3.x

Duck-Typing

> *Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnattert, dann nenne ich diesen Vogel eine Ente.*

- James Whitcomb Riley

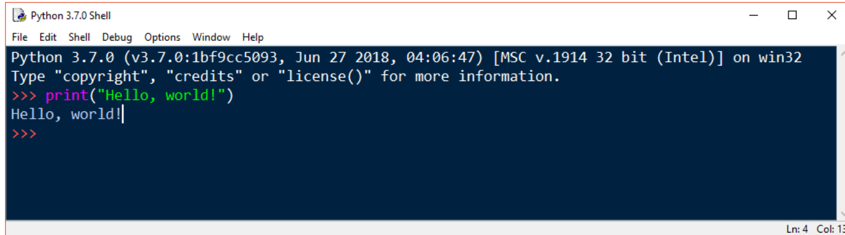
In Python ist das Verhalten eines Objekts nicht durch seine Klasse beschrieben, sondern durch das Vorhandensein bestimmter Methoden oder Attribute:

- Typprüfung zur Laufzeit
- Ist das Merkmal vorhanden, wird das Merkmal benutzt
- Ansonsten kommt es zu einem sogenannten TypeError („Typfehler“)

IDLE

Wir werden *für den Anfang* die von Python bereitgestellte „Integrated Development and Learning Environment“ IDLE verwenden.

Führen Sie das Statement `print('Hello, World!')` auf der IDLE aus. Falls Python richtig installiert wurde (und Sie Sich nicht vertippt haben), sollte der Text `Hello, world!` auf der Konsole angezeigt werden.



Anmerkung: Die Versionsnummer könnte auf Ihrem Rechner unterschiedlich sein.

Kommentare

Alle Zeichen, die einem „#“ folgen, werden bis zum Ende der Zeile als Kommentar gewertet und nicht ausgeführt.

Sie werden meist benutzt, um Erklärungen direkt in den Code einzufügen.

```
# Dies ist ein Kommentar

# Auch dies ist ein Kommentar

5 + 6 # Kommentar
```

Konsolenausgabe

Mit der Funktion `print` kann eine „Konsolenausgabe“ gemacht werden:

```
x = 5
print(x)

y = "Dies ist ein Test"
print(y)
```

Der Beispielcode sollte jeweils „5“ und „Dies ist ein Test“ ausgeben.

Variablen

Um das Ergebnis eines „Ausdrucks“ abspeichern zu können, brauchen wir Variablen.

Ein Ausdruck kann dabei sehr viel sein, zum Beispiel

- Eine mathematische Berechnung
- Der Rückgabewert einer Funktion
- Die Abfrage einer Nutzereingabe
- ...

Man kann sich eine Variable auch wie einen Container für Daten vorstellen.

Syntax: Variablen

Variablen haben einen Namen, einen Wert und einen Typen.

Der Wert einer Variable kann auch ein komplizierterer Ausdruck, der Typ der Variable entspricht dem Typen ihres Werts.

Eine Variable muss vor Verwendung initialisiert werden:

Notation: `name = wert`

Beispiele:

```
x = 5
y = 7 + 3
# y = 10
string = "Hallo Welt!"
```

```
z = (8 + 15j) + (7 - 3j)
# z = 15 + 12j
wahr = 8 < 15
# wahr = True
```

Variablennamen

Für Variablennamen gelten folgende Regeln:

- Variablennamen müssen mit einem Buchstaben oder einem „_“ beginnen
- Variablennamen dürfen keine reservierten Wörter sein (and, or, def, while, for, return, raise, if, else, finally, ...)

Es gibt außerdem folgende Konvention:

- Variablen starten normalerweise mit einem Kleinbuchstaben (Großbuchstaben für Klassen)
- „Private“ Variablen starten mit „_“ (eventuell später mehr dazu)
- „Sprechende“ Variablennamen sind nichtssagenden Namen vorzuziehen (Beispiel: breite ist besser als x, hoehe besser als y)

Einfache Datentypen

Datentypen: Wahrheitswerte

Der einfachste Datentyp in Python speichert die Antwort auf eine einfache Ja/Nein-Frage (Ist eine Zahl gerade, eine bestimmte Person älter als 18 Jahre alt?). Dieser Datentyp heißt *Boolean* (siehe Boolesche Algebra) und kann die Werte `True` und `False` annehmen.

Python kennt die drei Booleschen Operatoren **`and`**, **`or`** und **`not`** um mit booleschen Werten zu „rechnen“:

`x and y`	`True`, falls x **und** y den Wert `True` haben
`x or y`	`True`, falls x **oder** y den **`Wert`** True haben
`not z`	`True`, falls z den Wert `False` hat

Hinweis: Die beiden Operatoren **`and`** und **`or`** sind sog. „Short Circuit“-Operatoren, d. h. wenn das Ergebnis nach dem ersten Vergleich feststeht, wird der zweite nicht mehr ausgewertet!

Datentypen: Zahlen

Python kennt vier Datentypen für Zahlen:

Name	Typ	Beispiele
Ganzzahlen	`int`	`5, 7`
Gleitkommazahlen	`float`	`1., 0.5, 3.14159`
„Lange“ Ganzzahlen	`long`	`5L, 1234L`
Komplexe Zahlen	`complex`	`7 + 3j, 8 - 100j`

Eine Umwandlung (zum Beispiel eine Berechnungen mit verschiedenen Typen, wie `3 + 4.5 = 8.5`) ist evtl. automatisch möglich:

$$\mathit{int} \rightarrow \mathit{long} \rightarrow \mathit{float} \rightarrow \mathit{complex}$$

Berechnungen mit Zahlen

Rechenart	Ganzzahlen	Fließkommazahlen
Addition	`> 7 + 3` 10	`> 7.0 + 3.5` 10.5
Subtraktion	`> 7 - 3` 4	`> 7.0 - 3.5` 3.5
Multiplikation	`> 7 * 3` 21	`> 7.0 * 3.5` 24.5
Division	`> 7 / 3` 2.3333333333333335	`> 7 / 3.5` 2.0
Ganzzahldivision	`> 7 // 3` 2	–
Rest	`> 7 % 3` 1	–
Potenzen	`> 7 ** 3` 343	`> 49.0 ** 0.5` 7.0

Syntactic Sugar für Berechnung

Falls eine Berechnung mit dem (Zahlen-)Wert einer Variable durchgeführt werden soll und das Ergebnis wieder der Variable zugeordnet werden soll, kann dies kürzer geschrieben werden:

```
# 'Normal'
my_int = 0
my_int = my_int + 5   # 5
my_int = my_int - 1   # 4
my_int = my_int / 2   # 2
my_int = my_int * 3   # 6
my_int = my_int % 5   # 1
```

```
# Äquivalent:
my_int = 0
my_int += 5   # 5
my_int -= 1   # 4
my_int /= 2   # 2
my_int *= 3   # 6
my_int %= 5   # 1
```

Vergleiche von Zahlen

Python definiert wie viele andere Sprachen folgende Vergleichsoperatoren für Zahlen:

`==`	Gleichheit
`!=`	Ungleichheit
`>`	Größer
`<`	Kleiner
`>=`	Größer oder gleich
`<=`	Kleiner oder gleich

Das Ergebnis eines Vergleichs ist immer ein Boolean. Bei Vergleichen zwischen verschiedenen Typen wird - falls möglich - ein Typecast in den „weiteren“ Typen vorgenommen (siehe oben).

"Hausaufgabe": Python als Taschenrechner

Führen Sie verschiedene Berechnungen in der IDLE durch, z. B.

- Addition $+$, Subtraktion $-$, Multiplikation $*$ und Division / bzw. //
- Divisionsrest („modulo"), % Potenzen $**$
- Vergleiche $==$, $!=$, $>=$, $>$, $<=$, $<$

Benutzen Sie dabei verschiedene Zahlen:

- Ganzzahlen, z. B. `5, 8, 1337, -477`
- Gleitkommazahlen, z. B. `1.0, 7.34, 0.5`
- Komplexe Zahlen, z. B. `7 + 3j, 8 - 4j`

Fallen Ihnen Besonderheiten auf?

Datentypen: String

Um zum Beispiel dem Benutzer eines Programms eine sinnvolle (Fehler-)Meldung geben zu können, muss ein Programm mit Zeichenketten („Strings“) umgehen können.

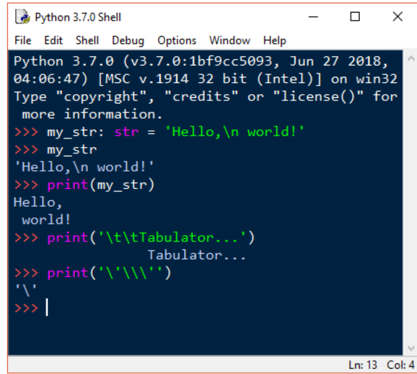
Dazu benutzt man in Python den Datentyp `str`:

```python
x = "Dies ist ein String"

y = 'Auch dies ist ein String'

long_string = """
Ein String mit drei Anführungszeichen darf sogar
über mehrere Zeilen geschrieben werden.
"""
```

Escapesequenzen

Zeichen in Strings, die mit „\“ eingeleitet werden, sind sogenannte „Escapesequenzen“ und werden gesondert interpretiert:

Zeichen	Wirkung
\n	Zeilenumbruch
\t	Tabulator einfügen
\’	Einfaches Anführungszeichen
\”	Doppeltes Anführungszeichen
\\	Einfacher Backslash (\)



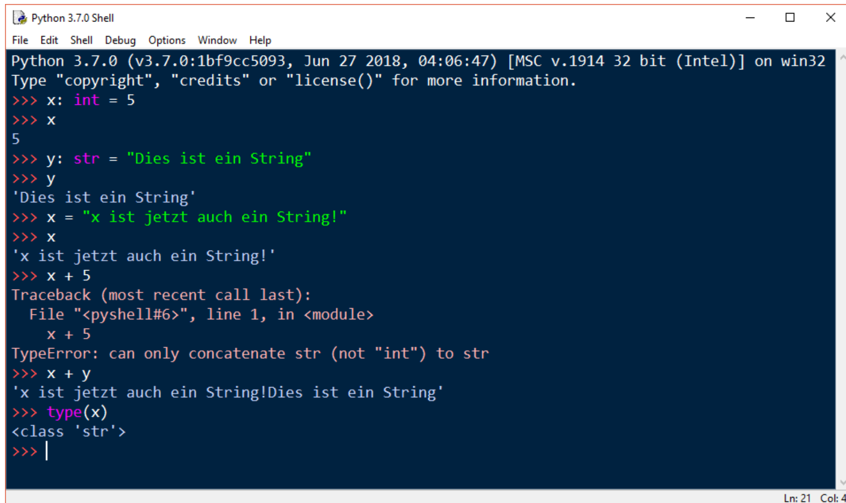
Type hints

In Python können Variablen mit einem sog. „Type hint" (Typhinweis) annotiert werden. Diese können – neben dem Variablennamen – weitere Hinweise über den Inhalt einer Variable liefern.

Im Gegensatz zu anderen Sprachen (wie z. B. Java) findet in Python **keine** Typprüfung statt, das heisst, man kann auch Werte mit anderen Typen in dieser Variable abspeichern

```python
x: int = 5

y: str = "Dies ist ein String"
```

Type hints - Nur Hinweise



Bedingungen

Bedingungen

Sehr oft werden bedingte Aufrufe im Code benötigt, zum Beispiel

- **Wenn** Kunde einen Hund besitzt, **dann** zeige Werbung für Hundefutter
- **Wenn** ein Spieler all seine Gegner bezwungen hat, **dann** beende das Spiel.
- **Falls** ein Student mehr als 50% der Punkte in der Klausur erzielt, **dann** lasse ihn bestehen, sonst lasse ihn durchfallen.
- ...

Bedingungen: Beispiele

Diese Anweisungen lassen sich direkt in Code übersetzen:

```
if client_has_dog:
    print("20 Prozent auf alles, auch auf Hundefutter!")

if number_of_opponents == 0:
        print("Sie haben das Spiel gewonnen!")

if points > 0.5 * maximal_points:
        print("Bestanden")
else:
        print("Nicht bestanden")
```

Bedingungen: Syntax

Verzweigungen in Python werden mit dem Operator `if` definiert, danach folgt die getestete Bedingung und ein Doppelpunkt.

Der Code innerhalb der Verzweigung wird ausgeführt, wenn die Bedingung mit „True“ evaluiert.

Der Code innerhalb einer Bedingung muss eingerückt werden!

Optional kann nach dem Schlüsselwort `else` Code angegeben werden, der ausgeführt wird, falls die Bedingung falsch ist.

```
if bedingung:
    # mache etwas
```

```
if bedingung:
    # mache etwas, wenn Bedinung wahr
else:
    # mache etwas, wenn Bedingung falsch
```

Verschachtelung von Bedingungen

Verzweigungen können beliebigen Code enthalten. Daher können auch Verzweigungen ineinander verschachtelt werden: $\Rightarrow$ „Entscheidungsbaum“

```
if dice_value == 1:
    print("Eins!")
else:
    if dice_value == 2:
        print("Zwei!")
    else:
        if dice_value == 3:
```

Verschachtelung mit `elif`

Das Schlüsselwort **`elif`** bildet eine Kombination aus **`else`** und **`if`**, und bezeichnet eine Alternative, die aber nur genommen wird, wenn eine zweite Bedingung zutrifft.
Es können mehrere **`elif`**s nacheinander vorkommen. Die Ausführung endet bei der ersten erfolgreichen Bedingung. Das folgende Codestücke ist äquivalent zu dem von letzter Folie:

```
dice_value = 5

if dice_value == 1:
    print("Eins gewürfelt!")
elif dice_value == 2:
    print("Zwei gewürfelt!")
elif dice_value == 3:
    print("Drei gewürfelt!")
elif dice_value == 4:
    print("Vier gewürfelt!")
elif dice_value == 5:
    print("Fünf gewürfelt!")
else:  # diceValue == 6
    print("Sechs gewürfelt!")
```

”Hausaufgabe”: Bedingungen

Schreiben Sie folgende Bedingungen:

1. Definieren Sie eine Variable `value` und testen Sie diese später mit den verschieden positiv und negativen Zahlen und mit 0. Schreiben Sie eine Bedingung, die Ausgibt, ob eine Zahl negativ, null, oder positiv ist.
2. Erstellen Sie eine Variable `punkte`.
    2.1 Schreiben Sie eine Bedingung, die „Gut gemacht“ ausgibt, wenn die Variable mindestens einen Wert von 80 hat. Setzten die diese Variable auf verschiedene Zahlen (0, 50, 80, 100) und überprüfen Sie, ob die Bedingung richtig ausgeführt wird.
    2.2 Schreiben Sie eine Bedingung, die die erreichte Note ausgibt. Die Verteilung startet bei 100 mit einem Punkteschritt von 10 Punkten.

Lösung: Bedingungen

```python
# Aufgabe 1
value = -2
if value < 0:
    print("Negativ")
elif value > 0:
    print("Positiv")
else:
    print("Null")

# Aufgabe 2.1
punkte = 80
if punkte >= 80:
    print("Gut gemacht")
```

```python
# Aufgabe 2.2
if 90 < punkte and punkte <= 100:
    print(1)
elif 80 < punkte:
    print(2)
elif 70 < punkte:
    print(3)
elif 60 < punkte:
    print(4)
elif 50 < punkte:
    print(5)
else:
    print(6)
```

Funktionen

Funktionen

Code, der an verschiedenen Stellen gebraucht wird, müsste theoretisch an jeder dieser Stellen stehen

Folgen: vor allem mehr Arbeit!

- Mehr Aufwand bei der Erstellung (Abschreiben mit Rechtschreibfehler!)
- Änderungen an diesem Code müssten an jeder Stelle gemacht werden, wobei leicht eine vergessen werden könnte
- Code wird unübersichtlicher

Daher: Verwendung von Funktionen!

Funktionen: Syntax

Funktionen werden mit dem Schlüsselwort `def` eingeleitet, danach folgen der Funktionsname, die Argumente (in Klammern) und ein Doppelpunkt.

```
def funktionsname(arg1, arg2, ...):
    # Funktionsrumpf
```

Der Funktionsrumpf steht **eingerückt** in der nächsten Zeile stellt die eigentliche Funktionalität dar. Er muss **konstant** eingerückt werden, ansonsten kann es zu Fehlern kommen, da die Einrückung Teil der Syntax ist!

Die Funktion endet, wenn eine andere Einrückung auftritt.

Der Aufruf einer Funktion erfolgt mit Angabe des Funktionsnamen und der Argumente.

Funktion: Beispiel

```python
# Name der Funktion: func
# Die Funktion hat keine Argumente
def func():
    # Hier beginnt der Funktionsrumpf
    print("Dies ist eine Funktion")

    # Leerzeilen und Kommentare werden ignoriert
    print("Auch dies ist noch Teil der Funktion")
    # Hier endet der Funktionsrumpf

# Einrückung "0", hier endet die Funktion
# Aufruf der Funktion mit Namen
func()
```

Rückgabewert(e) von Funktionen

- Mit dem Schlüsselwort `return` kann von einer Funktion **ein** Wert zurückgegeben werden.
- Die Ausführung der Funktion endet an der Stelle einer Rückgabe.
- Ein Return ohne einen Wert beendet die Funktion auch, gibt aber nichts (`None`) zurück.

```python
def test():
    print("Ausgeführt!")
    return True

    # print nicht ausgeführt
    print("Nicht ausgeführt!")


print(test())
# Ich werde ausgeführt!
# True
```

Funktionen: `print` versus `return`

- `print` Gibt ist eine Funktion und gibt etwas *auf der Konsole* aus
- `return` ist ein Schlüsselwort und gibt einen Wert *aus einer Funktion* zurück

```
def my_first_func():
    print(1234)

x = my_first_func()
# 1234
print(x)
# None
```

```
def my_second_func():
    return 1234

y = my_second_func()

print(y)
# 1234
```

Funktionsargumente

Um eine Funktion dynamisch zu machen, können ihr Argumente übergeben werden. Diese werden in den Klammern angeben und mit einem Komma getrennt.

```
def ausgabe(print1, print2, print3):
        if print1:
                print("1. Statement")
        if print2:
                print("2. Statement")
        if print3:
                return
        print("3. Statement")

ausgabe(True, False, True)
```

”Hausaufgabe”: Simple Funktion

Wie ist die Ausgabe des folgenden Programms?

Tippen Sie die Funktion ab, um das Ergebnis zu bestätigen.

Testen Sie verschiedene Parameterbelegungen (auch Zahlen oder Zeichenketten).

```
def ausgabe(print1, print2, print3):
    if print1:
        print("1. Statement")
    if print2:
        print("2. Statement")
    if print3:
        return
    print("3. Statement")
ausgabe(False, True, True)
```

Funktionen und Type hints

Funktionsargumente können wie Variable mit einem Type hint versehen werden. Ein Hinweis auf den Typen des Rückgabewertes einer Funktion kann mit der Syntax „`-> typ`“ nach den Argumenten angegeben werden.

```
def my_func(arg1: bool, arg2: int, arg3: str) -> int:
    if arg1:
        return -1
    elif arg3 == 'test':
        return arg2
    else:
        return 0
```

Weitere Beispiele Funktionen

Parabelgleichung: $y = a \cdot x^2 + b \cdot x + c$

```
def parabel(a: float, b: float, c: float, x: float) -> float:
    return a * x * x + b * x + c
```

Test, ob ein Wert x sich in einem Interval $[a; b]$ befindet:

```
def is_in_interval(x: int, a: int, b: int) -> bool:
    return a <= x and x <= b
```

Aufgabe: Mitternachtsformel

Gegeben ist die sogenannte Mitternachtsformel zur Berechnung der Nullstellen einer quadratischen Funktion $f(y) = ax^2 + bx + c$:

$$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Vervollständigen Sie folgende Funktion:

```python
def null_stellen(a: float, b: float, c: float):
    # TODO: Berechnung der Nullstellen x1, x2
    x1 = ...
    x2 = ...

    return x1, x2
```

Hinweis: Die Wurzel einer Zahl können Sie mit `zahl ** 0.5` berechnen.

Lösung: Mitternachtsformel

```python
def nullStellen(a: float, b: float, c: float):
    wurzel = ((b ** 2) - 4 * a * c) ** 0.5
    x1 = (-b + wurzel) / (2 * a)
    x2 = (-b - wurzel) / (2 * a)
    return x1, x2


print(nullStellen(-4, 4, 3))  # (-0.5, 1.5)
```

Default-Argumente

Falls eine Funktion Argumente besitzt, die sehr oft mit dem selben Wert belegt werden, kann das Argument mit einem Standardwert belegt werden. Damit kann die Funktion aufgerufen werden, ohne dieses Argument anzugeben und es wird der Standardwert verwendet.

```python
def greet(who: str, greetings: str = 'Hello, '):
    print(greetings + who)

greet('Monica')
# Hello, Monica

greet('Joey', 'Good to see you, ')
# Good to see you, Joey
```

Wiederholung: Lernziel

Wiederholung: Lernziel

```python
def biggest(a: int, b: int, c: int) -> int:
    current_biggest: int = a

    if b > a:
        current_biggest = b

    if c > current_biggest:
        current_biggest = c

    return current_biggest


print(biggest(4, 2, 8))
```