



Einführung

Syntax

Kontrollstrukturen

In & Output

Funktionen

Klassen

Module

Lambda

Datenbanken

Fehlerbehandlung

GUI

Tests

Einführung



Geschichte

- 1991 von Guido van Rossum entwickelt
- Inspiriert von C, C++, Java, Perl und Lisp
- Basiert auf C
- Sollte leicht zu lesen und mächtig sein
- Python 1.0 1994
- Python 2.0 2000
- Python 3.0 2008
- Aktuelle Version: 3.9.2

„Python is the most powerful language you can still read!“ – Paul Dubois

Python Kerneigenschaften

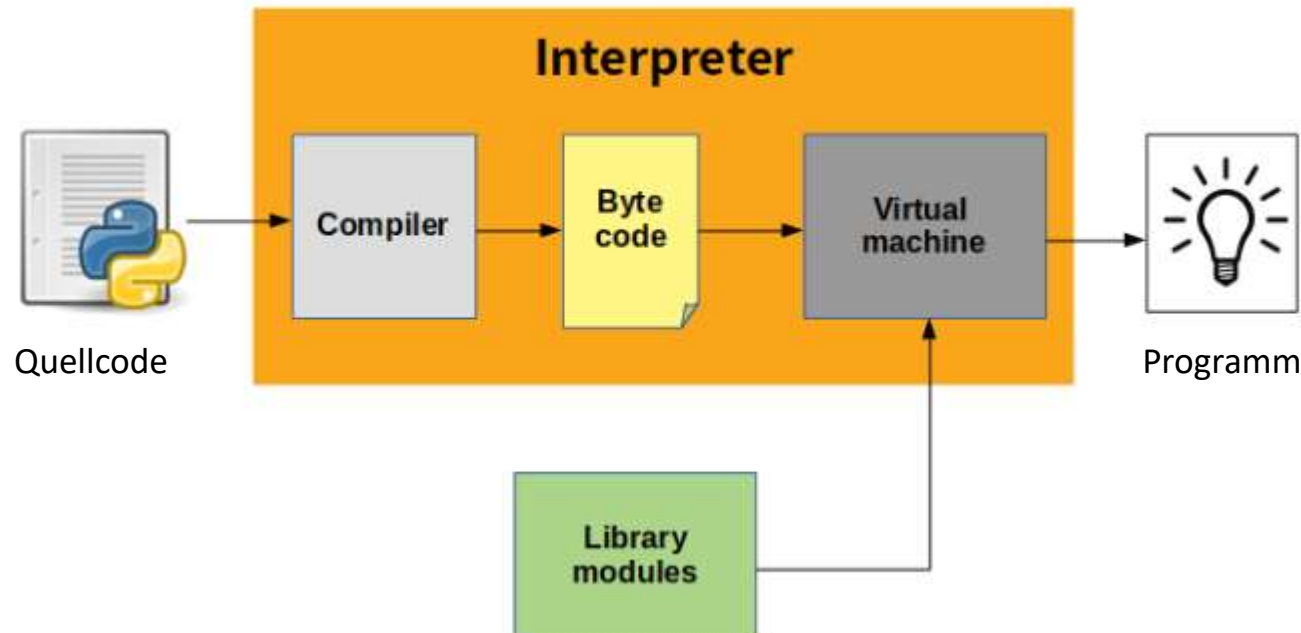
- Einfach zu lesen und zu schreiben
- Höhere Programmiersprache
- Dynamisch Typisiert
- Interpretersprache
- Crossplatform
- Open Source
- Objekt Orientiert
- Endung: .py

Python Anwendungsgebiete

- Webentwicklung mit Flask & Django
- Machine Learning mit Pandas, NumPy und Scikit-Learn
- Data Science mit Pandas und Numpy
- Embedded Systems
- CAD-Anwendungen mit Fandango
- Spieleentwicklung mit PyGame und PySoy

Python Interpretation

- Python-Code(.py) wird zu Byte-Code kompiliert(.pyc)
- Byte-Code wird von der Python-VirtualMachine ausgeführt



Programmierparadigmen in Python

- Funktional:
 - + einfaches Debuggen
 - + klare Struktur & gute Lesbarkeit
 - langsam
 - Schwer zu schreiben
- Prozedural:
 - + Wiederverwendbar
 - + Lesbarkeit
 - Ungeeignet für reale Projekte
- Objekt-Orientiert
 - + Vererbung & Wiederverwendbarkeit
 - + Lesbarkeit
 - Langsamer

IDEs

- Integrated Development Environment
- Kombiniert die wichtigsten Aufgabengebiete eines Entwicklers
- Erleichtert das Schreiben von Code durch visuelle und syntaktische Hilfen

Python IDEs

- [PyCharm](#)
- [Visual Studio](#)
- [Visual Studio Code](#)
 - [Einrichtung](#)



PyCharm

- Hersteller: JetBrains
- Zwei Varianten:
 - Community (Kostenlos)
 - Professional (Kostet)
- Auf Python zugeschnitten
 - Integrierte Python-Konsole
 - Integriertes Terminal
- VCS (Version Control System)
 - Git & GitHub



PyCharm-Shortcuts

Shortcut	Funktion
Strg + Umschalt + F10	Derzeit ausgewählte Datei ausführen
Shift + F9	Oben definierte Datei debuggen
Strg + /	Kommentar
Strg + Alt + L	Autoformatierung
2x Umschalt	Suche in allen Dateien
Alt + Enter	Kontextmenü anzeigen
F2/ Umschalt + F2	Zum nächsten/vorherigen Fehler springen
Umschalt + F6	Variable/Objekt umbenennen
Strg + K	Commit
Strg + Umschalt + K	Push



PyCharm Terminal

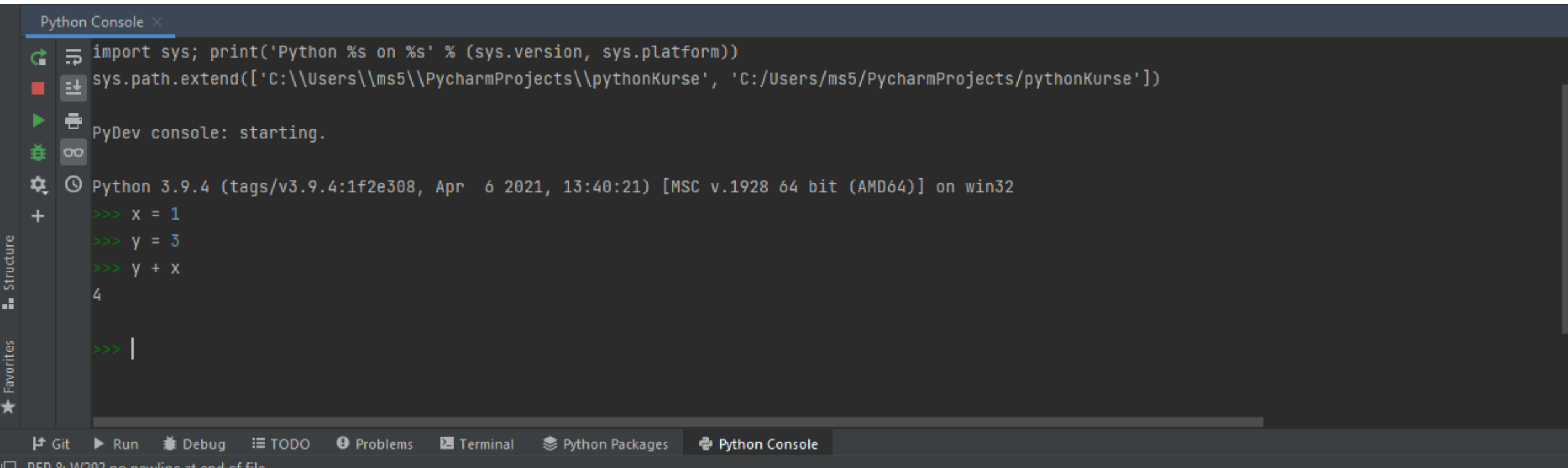
- Dateien im Terminal ausführen:
 - Reiter Terminal auswählen
 - Python {Dateiname.py} {Argumente}





PyCharm Python Konsole

- Erlaubt das schreiben von Python-Code sowie das direkte ausführen
- Nützlich zum testen von neuen Funktionen/Modulen etc.



The screenshot shows the PyCharm Python Console window. The console title bar is 'Python Console'. The left sidebar contains icons for Run, Debug, and other tools. The console output shows the following:

```
import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['C:\\Users\\ms5\\PycharmProjects\\pythonKurse', 'C:/Users/ms5/PycharmProjects/pythonKurse'])
PyDev console: starting.
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr 6 2021, 13:40:21) [MSC v.1928 64 bit (AMD64)] on win32
>>> x = 1
>>> y = 3
>>> y + x
4
>>> |
```

The bottom status bar shows various icons and text: Git, Run, Debug, TODO, Problems, Terminal, Python Packages, Python Console, and a file path: D:\PDP 2-W202 no newline at end of file.

Weiterführende Links

- Python Docs: [Our Documentation | Python.org](#)
- Python Neuigkeiten: [Our Blogs | Python.org](#)
- Python-Podcast: [Python Podcast \(python-podcast.de\)](#)
- Real-Python: [The Real Python Podcast – Real Python](#)



Einführung

Syntax

Kontrollstrukturen

In & Output

Funktionen

Klassen

Module

Lambda

Datenbanken

Fehlerbehandlung

GUI

Tests

Syntax & DatenTypen



Grundlagen

- Einschübe sind nicht optional => Ersetzen {}
- Semikolons werden nicht benötigt

test.py

```
1  print("Hallo!")
2  print("Auf Wiedersehen!")
3
4  if True:
5      print("klappt") # Braucht weder "{" und "}" noch das ";"
6  else:
7      print("Funktioniert nicht!") # Aber die Einschübe müssen stimmen
```

Variablen

- Werden bei Zuweisung eines Werts erstellt
- Müssen nicht mit Typ deklariert werden
- Können Typ auch nach Zuweisung eines Wertes ändern
- Können durch casting mit spezifischen Typ initialisiert werden

test.py > ...

```
1  x = 5 # Variable des Typs integer wird mit 5 initialisiert
2  y = "Ein interessanter Text" # Variable des Typs String
3  x = "Jetzt auch ein interessanter Text" # X ist nun ein String
4  z = str(4)
5  type(z) # Gibt den Typ von Z aus
6  print(type(z)) # Out: <class 'str'>
```

Kommentare

- Werden mit „#“ angegeben
- Müssen vor jeder Kommentarzeile eingefügt werden
- Alles hinter dem „#“ wird ignoriert

Best practice Tipp:

- Leerzeichen nach „#“
- Möglichst wenige Inline Kommentare
- Direkt nach Funktion/Methode/Klasse ein Kommentar mit kurzer Beschreibung

```
test.py > best_practice
1  # Das ist ein Kommentar
2  # Das ist ein
3  # mehrzeiliger Kommentar
4  x = 5 # Rest der Zeile wird ignoriert + 5
5  print(x) # out: 5
6
7  class best_practice():
8      # Beschreibe mich!
9      pass
```

Kommentare

- Werden mit „#“ angegeben
- Müssen vor jeder Kommentarzeile eingefügt werden
- Alles hinter dem „#“ wird ignoriert

Best practice Tipp:

- Leerzeichen nach „#“
- Möglichst wenige Inline Kommentare
- Direkt nach Funktion/Methode/Klasse ein Kommentar mit kurzer Beschreibung

```
test.py > best_practice
1  # Das ist ein Kommentar
2  # Das ist ein
3  # mehrzeiliger Kommentar
4  x = 5 # Rest der Zeile wird ignoriert + 5
5  print(x) # out: 5
6
7  class best_practice():
8      # Beschreibe mich!
9      pass
```

Datentypen

Typ	Kategorie	Typ	Kategorie
<u>str</u>	Text-Typ	<u>dict</u>	Mapping
<u>int</u>	Numeric	<u>list</u>	Sequence
<u>float</u>	Numeric	<u>tuple</u>	Sequence
<u>complex</u>	Numeric	<u>range</u>	Sequence
<u>bool</u>	Boolean	<u>set</u>	Set

String

- Text-Variablen
- Substrings durch slicing `string[<start>:<ende>:<schrittweite>]`
- f-string:
 - Deklaration: `<variable> = f„<text> {andere variable/ Ausdruck}“`
 - Formated-String
- Verfügen über mehrere Methoden

String Methoden

Methode	Funktion
<code><string>.count(<Ausdruck>)</code>	Zählt, wie oft der gesuchte Ausdruck, vorkommt
<code><string>.index(<Ausdruck>)</code>	Zeigt den Index, an dem der gesuchte Ausdruck zum ersten Mal auftritt
<code><string>.isalpha/isnumeric/isalnum()</code>	Gibt True/False aus, falls String nur aus Buchstaben/Zahlen oder nur aus Buchstaben und Zahlen besteht
<code><string>.title()</code>	Anfangsbuchstaben jedes Wortes werden groß
<code><string>.capitalize()</code>	Anfangsbuchstabe wird groß
<code><string>.lower()</code>	Alle Buchstaben werden klein
<code><string>.upper()</code>	Alle Buchstaben werden groß
<code><string>.l/rstrip()</code>	Entfernt anführenden/anhängende Leerzeichen
<code><string>.split(<Ausdruck>)</code>	Spaltet den Text anhand des gewählten Ausdrucks auf und gibt eine Liste der Ergebnisse aus
<code><string>.replace(<A>,)</code>	Ersetzt alle Vorkommnisse von <A> durch

Arithmetische Operatoren

Operator	Name	Beispiel
+	Addition	$1+1 = 2$
-	Subtraktion	$1-1 = 0$
*	Multplikation	$2*2 = 4$
/	Division	$15 / 7 = 2,14...$
%	Modulus	$15 \% 7 = 1$
**	Potenzierung	$2 ** 3 = 8$
//	Ganzzahl-Division	$15 // 7 = 2$

Module

- Module sind Code-Bibliotheken
- Können mit **import** importiert werden
- Entweder ganzes Modul oder einzelne Subelemente davon
- Alias kann zur einfacheren Benutzung definiert werden
- Können selbst erstellt werden

 test.py

```
1  import __hello__ # importiert das Modul __hello__
2  import tkinter as tk # Importiert das tkinter Modul und gibt ihm das Alias tk
3  from sys import argv # Importiert das Submodul argv vom Modul sys
```

Logische & identitäts Operatoren

Operator	Beispiel	Funktion
and	$5 < 10$ and $1 < 2$ \Rightarrow True	Gibt „True“ zurück, falls beide Angaben wahr sind
or	$4 > 1$ or $10 < 5$ \Rightarrow True	Gibt „True“ zurück, falls eine der Angaben wahr ist
not	$\text{not}(4 > 1 \text{ or } 10 < 5)$ \Rightarrow False	Kehrt Ergebnisse um \Rightarrow or beide Aussagen falsch and eine der beiden Aussagen falsch
is	$x = 1$ $y = 1$ $x \text{ is } y \Rightarrow$ True	Gibt „True“ zurück, falls die beiden Objekte gleich sind
is not	$x = 1$ $y = 2$ $x \text{ is not } y \Rightarrow$ True	Gibt „True“ zurück, falls die beiden Objekte nicht gleich sind
in	$x = 1$ $y = [1,2,3]$ $x \text{ in } y \Rightarrow$ True	Gibt „True“ zurück, falls das eine Objekt im anderen enthalten ist
not in	$x = 4$ $y = [1,2,3]$ $x \text{ not in } y \Rightarrow$ True	Gibt „True“ zurück, falls das Objekt nicht im anderen enthalten ist

Lists

- Können mehrere Werte in einer Variablen speichern
- Sind geordnet, d.h. sie haben eine feste Reihenfolge, die sich (fast) nie ändert.
- Sind veränderbar, d.h. es können neue Elemente hinzugefügt und bestehende entfernt werden
- Duplikate sind erlaubt
- Können verschieden Datentype enthalten
- Konstruktor: *list()*

```
1  meineListe = ['Kann', 'verschiedene', True, 'Typen', 1234, 'Enthalten']
2  print(meineListe[1]) # Out: verschiedene => Beginnen mit Index 0
3  print(meineListe[-2]) # Out: 1234
4  print(meineListe[1:5]) # Out: ['verschiedene', True, 'Typen', 1234]
```

Lists

Methode	Funktion
append()	Fügt Element am Ende der Liste hinzu
clear()	Entfernt alle Elemente aus der Liste
copy()	Gibt eine Kopie der Liste zurück
count()	Gibt Anzahl der Elemente mit angegebenen Wert zurück
extend()	Fügt das Element oder das iterierbare Objekt (z.B. Liste) am Ende der Liste an
index()	Gibt den Index des gegebenen Elements zurück
insert()	Fügt Element an gegebenen Index ein
pop()	Entfernt das Element am angegebenen Index
remove()	Entfernt das Element mit dem angegebenen Wert
reverse()	Kehrt die Reihenfolge der Liste um
sort()	Sortiert die Liste (Standard: Alphanumerisch absteigend)

Tuples

- Können mehrere Werte in einer Variablen speichern
- Sind geordnet, d.h. sie haben eine feste Reihenfolge, die sich (fast) nie ändert
- Sind nicht veränderbar, d.h. es können keine neuen Elemente hinzugefügt oder bestehende entfernt werden
- Duplikate sind erlaubt
- Können verschieden Datentype enthalten
- Konstruktor: *tuple()*

```
1  meinTupel = ('Hello', 'there!') # !Runde statt eckige Klammern
2  print(meinTupel) # Out: ('Hello' , 'there!')
3  x = ('General', 'Kenobi')
4  y = list(x) # Kopiert das Tupel x in die Liste y
5  y[1] = 'Patton'
6  x = tuple(x) # Umgeht die Einschränkung, dass Tupel nicht verändert werden können
```

Tuples

Methode	Funktion
count()	Gibt aus wie oft das gegebene Element im Tupel vorkommt
index()	Gibt die Index-Position des gegebenen Elements aus

```
1  Tiere = ('Hund', 'Katze', 'Maus')
2  (groß, mittel, klein) = Tiere # Tuple wird "ausgepackt"
3  print(groß) # Out: Hund
4  print(mittel) # Out: Katze
5  print(klein) # Out: Maus
6
7  Tupel1 = (1,2,3)
8  Tupel2 = (4,5,6)
9  Tupel3 = Tupel1 + Tupel2 # Kombiniert die beiden Tupel
10 print(Tupel3) # Out: (1, 2, 3, 4, 5, 6)
```

Range

- Nichtveränderbare Sequenz von Integern
- Inklusive Startzahl
- Exklusive Endzahl
- Schrittweite ist optional

```
11  range(ende)
12  range(100) # Zahlen von 0 - 99
13
14  range(start, ende, schrittgröße) # Schrittgröße ist optional, Standardwert = 1
15  range(1, 101, 1) # Alle Zahlen von 1 - 100
16
```

Dictionaries

- Speichern Key:Value Paare
- Sind geordnet
- Es können neue Elemente hinzugefügt und entfernt sowie vorhandene verändert werden
- Duplikate sind nicht erlaubt

```
1  meinAuto = {  
2      "Marke": "Audi",  
3      "Modell": "R8",  
4      "Baujahr": 2019  
5  }  
6  
7  print(meinAuto["Marke"]) # Out: "Audi"  
8  meinAuto["Marke"] = "VW" # Funktionieren identisch  
9  meinAuto.update({"Marke": "Vw"})
```


Dictionaries

Methode	Funktion
<code>clear()</code>	Entfernt alle Elemente des Dictionarys
<code>copy()</code>	Gibt eine Kopie des Dictionarys zurück
<code>get()</code>	Gibt das Value des gegebenen Keys zurück
<code>items()</code>	Gibt alle Key:Value Paare als Tupel zurück („Key1“, „Value1“), usw.
<code>keys()</code>	Gibt alle Keys als Liste zurück
<code>pop()</code>	Entfernt das Key:Value Paar mit dem angegebenen Key
<code>popitem()</code>	Entfernt das letzte Key:Value Paar
<code>setdefault()</code>	Gibt den Wert des angegebenen Keys zurück, falls er noch nicht existiert wird er mit dem angegebenen Wert angefügt
<code>update()</code>	Setzt den Wert des Keys auf den angegebenen Wert
<code>values()</code>	Gibt eine Liste aller Values aus

Sets

- Können mehrere Werte in einer Variablen speichern
- Sind ungeordnet, d.h. nicht indexiert
- Es können Elemente hinzugefügt und entfernt werden, aber vorhandene Elemente können nicht geändert werden
- Duplikate sind nicht erlaubt
- Können verschieden Datentype enthalten
- Konstruktor: `set()`

```
1  meinSet = {'Funk', 'tioniert'}
2  meinSet2 = {'das', 'das', 'was'}
3  print(meinSet2) # Out: { 'das', 'was' } Duplikate werden nicht gespeichert
4  meinSet2.add('was') # Kein Fehler, aber verändert das Set nicht
5  print(meinSet2) # Out: { 'das', 'was' }
```

Sets

Methode	Funktion
<code>add()</code>	Fügt Element am Ende des Sets hinzu
<code>clear()</code>	Entfernt alle Elemente aus dem Set
<code>copy()</code>	Gibt eine Kopie des Sets zurück
<code>difference()</code>	Gibt ein Set zurück, das aus den nicht geteilten Elementen von 2 oder mehr Sets, besteht
<code>difference_update()</code>	Entfernt die Elemente des Sets, die auch in dem anderen enthalten sind
<code>discard()</code>	Entfernt das angegebene Element
<code>intersection()</code>	Gibt ein Set aus, das aus den Gemeinsamkeiten der gegebenen Sets besteht
<code>intersection_update()</code>	Entfernt die Elemente, die nicht in dem anderen Set enthalten sind
<code>isdisjoint()</code>	Gibt True aus, falls keins der Elemente in dem anderen Set enthalten sind
<code>issubset()</code>	Gibt True aus, falls das Set in dem anderen Set enthalten ist
<code>issuperset()</code>	Gibt True aus, falls das Set das andere Set beinhaltet
<code>pop()</code>	Entfernt ein zufälliges Element aus dem Set und gibt es aus
<code>remove()</code>	Entfernt das angegebene Elemente aus dem Set. Gibt Fehlermeldung zurück, falls das Element nicht im Set enthalten ist
<code>union()</code>	Gibt ein Set zurück, das die gegebenen Sets kombiniert
<code>update()</code>	Fügt noch nicht vorhandene Elemente in das gegebene Set ein

Übung:

1. Hello World

- Erstelle eine neue Python-Datei mit dem Namen „Lab1.py“
- Definiere einen beliebigen String und gebe ihn in der Konsole aus

2. Teilnehmer-Collection

- Erstelle eine Beliebige Collection mit den Vor- und Nachnamen der Teilnehmer
- Füge einen neuen Eintrag zur Collection hinzu und lasse nur diesen in der Konsole ausgeben

3. Einfache Arithmetik

- Definiere jeweils zwei Integer und Floats
- Führe Addition, Division, Floordivision, Potenzierung und Multiplikation durch
- Lasse die Ergebnisse jeweils in der Konsole ausgeben



Einführung

Syntax

Kontrollstrukturen

In & Output

Funktionen

Klassen

Module

Lambda

Datenbanken

Fehlerbehandlung

GUI

Tests

Kontrollstrukturen



if-Anweisungen

- Werden in Kombination mit logischen Operatoren benutzt um komplexere Programmabläufe zu ermöglichen
- Vergleichen die gegebenen Bedingungen und führen dementsprechend Anweisungen aus
- Können alleine stehen
- Syntax: **if** *Bedingung*:

```
1  a = 1
2  b = 2
3
4  if a > b: # Falls a größer ist als b
5      print('a ist größer') # Gebe das aus
6
```

elif-Anweisungen

- Müssen nach einem if stehen
- Werden nur beachtet, falls das voranstehende if nicht eintritt
- Ansonsten selbe Funktion wie if
- Syntax: **elif** *Bedingung*:

```
1  a = 1
2  b = 2
3
4  if a > b: # Falls a größer ist als b
5      print('a ist größer') # Gebe das aus
6  elif a < b: # Falls a kleiner ist als b
7      print('a ist kleiner') # Gebe das aus
8
```


else-Anweisung

- Müssen nach einem if oder elif stehen
- Werden nur beachtet, falls die voranstehenden Bedingungen nicht erfüllt werden
- Syntax: **else** :

```
1  a = 1
2  b = 2
3
4  if a > b: # Falls a größer ist als b
5      print('a ist größer') # Gebe das aus
6  elif a < b: # Falls a kleiner ist als b
7      print('a ist kleiner') # Gebe das aus
8  else: # Falls a weder größer noch kleiner als b ist
9      print("a ist gleich groß wie b") # Gebe das aus
10
```

Kurzschreibweise & Ternär Operator

- Falls man nur einen Ausdruck für `if` bestimmt kann man es auf eine Zeile kürzen

```
11  if a > b: print('a ist größer als b')
```

- Falls man nur einen Ausdruck für `if` und einen für `else` hat, kann man es auch auf eine Zeile kürzen

```
11  print('a ist größer als b') if a > b else print('a ist kleiner als b')
```

- Ternärer Operator erlaubt das kürzen multipler `if` Anweisungen auf eine Zeile

```
11  print('a ist größer als b') if a > b else print('a ist gleich groß wie b') if a == b else print('a ist kleiner als b')
12  # Das selbe Statement wie oben, aber auf eine Zeile gekürzt
```

- Ist zwar kürzer, aber meist schwerer zu lesen und sollte bei komplexeren Anweisungen vermieden werden

Verschachtelte if-Bedingungen

- Funktionieren wie reguläre if-Anweisungen
- Erlauben die genauere Steuerung des Programmflusses

```
15  if a < b:
16      print('a ist kleiner als b')
17      if a % 2 == 0:
18          print('a ist gerade')
19      else:
20          print('a ist ungerade')
21  elif a > b:
22      print('a ist größer als b')
23      if a % 2 == 0:
24          print('a ist gerade')
25      else:
26          print('a ist ungerade')
27  else:
28      print('a und b sind gleich groß')
29      if a % 2 == 0:
30          print('a ist gerade')
31      else:
32          print('a ist ungerade')
```

while-Schleifen

- Führen Anweisungen aus solange die Bedingung wahr ist
- Sind kopfgesteuert, d.h. sie kann übersprungen werden, wenn die Bedingung nie eintritt
- Können mit break vorzeitig abgebrochen werden
- Einzelne Schritte können mit continue übersprungen werden

```
14 i = 0
15 while i < 100:
16     if i == 97: # Falls i 97 ist wird die Schleife vorzeitig beendet
17         break
18     i += 1
19     if i % 10 == 0: # Falls i ohne Rest durch 10 teilbar ist, wird die ausgabe übersprungen
20         continue
21     print(i)
22
```

while-Schleifen

- Können mit else Bedingungen kombiniert werden um Ausgabe nach Ende der Schleife zu ermöglichen
- Mit einem if kann while von einer kopf- zu einer fußgesteuerten Schleife umgewandelt werden

```
14 i = 0
15 while i < 10:
16     print(i)
17     i += 1
18 else:
19     print(f'i ist jetzt: {i}')
20
21 x = 10
22 while True:
23     print(x)
24     x = +1
25     if x < 10:
26         break
27 # Out: 10 obwohl x von Anfang an nicht kleiner als 10 ist
```

for-Schleifen

- Iteriert über eine Sequenz(List, tuple,dictionary,set oder string)
- Die break und continue Anweisungen funktionieren wie bei while

```
30 list = ('Luke', 'ich', 'bin', 'dein', 'Vater')
31
32 for x in list:
33     print(x)
34 else:
35     print('Neeeeeein!')
36
37 for x in list:
38     if x == 'Luke':
39         continue
40     if x == 'dein':
41         break
42     print(x)
43 else:
44     print('dein Trainer') #! Wird nicht ausgeführt, da das break die dazugehörige Schleife beendet
45
```

Verschachtelte for-Schleifen

- Nützlich um über mehrdimensionale Sequenzen zu iterieren
- Die zweite for-Schleife wird komplett durchiteriert bevor das nächste Element der ersten Sequenz drankommt

```
46 nestedList = (('Andere', 'Film', 'zitate'), ('Und', 'ein', 'paar', 'Easter', 'eggs'))
47
48 for list in nestedList:
49     for element in list:
50         print(element)
```

Übung

1. Das kleine Einmaleins

- Erstelle eine neue Datei namens „L03.py“
- Schreibe eine Schleife, die dir das Einmaleins von 1 bis 10 berechnet und, die dir jeden Schritt in der Konsole anzeigt
- Optional: Erweitere die Schleife auf das große Einmaleins

2. Kardinal zu Ordinal:

- Schreibe eine Schleife die dir von den zahlen von 1 bis 100 jeweils die kardinale und die ordinale darstellt (Zahl + Endung ‚st‘, ‚nd‘, ‚rd‘ oder ‚th‘)

3. FizzBuzz

- Erstelle eine Schleife, die die Zahlen von 1 bis 100 auf ihre Teilbarkeit durch 3 und 5 prüft.
- Falls die Zahl durch 3 teilbar ist, soll die Konsole „Fizz“, falls sie durch 5 teilbar ist „Buzz“, falls sie durch 3 und 5 teilbar ist „Fizzbuzz“ und falls sie durch keine von beiden teilbar ist eifnach die Zahl wiedergeben.



Einführung

Syntax

Kontrollstrukturen

In & Output

Funktionen

Klassen

Module

Lambda

Datenbanken

Fehlerbehandlung

GUI

Tests

In- & Output



input()

- Ermöglicht Benutzern Informationen einzugeben
- Enthaltener Text wird dem Benutzer vor der Eingabe angezeigt
- Standardmäßig vom string-Typ
- Typ kann durch casting verändert werden

m004-In&Output >  m004.py > ...

```
1  name = input('Wie ist dein Name?: \n')
2  print(f'Hallo {name}!')
3
4  x = input('Gib einen Integer ein: \n')
5  y = input('Gib einen zweiten Integer ein: \n')
6  print(f'{x} + {y} = {x+y}') # Ergebnis 'xy', da x und y strings sind
7
8  x = int(input('Gib einen Integer ein: \n'))
9  y = int(input('Gib einen zweiten Integer ein: \n'))
10 print(f'{x} + {y} = {x+y}') # Nun das erwartete Ergebnis, da beide Werte integer sind
11
```

open()

- Öffnet die angegebene Datei im ausgesuchten Modus
- ,r' für read, Standardmodus, falls nur Datei angegeben wird
- ,w' für write, falls die Datei nicht existiert wird sie erstellt
- ,a' für append
- ,r+' oder ,w+' für read&write, falls die Datei nicht existiert wird sie erstellt
- Wird standardmäßig im Text-Modus geöffnet, kann mit ,b' in binären Modus geöffnet werden
- Binärer Modus eignet sich z.B. für Bild-Dateien

```
12 f = open('m004-In&Output\\test.txt', 'r') # Öffnet die Datei test.txt im 'read' Modus
13 for line in f: # iteriert über jede eingelesene Zeile
14     print(line) # Gibt die einzelnen Zeilen in der Konsole aus
15 f.close() # Schließt die Datei 'test.txt' wieder
```

close()

- Schließt die geöffnete Datei wieder
- Geschlossene Dateien können nicht mehr gelesen oder beschrieben werden
- Datei wird automatisch geschlossen, falls die Referenz einer anderen Datei zugewiesen wird
- Geöffnete Dateien sollte immer am Ende geschlossen werden um Fehler zu vermeiden

Das with-Statement

- Öffnet die Datei mit dem Variabel-Namen hinter dem as
- Schließt die Datei wieder, nach Ende des with Blocks => kein `newFile.close()` nötig
- Verhindert mögliche Fehler

```
17  ∨ with open('m004-In&Output\\test2.txt', 'w+') as newFile:
18  ∨     for i in range(100):
19  ∨         newFile.write(f'Zeile {i} \n')
20
```

Module

- Es existieren eine Vielzahl an Modulen für Dateibehandlung
- Meistens auf ein Dateiformat abgestimmt
- Wichtige:
 - [PIL \(Python Imaging Library\) für Bilder](#)
 - [csv für .csv Dateien](#)
 - [json für json Dateien](#)
 - [Os für Interaktion mit dem Betriebssystem](#)

Das os-Modul

- Enthalten in der Python Standard Library
- Ermöglicht Interaktion mit dem Betriebssystem
- Die häufigsten Befehle:

Befehl	Funktion
<code>os.listdir({pfad})</code>	Gibt Liste von Inhalten des angegebenen Pfades zurück
<code>os.mkdir({Pfad})</code>	Erstellt einen Ordner mit dem Pfadnamen, wirft Fehler, falls dieser bereits existiert
<code>os.path.exists({Pfad})</code>	Gibt zurück ob die Datei am spezifizierten Pfad existiert

Das sys.argv-Modul

- Enthalten in der Python Standard Library
- Ermöglicht das Benutzen von Commandline-Parametern
- Ermöglicht es Skripte je nach Parameter auszuführen

```
Terminal: Local x +
(venv) C:\Users\ms5\PycharmProjects\pythonKurse>python argvFile.py
Es muss genau ein Parameter angegeben werden!

(venv) C:\Users\ms5\PycharmProjects\pythonKurse>python argvFile.py m
Ungültiger Parameter bitte wähle einen der folgenden:

r
w
c

(venv) C:\Users\ms5\PycharmProjects\pythonKurse>python argvFile.py r
Programm ist im Read-Mode

(venv) C:\Users\ms5\PycharmProjects\pythonKurse>python argvFile.py w
Programm ist im Write-Mode

(venv) C:\Users\ms5\PycharmProjects\pythonKurse>python argvFile.py c
Programm ist im Copy-Mode
```

Übung

- 1. Taschenrechner:
 - Erstelle ein Programm, das zwei Integer abfragt
 - Gib dem Benutzer die Möglichkeit per Tastendruck zwischen Addition, Subtraktion, Multiplikation und Division zu wählen.
 - Bei ungültiger Eingabe soll der Benutzer erneut nach seiner Entscheidung gefragt werden
 - Lasse das Ergebnis inklusive der Rechnung in der Konsole ausgeben

Übung

- 2. Logbuch:
 - Erstelle ein Programm, das prüft, ob die logbuch.txt bereits existiert
 - Falls nein, soll die Datei neuerstellt werden und der Benutzer nach seinem ersten Eintrag gefragt werden
 - Falls ja, soll der Benutzer direkt nach Eintrag gefragt werden und dieser an die bestehenden Einträge angefügt werden
 - Das Programm soll den Benutzer nach jeder Zeile fragen, ob ein weiterer erstellt werden soll und sich nur beenden, falls der Nutzer dies ablehnt.



Einführung

Syntax

Kontrollstrukturen

In & Output

Funktionen

Klassen

Module

Lambda

Datenbanken

Fehlerbehandlung

GUI

Tests

Funktionen



Grundlagen

- Ermöglichen das Wiederverwenden von Code
- Können optional Input (s.g. Parameter) erhalten und Werte zurückgeben
- Vereinfachen die Organisation und das Testen
- DRY (Don't Repeat Yourself), bei Mehrfachverwendung von Code => Funktion

Definieren neuer Funktionen

- Werden mit dem **def** Keyword kreiert
- Werden über den Namen der Funktion aufgerufen
- Können Parameter erhalten, s.g. Positional Parameters
- Übergebene Werte werden der Reihe nach festgelegt
- **return** gibt einen Wert zurück

```
1  def meineFunktion():
2      print('Hallo!')
3
4  def Addieren(x, y): # Der erste Parameter ist immer x
5      return x + y
6
7  meineFunktion()
8  print(Addieren(2, 98))
9  x = Addieren(14, 7) # Das Ergebnis der Funktion wird zum Wert von x
10 print(x)
11 Addieren(2,4,5) # Wirft einen Fehler, da zu viele Argumente übergeben wurden
```

Rekursive Funktionen

- Eine Funktion die sich in ihrem Körper selbst aufruft
- Besteht aus einer Basiskondition und dem rekursiven Teil
- In Python auf 1000 Schritte begrenzt

```
def Factorial(integer):  
    if integer == 1:  
        return 1  
    else:  
        return (integer * Factorial(integer - 1))
```


Arbitrary Arguments

- Werden mit ***** gekennzeichnet
- Erlaubt es eine unbekannte Anzahl an Argumenten zu erhalten
- Behandelt übergebene Parameter als Tuple
=> Innerhalb der Funktion sollte iteriert werden
- In den Python docs als ***args** abgekürzt

```
14 def Addieren(*numbers):#
15     sum = 0
16     for i in numbers:
17         sum += i
18     return sum
19
20 print(Addieren(1,2,3,4,5,6,7,8,9,10.122,123))
21 print(Addieren(1,2,91223123,213)) # Egal wie viele Parameter übergeben werden
```

Keyword Arguments

- Erlaubt das Übergeben von Parametern mit key = value Verhalten
- Parameter können dann in willkürlicher Reihenfolge übergeben werden
- In den Python Docs als kwargs abgekürzt

```
23 def Keywords(ParameterX, ParameterY, ParameterZ):
24     print(f'1: {ParameterZ}, 2: {ParameterX} und 3: {ParameterY}')
25
26 Keywords(ParameterY = 'ABC', ParameterZ = 123, ParameterX = 'Willkür')
27 # Out: 1: 123, 2: Willkür und 3: ABC
```

Arbitrary Keyword Arguments

- Funktioniert wie Arbitrary Arguments, aber akzeptiert benannte Argumente
=> Beliebig viele **benannte** Argumente in beliebiger Reihenfolge
- Python behandelt `**kwargs` wie ein Dictionary
- In Python Docs als `**kwargs` abgekürzt

```
29  ✓ def ArbKeyArg(**words):  
30  ✓     for key, value in words.items():  
31  |         print(f'{key} : {value}')
```



```
33  ArbKeyArg(Name = 'Dude', Alter = 27, Alles = 'ist möglich', Zahl = 8779.982, ABC = 'blabla')
```

* und ** Operatoren

- Sogenannte unpacking Operatoren
- * funktioniert bei jedem iterierbaren Objekt
- Teilt die Objekte in ihre einzelnen Wert auf
- ** nur bei Dictionaries
- Teilt das dictionary in seine Key : value Paare

```
35  digits = (1,2,3,4,5,6)
36  print(Addieren(*digits))
37  print(Addieren(digits)) # Muss zwingend * verwenden
38
39  dict = {'key1' : 1, 'key2' : 2, 'key3' : 3}
40  print(ArbKeyArg(**dict))
41  print(ArbKeyArg(*dict)) # Muss zwingend ** verwenden
```

Default-Arguments

- Beim definieren der Funktion festgelegt
- Werden benutzt, falls kein Argument übergeben wird
- Werden überschrieben, falls Argument übergeben wird

```
43  ✓ def defaultValue(wert = 0):  
44      print(wert)  
45  
46  defaultValue() # out: 0  
47  defaultValue(123)
```

Parameter-Reihenfolge

1. Positional/Keyword-Arguments
2. Default-Arguments
3. Arbitrary-Arguments
4. Arbitrary Keyword-Arguments

```
49  ▾ def orderFunction(positionals, default = 0, *args, **kwargs):  
50      |     pass
```

Namespaces

- Eine Art Wörterbuch, die Namen einem Objekt zuweisen
- Vier Namespace-Stufen existieren:
 1. Built-in (Existiert bis zum Ende des Programms/Skripts)
 2. Global (Existiert bis zum Ende des Programms/Skripts)
 3. Enclosing (Wird beim Aufrufen einer Funktion erstellt und danach geschlossen)
 4. Local (Namespace der derzeit aufgerufenen Funktion)
- Erlaubt Mehrfachverwendung von Variabel- und Funktionsnamen

Scope

- Der Bereich in dem Variablen aufrufbar sind und Namen eine Bedeutung haben
- Gelten immer nur in dem Block in dem sie erstellt wurden
- Innerhalb von Funktionen können globale Variablen mithilfe des `global` Keywords verändert werden
- LEGB-Regel (Namensauflösungsreihenfolge):
 1. Local-Namespace
 2. Enclosing-Namespace
 3. Global-Namespace
 4. Built-in-Namespace
- Dictionary von globalen Variablen kann mittels `globals()` wiedergegeben werden

Übung

1. Größter Wert

- Erstelle eine Funktion, die eine beliebig große Liste aus Integern als Parameter akzeptiert und den größten Wert ausgibt

2. Rückwärts

- Erstelle eine Funktion, die einen String als Parameter akzeptiert
- Optional: Lasse die Funktion den Benutzer nach einem String fragen

3. Klein-/Großzählen

- Erstelle eine Funktion die einen String akzeptiert und ausgibt wie viele Klein- und Großbuchstaben enthalten sind

4. Fibonacci-Folge

- Erstelle eine Funktion, die die Fibonacci-Folge bis zu gegebenen Stelle wiedergibt