Inhaltsverzeichnis

[1. Kommandos und Programme 3](#_Toc98767220)

[a. Interaktive Shell 3](#_Toc98767221)

[2. Programme schreiben 3](#_Toc98767222)

[3. Bytecode und Maschinencode 3](#_Toc98767223)

[4. Datentypen und Variablen 4](#_Toc98767224)

[a. Referenzieren zweier Variablen überprüfen: id() und „is“ 4](#_Toc98767225)

[b. Datentyp: Binär 4](#_Toc98767226)

[c. Komplexe Zahlen 5](#_Toc98767227)

[5. Sequenzielle Datentypen 5](#_Toc98767228)

[a. Strings 5](#_Toc98767229)

[a.1. Escape-Zeichen und Raw-String 6](#_Toc98767230)

[b. Listen 7](#_Toc98767231)

[c. Tupel 7](#_Toc98767232)

[c.1. Tupel entpacken 7](#_Toc98767233)

[6. Dictionaries 7](#_Toc98767234)

[a. Definition 7](#_Toc98767235)

[b. Fehlerfreie Zugriffe auf Dictionaries 7](#_Toc98767236)

[c. Zulässige Type für Schlüssel und Werte 8](#_Toc98767237)

[d. Verschachtelte Dictionaries 8](#_Toc98767238)

[e. Methode auf Dictionaries 8](#_Toc98767239)

[f. Operatoren auf Dictionaries 9](#_Toc98767240)

[g. Dictionaries aus Listen erzeugen 9](#_Toc98767241)

[7. Mengen 9](#_Toc98767242)

[a. Deklaration 10](#_Toc98767243)

[b. Operationen auf „set“-Objekten 10](#_Toc98767244)

[8. Verzweigungen 11](#_Toc98767245)

[9. Schleifen 11](#_Toc98767246)

[a. While 11](#_Toc98767247)

[b. For- Schleife 11](#_Toc98767248)

[10. Module importieren 11](#_Toc98767249)

[11. Eingaben 12](#_Toc98767250)

[12. Dateien lesen und schreiben 12](#_Toc98767251)

[a. Text aus einer Datei lesen 12](#_Toc98767252)

[b. Schreiben in eine Datei 13](#_Toc98767253)

[c. Die Methoden read() und readlines() 14](#_Toc98767254)

[13. Formatierte Ausgabe und Strings formatieren 14](#_Toc98767255)

[a. Print-Funktion 14](#_Toc98767256)

[b. Formatierte Stringliterale oder F-Strings 15](#_Toc98767257)

[c. Zahlen formatieren 16](#_Toc98767258)

[d. Weitere String-Methoden: ljust, rjust, center und zfill 16](#_Toc98767259)

[14. Referenzen, flaches und tiefes Kopieren 17](#_Toc98767260)

[a. Kopieren einfacher Variablen 17](#_Toc98767261)

[b. Kopieren einer Liste 17](#_Toc98767262)

[c. Flache Kopien 17](#_Toc98767263)

[d. Kopieren mit deepcopy 18](#_Toc98767264)

[e. Deepcopy für Dictionnaries 18](#_Toc98767265)

[15. Funktionen 18](#_Toc98767266)

[a. Funktionsdefinition 18](#_Toc98767267)

[b. Docstring 18](#_Toc98767268)

Quelle: Einführung in Python 3 von Bernd klein, Hanser-Verlag 2013

# Kommandos und Programme

## Interaktive Shell

Er steht zwischen den Anwender und dem Betriebssystem. Dabei schützt der Shell das Betriebssystem vor dem Anwender und anderer Seite erspart er dem Anwender von dem Basis-Funktionen des Betriebssystems.

Eigenschaft:

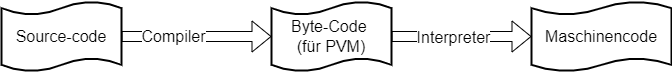
* Ein Befehl wird nach dem Drücken der Eingabetaste direkt ausgeführt.
* Der Unterstrich(\_) wird als Variable zum Aufrufen des vorherigen Ergebnisses verwendet.
* Vorheriger Befehl mit (↑) aufrufbar
* Verlassen des Shells: Ctrl-D oder mit der Funktion exit()
* Variable können deklariert werden und bleiben während der ganzen Session erreichbar.
* Schleife sind auch möglich

# Programme schreiben

* Ein Programm von der Datei hello.py 🡪 in dem Linux oder Windows Shell: python3 hello.py

# Bytecode und Maschinencode

Ablauf zur Ausführung eines Python-Script



PVM: Python-Virtual-Maschine

Fragen:

* Handelt es sich bei Python um eine Programmier- oder eine Skriptsprache?

Programm: Vor Ausführung wird es übersetzt bzw. kompiliert und dann ausgeführt. Es kann mehrere Zeilen(tausend, Million) haben.

Skript: wird unmittelbar ausgeführt bzw. interpriert.

Python ist keine Programmiersprache und auch keine Skript Sprache. Aus der oberen Abbildung kann man entnehmen, dass ein Source code zuerst in Byte-Code in einer PVM übersetzt wird und in einer bestimmten Maschinencode übersetzt.

* Wird ein Python-Programm übersetzt oder interpretiert?

Siehe obige Abbildung

* Kann ein Python-code auf verschiedenen Rechnern oder Betriebssystemen laufen?

Ja, sie ist Betriebssystem unabhängig

* Worin liegt der Zusammenhang zwischen Python, Jython oder Java?

Der Ausführungsablauf entspricht dem von Java.

# Datentypen und Variablen

In Python wird der Datentyp einer Variablen automatisch erkannt und kann im Programmablauf geändert werden: Dynamische Typdeklaration.

*Type(i)*

*Isinstance(object,klasse)*

Die Funktion „type“ gibt der aktuelle Datentyp von der Variabel i.

Die Funktion „isinstance“ überprüft die Variable/Objekt auf die vorgegebene Klasse/Typ und gibt entweder „True“ oder „False“ zurück. Man kann auch mehrere Datentypen eingeben:

*Isinstance(object,(typ1, typ2))*

Typverletzung: Wenn ungültige Operationen zwischen Variablen verschiedener Datentyp auftreten. ZB. Addieren eines Strings und Integer.

## Referenzieren zweier Variablen überprüfen: id() und „is“

In Python kann man überprüfen, ob zwei Variablen auf das gleiche Objekt zeigen. Das ist mit den Operatoren „ is“ und Vergleich auf ID’s:

x = 22.0

y = 22.0

print(id(x) == id(y))

🡪 True

Die Funktion id() gibt die ID einer Variablen. Haben zwei IDs die gleiche Variable, dann zeigen sie auf das gleiche Objekt. Das gleiche kann mit dem Operator „is“ überprüft werden. Haben die Variablen nur den gleichen Wert, gibt der Operator „is“ False zurück, denn er überprüft, ob die Objekte identisch sind.

x = 22.0

y = 22.0

print(x is y)

🡪 False

x=y

print(x is y)

🡪 True

## Datentyp: Binär

* Eine Binärzahl begint mit „0b“ oder 0B““.
* Eine Oktalzahl beginnt mit „0o“ ider „0O“.
* Eine Hexadezimalzahl beginnt mit „0x“ ider „0X“

Die Funktion hex(int) und oct(int) wandelt der Int-Parameter in hexadizamler bzw. Octalzahl um und gibt sie als String zurück.

## Komplexe Zahlen

x = 2+2j

y = 5+4j

print(x+y)

🡪 (7+6j)

# Sequenzielle Datentypen

Sequenzielle Datentypen sind Variablen, die mehrere Werte haben oder sequenziell angeordnet sind. Man unterscheidet:

* Strings: Anordnung beliebiger Zeichen. Ein Zeichen kann nicht durch Zuweisung geändert werden
* Lists: Anordnung beliebigen Objekten in eckigen Klammern: x = [4, 4.6, “text“]
* Tupel:

Anordnung beliebigen Objekten in runden Klammern: x =(4, 4.6, “text“) und kann nicht mehr im Programm verändert werden.

Tupel-Entpacken:

*T =(“Philip”, “Glass”)*

*(first,last))= T*

*print(first) 🡪 Philip*

*first, \*remainder = 35,99,27,18*

*print(first) 🡪 35*

*print(remainder) 🡪 [99,27,18]*

*first, \*middle, last = 35,99,27,18*

*print(first) 🡪 35*

*print(middle) 🡪 [99,27]*

*print(last) 🡪 18*

Eigenschaften von sequenziellen Datentypen:

* Indexierung ist gleich.
* Die Slice- oder Ausschneiden-Funktionen sind bei allen gültig.

## Strings

Ein String kann über mehrere Zeilen definiert werden: mit “““ oder /

x = "erste Zeile \

zweite Zeile \

Dritte Zeile"

print(x)

*🡪 erste Zeile zweite Zeile Dritte Zeile*

x = """erste Zeile

zweite Zeile

Dritte Zeile"""

print(x)

🡪 erste Zeile

zweite Zeile

Dritte Zeile

x = """erste Zeile

zweite Zeile \

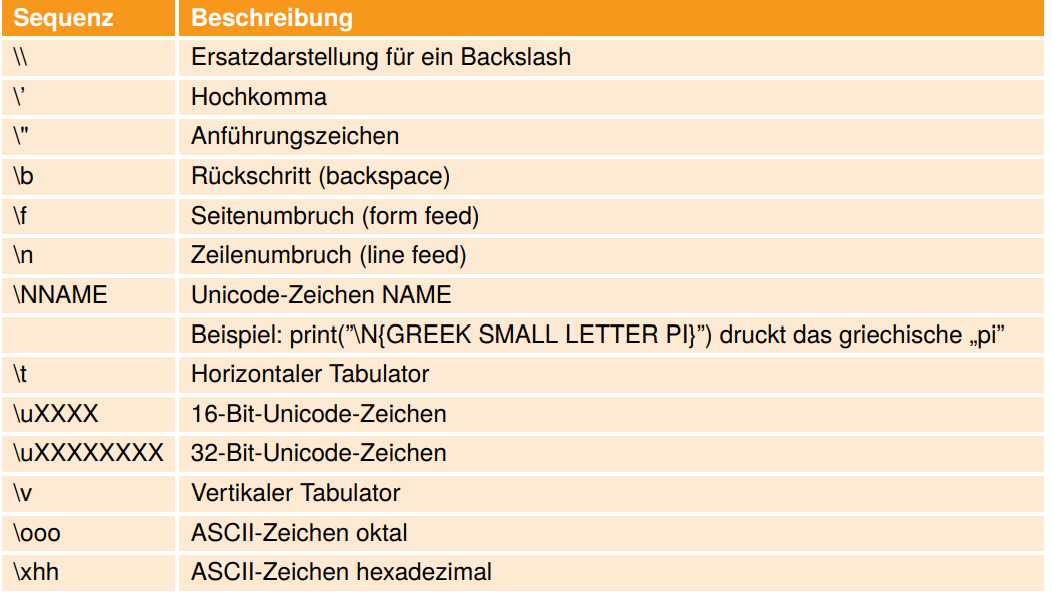
Dritte Zeile"""

print(x)

🡪 erste Zeile

zweite Zeile Dritte Zeile

### Escape-Zeichen und Raw-String



Sollen diese Zeichen in einem String nicht interpretiert, werden das String, in dem sie sich befinden, als Raw-String(mit einem r oder R vorne) deklariert werden.

print(r"Manger c\'est super \n magnifique !")

🡪 Manger c\'est super \n magnifique !

## Listen

Var =[2,5,"Manger", ["L",4]]

Listen sind veränderlich und können Objekte verschiedener Klassen/ Datentypen enthalten.

## Tupel

Var =(2,5,"Manger", ["L",4])

Var =2,5,"Manger", ["L",4]

Tupel können wie Listen verschiedene Datentypen enthalten. Sie sind aber nicht veränderbar. Folgende Operationen sind dabei nicht möglich:

* Löschen
* Hinzufügen
* Element ändern

### Tupel entpacken

minimum, maximum, text = 3, 99, "Ein Text"

print(str(minimum) +"  "+ str(maximum)+ "  "+ text )

# Dictionaries

## Definition

Es gibt in Python noch die Kategorie Mapping als Datentyp und diese Kategorie hat einen einzigen implementierten Typ: Dictionary. Dictionary beinhaltet mehrere Objekte, die durch Schlüsselwörter abgebildet sind. Sie sind erweiterbar und veränderbar.

*Empty={} 🡪 Leer Dictionary*

*Empty = {“ham” : ‘yes’ , “spam” : ‘no’}*

*print(Empty) 🡪 {‘ham’ : ‘yes’ , ‘spam’ : ‘no’}*

*print(Empty[“ham”]) 🡪 ‘yes’*

*Empty[“spam”]=”yes”*

*print(Empty) 🡪 {‘ham’ : ‘yes’ , ‘spam’ : ‘yes’}*

* Python kann die Reihenfolge der Dictionary beliebig ausgeben.
* Der Zugriff auf einem wert ist durch das Schlüsselwort.
* Bei Zugriff auf einem nicht existierenden Schlüsselwort wird den Error: KeyError erzeugt.

## Fehlerfreie Zugriffe auf Dictionaries

Vor dem Zugriff auf einem Element der Dictionary kann mithilfe des Schlüsselwortes „in“ überprüft werden, ob dieses Element vorhanden ist.

*Empty={} 🡪 Leer Dictionary*

*Empty = {“ham” : ‘yes’ , “spam” : ‘no’}*

*if “ham” in Empty:*

*print(Empty[“ham”]*

*else:*

*print(“Eintrag nicht bekannt”)*

## Zulässige Type für Schlüssel und Werte

Für die Werte gibt es keine Einschränkungen. Für den Schlüssel allerdings können folgende Datentypen nicht verwendet werden: Lists und Dictionary.

## Verschachtelte Dictionaries

*en\_fr={“red” : “rouge”, “green” : “vert”}*

*de\_fr={“rot” : “rouge”, “grün” : “vert”}*

*# Verschachtelte*

*dictionary={“en\_fr” : en\_fr, “de\_fr” : de\_fr}*

*# Zugriff*

*print(dictionary[en\_fr][“red”]) 🡪 rouge*

## Methode auf Dictionaries

D ist im Folgenden eine Variable des Datentyps Dictionary

* D.Clear(): Löschen alle Einträge.
* D.copy(): Erzeugt eine flache Kopie.
* D2= D.fromkeys(D, “yellow”): liefert ein Dictionary zurück, bei dem alle Einträge von dem Dictionary D den Wert „yellow“ haben. Sollte der zweite Parameter fehlen, werden die Werte des Dictionary auf „None“ gesetzt.
* D.get(“Schlüssel“): Gibt den Wert vom „Schlüssel“ oder None, wenn der Eintrag nicht vorhanden ist.
* D.items(): liefert ein Objekt vom Typ „dic\_item“ zurück , das vom ähnlichen Typ wie Dictionary ist: Ein Feld mit Tupeln aus Schlüssel und Werte vom D als Element.
* D.keys(): Liefert ein Objekt vom Datentyp „dic\_item“ zurück: Ein Feld mit Werten vom D als Einträge.
* D.pop(“Schlüssel“): Entfernt den Schlüssel mit seinem Wert und gibt ihn dabei der Wert zurück zur Bestätigung.
* D.popitem(): Entfernt einen beliebigen Schlüssel mit seinem Wert und gibt beide zurück
* D.setDefault(“Schlüssel”,”wert”): Fügt den Schlüssel mit dem Wert hinzu und gibt den Wert zurück. Wenn der Wert nicht eingegeben wird, wird der Schlüssel auf None gesetzt.
* D.update(D2): Fügt die Elemente von D2 in D hinzu. Dabei werden die Schlüssel von D, die bereits in D2 existiert, überschrieben.

## Operatoren auf Dictionaries

D ist eine Variable des Datentyps Dcitionary.

* len(D): gibt die Anzahl der Schlüssel-Werte-Paare zurück.
* Zip(lists1, lists2,lists3):

Lists1, lists2 und lists3 bilden hier eine Matrix. Durch die Funktion Zip wird diese Matrix transponiert. Sind die Listen verschieden lang, bestimmt der kleinste Vektor die Länge der transponierten Matrix.

Zum Entpacken der Transponierten Matrix: Zip(\*T)

Z1=[11,12,13]

Z2=[21,21,23]

Z3=[31,32,33]

T=zip(Z1,Z2,Z3)

print(list(T)) 🡪[(11, 21, 31), (12, 21, 32), (13, 23, 33)]

## Dictionaries aus Listen erzeugen

Die Liste muss zweier Tupel als Element haben.

Z1=["Pizza","Sauerkraut","Paela"]

Z2=["Italien","Deutschland","Spanien"]

T=zip(Z1,Z2)

print(list(T)) 🡪[('Pizza', 'Italien'), ('Sauerkraut', 'Deutschland'), ('Paela', 'Spanien')]

T=list(T)

Dictionary=dict(T)

print(Dictionary) 🡪 {'Pizza': 'Italien', 'Sauerkraut': 'Deutschland', 'Paela': 'Spanien'}

# Mengen

In Python entspricht der Datentyp „set“ („Collection“-Typ) eine Menge, bei der ein Element nur einmal vorkommt.

## Deklaration

Sollte bei der Deklaration mit „set“ Elemente doppelt vorkommen, werden sie bei Speichern vereinzeln.

laender= {"Italien","Deutschland","Spanien"}

laender= set(("Italien","Deutschland","Spanien","Italien"))

print(laender) 🡪{"Italien","Deutschland","Spanien"}

* Element einer Menge können nicht veränderlichen (mutable) Objekte sein. Wie Zum Beispiel Liste oder Dictionary

NB.: „frozensets“ sind wie „set“ aber nicht veränderbar.

## Operationen auf „set“-Objekten

Methode:

S eine Variable des Datentyps „set“

* S.add(element): Fügt ein Element in der Menge hinzu, falls es nicht existiert
* S.clear(): leert die Menge
* S.copy(): Erzeugt eine flache Kopie.
* S1.difference(S2): liefert die Elemente in Menge von S1, die sich nicht in S2 befinden.
* S1.difference(S2): Liefert eine Menge, wobei die Elemente von S2, die sich in S1 befinden, gelöscht werden.

x={"a","b","c", "d","e"}

y={"b","c"}

z={"c","d"}

x.difference\_update(z)

print(x) 🡪 {'a', 'b', 'e'}

* S.discard(element): Löscht das Element in S. Wenn das Element nicht vorhanden ist, passiert nichts.
* S.remove(element): Löscht das Element in S. Wenn das Element nicht vorhanden ist, wird ein Fehler generiert.
* S1.intersection(S2) oder S1 & S2: Gibt die Schnittmenge von S1 und S2: S1∩S2
* S1.isdisjoint(S2): Gibt True Zurück, wenn die Schnittmenge leer ist. Andersfalls False.
* S1.issubset(S2) oder S1<S2: Gibt True zurück, wenn S1 eine Untermenge von S2 ist. Andersfalls False.
* S1.issuperset(S2) oder S1>S2: Gibt True zurück, wenn S1 eine Obermenge von S2 ist. Andersfalls False.
* S.pop(): Löscht ein beliebiges Element in der Menge und gibt es zurück.

# Verzweigungen

# Schleifen

## While

Neu: Es ist möglich „else“ bei einer „While“-Schleife hinzufügen. Der Teil wird ausgeführt, wenn die Bedingung der „While“-Schleife nicht erfüllt wird und die Schleife nicht durch eine Break-Anweisung abgebrochen wurde.

x= "YES"

y="No"

while x!=y:

    y=input()

    print("raté")

else:

    print("trouvé")

## For- Schleife

Die For-Schleife in Python dient dazu eine Datensequenz zu iterieren. Eine Variable (index) wird bei jeder Iteration den Wert des entsprechenden Elements sein.

Es ist auch möglich einen else-Block hinzufügen und funktioniert wie bei der While-Schleife.

x= ["C", "C+", "Perl", "Python"]

for index in x:

    print(index)

else:

    print("Terminé")

🡪 C

C+

Perl

Python

Terminé

# Module importieren

Import math

math.sin(zahl)

Pi= math.pi

dir(math) 🡪 Zeigt alle Funktionen der Klasse math

help(math.tan) 🡪 Gibt Information zu der Fonction tan in der klasse math

# Eingaben

Eingaben durch die Tastatur wird von der Funktion input() ermöglicht.

1. s = input("Bitte Ihre Eingabe: ")
2. s = int(input("Bitte Ihre Eingabe: "))
3. s = eval(input("Bitte Ihre Eingabe: ")) Bitte Ihre Eingabe: [2,6,9]
4. print(s) 🡪 [2, 6, 9]
5. print(type(s)) 🡪 <class 'list'>

Hier gibt es mehrere Möglichkeiten, die Eingabe einzufordern.

1 🡪 Ein String wird zurückgegeben.

2 🡪 Die Eingabe wird direkt als integer umgewandelt. Wird kein Integer eingegeben, erzeugt der Interpreter einen Fehler.

3 🡪 Die Funktion eval() überprüft die Eingabe und ordnet ihr automatisch einen Datentyp zu. Somit kann man eine Liste, Dictionary, Tupel als String eingeben und die Eingabe wird automatisch in den entsprechenden Datentype umgewandelt.

# Dateien lesen und schreiben

Datei: ist eine Menge von logisch zusammenhängenden und meist sequentiell geordneten Daten, die auf einem Speichermedium dauerhaft gespeichert werden und mittels eines Bezeichners bwz. Namen wieder identifizierbar und damit ansprechbar sind.

Der Inhalt jeder Datei besteht grundsätzlich aus einer eindimensionalen Aneinanderreihung von Bits, die normalerweise in Byte-Blöcke zusammengefasst interpretiert werden. Dabei wird durch das Anwendungsprogramm und das Betriebssystem entschieden, ob es sich um eine Textdatei, ausführbares Programm, Musikstück oder Bild handelt.

## Text aus einer Datei lesen

Zum Lesen einer Textdatei wird die Funktion open(filename, mode) verwendet:

* Filename: Datentyp Sting und kann auch ein Pfad sein.
* Mode: „r“, „w“. Ohne diese Angabe wird der Mode automatisch „r“.

Beispiel:

fobj=open("C:\\Users\\E1376231\\Desktop\\Document.txt", "r")

iterator=0

for line in fobj:

    if iterator==0:

        print(line.rstrip())

        break

    iterator +=1

fobj.close()

Dieser Code liest die erste Zeile der Textdatei „Document.txt“.

* Die Funktion open() erzeugt eine Dateiobjekt und liefert eine Referenz zu diesem Objekt als Ergebniswert zurück.
* Die For-Schleife bearbeitet die Textdatei hier Zeile pro Zeile. Das Ende einer Zeile wird durch Zeilenumbruch identifiziert.
* rstrip() ist eine Funktion der Klasse String und entfernt Leerzeichen und newlines vom rechten Rand
* Durch die Funktion close() wird die Datei geschlossen.

## Schreiben in eine Datei

Beispiel:

fobj\_in=open("C:\\Users\\E1376231\\Desktop\\Document.txt", "r")

fobj\_out=open("C:\\Users\\E1376231\\Desktop\\Document2.txt", "w")

iterator=0

for line in fobj\_in:

    lineout= str(iterator)+ " "+ line.rstrip() +"\n"

    fobj\_out.write(lineout)

    iterator +=1

    print(lineout)

fobj\_in.close()

fobj\_out.close()

* Mit der Mode „w“ (write) wird die Datei „Document2.txt“ erstellt, wenn er im Ordner nicht vorhanden ist. Wenn die Datei schon existiert, wird der Inhalt gelöscht.
* "\n" ermöglicht den Zeilenumbruch.
* Mit der Funktion write() wird in der Datei geschrieben.
* Am Ende müssen die zwei Dateien geschlossen werden.

## Die Methoden read() und readlines()

* Mit readlines () wird die Datei gelesen und der Inhalt als Feld mit Zeilen der Textdatei als Elemente zurückgeliefert. Somit können die Zeilen der Textdatei durch Indexierung angesprochen werden.

Readlines() öffnet und schließt direkt die Datei.

Beispiel:

text=open("C:\\Users\\E1376231\\Desktop\\Document.txt").readlines()

print(text)

print(text[0])

Ausgabe

["je m'appele Franc. \n", "C'est bien moi\n", 'et on va manger. ']

je m'appele Franc.

* Mit read() wird die Datei gelesen und der Inhalt als String zurückgeliefert. Somit können die Zeichen der Textdatei durch Indexierung angesprochen werden.

text=open("C:\\Users\\E1376231\\Desktop\\Document.txt").read()

print(text)

print(text[0])

Ausgabe

je m'appele Franc.

C'est bien moi

et on va manger.

j

Leerzeichen am Ende der Zeilen werden aufgenommen.

# Formatierte Ausgabe und Strings formatieren

In Python können Ausgaben durch formatierte Stringliterale, die Stringmethode format() oder Stringmodulo formatiert werden.

## Print-Funktion

Das Aufrufverhalten von der Print-Funktion wird in Python folgendermaßen definiert:

print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Sie nimmt eine beliebe Anzahl an Wert und gibt sie per default mit einem leer-Zeichen aus.

a = 3.564

print("a =", a) 🡪 a = 3.564

Das Trennzeichen kann durch den Parameter „sep“ geändert werden :

a = 3.564

b= "e"

print(a, b, sep="") 🡪 3.564e

Per Default endet Print-Funktion die Ausgabe mit „\n“. Dies kann durch den Parameter „end“ geändert werden:

a = 3.564

b= "e"

print(a, b, end=" :-)") 🡪 3.564 e :-)

Der String kann in einer Datei oder in einem Standardfehlerkanal geschrieben werden, anstatt es in dem Terminal:

fojb= open("C:\\Users\\E1376231\\Desktop\\Test.txt","w")

a= "Das ist ein Textfür die Datei Test."

print(a, end=" :-)", file=fojb)

fojb.close()

import sys

# Ausgabe in Standardfehlerkanal:

print("Error: 42", file=sys.stderr)

🡪 Error: 42

Dieser Kanal ist in Linux vorhanden.

## Formatierte Stringliterale oder F-Strings

F-Strings bietet eine Möglichkeit, Ausdrücke mit einer minimalen Syntax in String-Literale einzubetten.

name= "Franc"

alter= 25

print(f" Ich heiße {name} und bin {alter} alt.")

🡪 Ich heiße Franc und bin 25 alt.

Sie kann auch if-Anweisungen beinhalten:

def volljährig(alter: int) -> bool:

    if alter >=18:

        return True

    else:

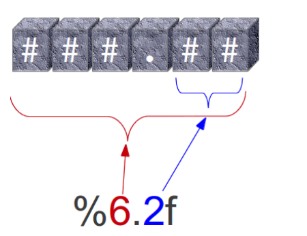
        return False

personen = [("Katja", 17), ("Anja", 19), ("Jonas", 19)]

for name, alter in personen:

    print(f"{name} ist {'volljährig' if volljährig(alter) else 'minderjährig'}")

## Zahlen formatieren

Das Bild bedeutet, dass die Zahl insgesamt 9 Zeichen haben muss und 2 Stellen nach dem Komma sind erlaubt.

stadt = "New York"

temperatur\_fahrenheit = 37.2

print(f"{stadt}: {(temperatur\_fahrenheit - 32) \* 5 / 9:9.2f} Grad Fahrenheit!")

🡪 New York: 2.89 Grad Fahrenheit!

Mit „:09.2f“ 🡪 Werden die Führenden Stellen mit 0 gefühlt.

stadt = "New York"

temperatur\_fahrenheit = 37.2

umwandeln= (temperatur\_fahrenheit - 32) \* 5 / 9

print(f"{stadt}: {umwandeln:09.2f} Grad Fahrenheit!")

🡪 New York:000002.89 Grad Fahrenheit!

|  |  |
| --- | --- |
| Daten-Typ | Formatierung |
| Integer | 6d |
| String | 20s |

## Weitere String-Methoden: ljust, rjust, center und zfill

s = "Python"

print(s.center(10)) 🡪 Python

print(s.center(10, "\*"))🡪 \*\*Python\*\*

print(s.ljust(10, "\*")) 🡪 Python\*\*\*\*

print(s.rjust(10, "\*")) 🡪 \*\*\*\*Python

print(s.zfill(10)) 🡪 0000Python

Ist die angegebene Zahl kleiner als die String-Länge, wird das String normal ausgegeben.

# Referenzen, flaches und tiefes Kopieren

## Kopieren einfacher Variablen

1. x= "Transfert"
2. y=x
3. print(id(x)==id(y)) 🡪 True
4. y= "Straße"
5. print(id(x)==id(y)) 🡪 False

Die Zuweisung in der Zeile 2 ist eine einfache Kopie. Dabei zeigen die zwei Variablen x und y auf das gleiche Objekt ( Sie referenzieren auf das gleiche Objekt). Aus diesem Grund haben die beiden die gleiche ID- Nummer ( id() ). Sobald eine von den Variablen ein neues Objekt zugewiesen bekommt, ist es nicht mehr der Fall.

## Kopieren einer Liste

1. x= ["un", "deux"]
2. y=x
3. print(id(x)==id(y)) 🡪 >>> True
4. y[0]=["Trois"]
5. print(id(x)==id(y)) 🡪 >>> True
6. print(f"x: {x} und y: {y}") >>> x: [['Trois'], 'deux'] und y: [['Trois'], 'deux']
7. y=["cing", "six"]
8. print(id(x)==id(y)) 🡪 >>> False

Die Operation in der Zeile 2 ist keine Kopie. Diese Operation bewirkt, dass y das gleiche Objekt referenziert. Sie haben also die gleiche Adresse. Deswegen True in der Zeile 3. Sollte der Inhalt von y geändert werden, bleibt die Adresse von y gleich und die Änderung gilt auch für die Variable x. Deswegen True in Zeile 5 und die Ausgabe in der Zeile 6. Sollte aber y eine andere Liste zugewiesen bekommen, gilt das nicht mehr, denn es wird für y einen neuen Speicherplatz reserviert.

## Flache Kopien

1. x= ["un", "deux"]
2. y=x.copy()
3. print(id(x)==id(y)) >>> Flase
4. x= ["Un",["a","b"]]
5. y=x.copy()
6. print(id(x)==id(y)) >>> Flase
7. y[0]="Quatre"
8. y[1][0]="Z"
9. print(f"x: {x} und y: {y}") >>> x: ['Un', ['Z', 'b']] und y: ['Quatre', ['Z', 'b']]
10. print(f"x[1]: {id(x[1])} und y: {id(y[1])}") >>> x[1]: 1559047777600 und y: 1559047777600

Die Verwendung der Funktion copy() in der Zeile 2 und 5 wird jedes Mal eine flache Kopie der Variable x erzeugt. Sie referenzieren aber nicht auf die gleichen Objekte. Deswegen False in den Zeilen 3 und 6. Ist aber ein Element von x eine Liste, wird beim flachen Kopieren nur der Link zu der Liste kopiert (Zeile 10). Somit zeigen beiden Elementen auf das gleiche Objekt.

## Kopieren mit deepcopy

1. from copy import deepcopy
2. x= ["Un",["a","b"]]
3. y=deepcopy(x)
4. print(id(x)==id(y))
5. print(f"x[1]: {id(x[1])} und y: {id(y[1])}") >>> x[1]: 3021134492544 und y: 3021134441920

Die Funktion deepcopy() löst das oben beschriebene Problem.

## Deepcopy für Dictionnaries

Das gleiche gilt für Dictionnaries.

# Funktionen

Eine Funktion ist ein Strukturierungselement in einer Programmierungssprache. Die dient dazu, Wiederholungen in Code zu vermeide. Es folge eine bessere Lesebarkeit der Code und erleichtert die Wartung der Software.

## Funktionsdefinition

1. def funktionsname(parameter1, paramter2):
2. x=parameter1\*paramter2
3. return x
4. # Funktionsaufruf
5. mul= funktionsname(2,6) \*2
6. print(mul) 🡪 >>> 24

Die Funktionsdefinition wird mit dem Schlüsselwort „def“ eingeleitet. Eine Funktion kann einen, mehrere oder keine Parameter haben. Die Erste Zeile wird Funktionskopf genannt. Der Code mit der Einrückung(2,3) bildet den Funktionskörper oder Funktionsrumpf. „return“ liefert das Ergebnis „x“ an die Variable in der aufrufenden Stelle. Eine Funktion kann auch ohne „return“ geschrieben werden.

Gut Praxis: Den Code schreiben, sodass die „return“-Anweisung am Ende der Funktion steht.

Die Zeile 5 ruft die Funktion auf und liefert das Ergebnis multipliziert mit 2 an die Variable x zurück. Die Funktion kann somit so oft wie nötig aufgerufen werden.

## Docstring

Docstring ist ein String in einenm Python-Programm zu Dokumentationszwecken.

Dieser String steht direkt nach dem Funktionskopf und beschreibt, was die Funktion leistet.

1. def funktionsname(parameter1, paramter2):
2. """
3. Diese Funktion rechnet die Multiplikation aus parameter1 und parameter2
4. """
5. x=parameter1\*paramter2
6. return x
7. # Funktionsaufruf
8. mul= funktionsname(2,6)
9. print(mul)

Die Zeile 2 bildet den Docstring, eingeleitet durch „"""“.

Der Docstring wird in der Variable „\_\_doc\_\_“ gespeichert und kann auch mit der Funktion help aufgerufen werden.

print(funktionsname.\_\_doc\_\_)

- >>> Diese Funktion rechnet die Multiplikation aus parameter1 und parameter2

help(funktionsname)

>>>

Help on function funktionsname in module \_\_main\_\_:

funktionsname(parameter1, paramter2)

Diese Funktion rechnet die Multiplikation aus parameter1 und parameter2

PS: Jedes Python-Programm kann als Modul verwendet werden.

Ein Modul „math\_franc“ kann folgendermaßen definiert werden:

"""

Das Modul beinhaltet mathematische Operationen mit zwei Operanden

"""

def multiplikation(parameter1, paramter2):

    """

    Diese Funktion rechnet die Multiplikation aus parameter1 und parameter2

    """

    x=parameter1\*paramter2

    return x

def division(parameter1, paramter2):

    """

    Diese Funktion rechnet die Division aus parameter1 und parameter2

    """

    x=parameter1/paramter2

    return x

Der erste Docstring beschreibt, was das Modul generell macht und jede Funktion in dem Modul hat seinen Docstring.

Dieses Modul soll in einem anderen Programm verwenden. Dafür muss die Datei „math\_franc.py“ in denselben Ordner mit dem nächsten Programm gespeichert werden.

import math\_franc

help(math\_franc)

>>>

Help on module math\_franc:

NAME

math\_franc - Das Modul beinhaltet mathematische Operationen mit zwei Operanden

FUNCTIONS

division(parameter1, paramter2)

Diese Funktion rechnet die Division aus parameter1 und parameter2

multiplikation(parameter1, paramter2)

Diese Funktion rechnet die Multiplikation aus parameter1 und parameter2

FILE

c:\users\e1376231\desktop\test python\math\_franc.py

Eine Funktion des Moduls „math\_franc“ aufrufen:

import math\_franc

x= math\_franc.division(4,2)

print(x)

>>> 2.0

[("GRANT USAGE ON \*.\* TO `admin`@`%` IDENTIFIED BY PASSWORD '\*D5F0FC0BD02EBD92D2E85A254E66AE2DAD4CA636'",)]