5. Ein- und Ausgabe in Python

Ein Computer arbeitet nach dem *EVA-Prinzip* (**E**ingabe-**V**erarbeitung-**A**usgabe), d.h. er empfängt Daten über eine Eingabeeinheit, verarbeitet sie und liefert das Ergebnis über ein Ausgabegerät an seine Umgebung. E/A-Operationen sind z.B.:

Eingabe:

- Eingabe von Zeichen über die Tastatur
- Lesen von Dateien, die auf Peripheriespeichern gespeichert sind (Festplatte, CD, Memory-Stick, etc.)

Ausgabe:

- Wiedergabe von Texten und Zahlen auf dem Bildschirm in einem Konsolenfenster
- Schreiben in Dateien, die auf Peripheriespeichern gespeichert sind

All diese Operationen verlaufen bei Python über Objekte des Typs `file`.

5.1 Files

5.1.1 Was ist ein `file`?

Ein `file` (deutsch: Datei) ist eine Aneinanderreihung von *Bits*. Eine 8-Bit-Einheit (Oktette) bezeichnen wir als *Byte*. Damit wissen wir also, dass alle Daten im Speicher *binär* abgelegt sind.

Diese Pattern aus Nullen und Einsen können je nach Kontext auf unterschiedliche Weise interpretiert werden. Um eine Binärzahl als Ganzzahl zu interpretieren, kann man diese Formel verwenden:

$$z = \sum_{i=0}^{n-1} 2^i \; b_i$$

In [1]:
```
# TODO Beispiel Binärzahl
zahl = 0b00101010
zahl
```

Out[1]: 42

Das gleiche Pattern kann aber auch für ein Schriftzeichen stehen. In diesem Fall wird mittels einer Tabelle übersetzt.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Quelle: https://upload.wikimedia.org/wikipedia/commons/1/1b/ASCII-Table-wide.svg

In [3]:
```
# TODO Beispiel Schriftzeichen
schrift = chr(0b00101010)
schrift
ord('*')
```

Out[3]: 42

Das `file`-Objekt kann (im Prinzip) beliebig lang sein. Das Ende wird durch das Sonderzeichen `eof` (*end of file*, ASCII: `0x04`) gekennzeichnet.

5.1.2 Ein `file`-Objekt erzeugen, lesen und schließen

Ein `file`-Objekt wird durch einen Aufruf der Standardfunktion `open()` erzeugt. Der allgemeine Syntax lautet `open(filename, mode = "r")`, wobei:

- `filename`: Pfad bestehend aus zwei `string`-Teilen: Bezeichnung des Verzeichnisses (z.B. */python/programme/*) und Dateiname (z.B. *textdatei.txt*)
- `mode`: beschreibt in welchem Modus die Datei geöffnet werden soll. Default = `"r"`

`mode`	**Erklärung**
`"r"`	Datei wird ausschließlich zum Lesen geöffnet. Sie muss bereits existieren
`"w"`	Datei wird ausschließlich zum Schreiben geöffnet, existiert bereits eine Datei gleichen Namens, wird ihre Länge auf null gesetzt und sie wird **überschrieben**
`"a"`	"append": Datei wird ausschließlich zum Schreiben geöffnet, existiert bereits eine Datei gleichen Namens, so wird diese **erweitert**
`"r+"`	Die Datei wird zum Lesen und Schreiben geöffnet. Sie muss bereits existieren

`mode`	**Erklärung**
`"w+"`	Die Datei wird zum Lesen und Schreiben geöffnet, existiert bereits eine Datei gleichen Namens, so wird diese **überschrieben**
`"a+"`	Die Datei wird zum Lesen und Schreiben geöffnet, existiert bereits eine Datei gleichen Namens, so wird diese **erweitert**

Die Datei wird standardmäßig im *Textformat* geöffnet. Durch das Hinzufügen eines `"b"` (z.B. `"rb"`, `"wb"`, `"r+b"`, `"w+b"`) wird diese im *Binärformat* geöffnet, d.h. deren Inhalt wird als `byte`-Objekte zurückgegeben.

Anmerkung zum `filename`: Absolute Pfadangaben sind von Natur aus unflexibel, daher sind stets relative Pfadangaben zu verwenden. Konzipieren Sie nach Möglichkeit Ihre Programme so, dass notwendige Dateien in Unterordnern liegen, die vom Hauptprogramm relativ gesehen einfach zu erreichen sind!

Zum Lesen wird die Methode `read(size)` verwendet. `size: int` ist hierbei ein optionales Argument. Wird die Angabe von `size` weggelassen oder ist diese negativ, wird das gesamte `file` gelesen und zurückgegeben, ansonsten wird maximal die übergebene Anzahl an Bytes gelesen.

Die offizielle Python-Dokumentation kommentiert hierzu lakonisch:

> *It's your problem if the file is twice as large as your machine's memory.*

In [8]:
```
# Die Datei "LICENSE_PYTHON.txt" muss im Ordner "daten",
# der auf der gleichen Ebene wie das Workbook ist, liegen
daten = open("daten/LICENSE_PYTHON.txt")
#print(daten.read(10))
#print(daten.read())
#print("Nach dem Ende")
#print(daten.read())
```

Häufig sollen Textdateien Zeile für Zeile bearbeitet werden. Dafür existiert die Methode `readline(size)`. `size` schreibt die maximal zu lesende Datenmenge vor. Wird dieses Argument ausgelassen, so liest Python *die gesamte Zeile*.

Ähnlich ist die Methode `readlines()`. Diese liest die *gesamte Datei*, zerlegt sie dabei aber bereits an den Zeilenumbrüchen und packt die Teile in eine `list`.

In Python haben Zeilenumbrüche das Format `\n`.

In [ ]:
```
print(daten.readline())
# TODO readlines()
lines = daten.readlines()
for zeile in lines:
    print(zeile)
```

Anmerkungen:

→ die verschiedenen Betriebssysteme haben verschiedene Zeilenenden (`\n` unter Linux, `\r\n` unter Windows, `\r` bei Macs)

→ Python funktioniert **plattform-unabhängig**, da es beim Lesen die plattformspezifischen Zeilenenden in `\n` und beim Schreiben wieder zurück konvertiert.

→ `file`-Objekte sind *Iteratoren*. Deshalb kann man diese auch nur einmal lesen/durchlaufen.

In [ ]:
```
# weil file-Objekte Iteratoren sind, können diese mit einer
# for-Schleife durchlaufen werden
daten = open("daten/LICENSE_PYTHON.txt")
# TODO
for d in daten:
    print(d, end="")
```

Jede Datei, die geöffnet wurde, sollte auch wieder geschlossen und gespeichert werden. Dies kann man mit dem Attribut `closed` überprüfen und die Datei mit der Methode `close()` schließen.

In [15]:
```
# TODO Datei schließen
daten.closed
daten.close()
daten.closed
#daten.read()
```

Out[15]:
```
True
```

5.1.3 Schreiben eines `file` - Objekts

Während eine Datei im Schreibemodus(!) geöffnet ist (`closed == False`), kann man mit der Methode `write(text)` Text in die Datei schreiben. `text` muss hierbei vom Datentyp `string` sein. Der Rückgabewert ist die Anzahl der geschriebenen Zeichen.

In [16]:
```
# Erzeugen einer neuen Datei eigeneDatei.txt und Schreiben mit write()
eigeneDatei = open("daten/eigeneDatei.txt", "w")

eigeneDatei.write("Sehr geehrte Damen und Herren\n")
eigeneDatei.write("bla"*5 + "\n")
eigeneDatei.write("Das war jetzt mit write()\n")

eigeneDatei.close()     # Speichern und Schließen
```

Alternativ kann man eine Datei auch mit der `print()`-Funktion beschreiben. Wir erinnern uns an den Syntax dieser Funktion und den optionalen Parameter `file`. Damit können wir obige Codezelle auch schreiben als:

In [18]:
```
# Erzeugen einer neuen Datei eigeneDatei.txt und Schreiben mit print()
eigeneDatei = open("daten/eigeneDatei.txt", "w")

print("Sehr geehrte Damen und Herren", file=eigeneDatei)
print("bla"*5, file=eigeneDatei)
print("Das war jetzt mit print()", file=eigeneDatei)

eigeneDatei.close()     # Speichern und Schließen
```

In [19]:
```
# Lesen der selbsterstellen Datei "eigeneDatei.txt"
eigeneDatei = open("daten/eigeneDatei.txt", "r")
text = eigeneDatei.read()
eigeneDatei.close()
print(text)
```

```
Sehr geehrte Damen und Herren
blablablablabla
Das war jetzt mit print()
```

5.1.4 Zwischenspeichern, ohne zu schließen

Man kann in Python Dateien einfach zwischenspeichern, ohne diese gleich zu schließen. Dies ermöglicht die Methode `flush()`.

In [24]:
```
# Beispiel zu flush()
flush = open("daten/flush.txt", "w")
flush.write("Hello darkness\n")
flush.flush()
```

In [25]:
```
# weitere Bearbeitung der Datei
print(flush.closed)
flush.write("my old friend\n")
flush.close()
flush.closed
```

```
False
```

Out[25]:
```
True
```

In [26]:
```
# Lesen der Datei
song = open("daten/flush.txt")
print(song.read())
song.close()
```

```
Hello darkness
my old friend
```

5.1.5 Dateicursor bewegen und bestimmen

Manchmal ist erforderlich, die aktuelle Cursorposition in der Datei zu kennen und zu verändern. Dafür dienen die Methoden `tell()` und `seek(offset, from)`.

`tell()` gibt die aktuelle Position des Cursors ab Dateianfang zurück.

`seek(offset, from)` verschiebt den Cursor, wobei:

- `offset`: Erforderlich. Gibt an, um wie viele Bytes der Cursor gegen einen bestimmten Referenzpunkt verschoben werden soll.
- `from`: Optional. Kann nur 0, 1 oder 2 sein:
  - `from=0`: Default. *offset* bezieht sich auf den Dateianfang.
  - `from=1`: *offset* bezieht sich auf die *aktuelle Position*.
  - `from=2`: *offset* bezieht sich auf das *Dateiende*

In [27]:
```
# TODO Funktion für Länge einer Datei mit Cursorn
def laengeFile(datei):
    oldPos = datei.tell()

    datei.seek(0,2) # springe ans Dateiende
    laenge = datei.tell() # Cursorposition am Ende = Länge

    datei.seek(oldPos)
    return laenge

with open("daten/LICENSE_PYTHON.txt", "r") as datei:
    print(f"{datei.name} ist {laengeFile(datei)} Bytes lang")
```

```
daten/LICENSE_PYTHON.txt ist 13936 Bytes lang
```

Damit ist es im Vergleich zu *klassischen Iteratoren* möglich, das `file`-Objekt *zurückzuspulen*, und durch dieses beliebig oft zu iterieren.

5.1.6 Die `with` - Anweisung

Wir haben gesehen, dass es für den fehlerfreien Ablauf eines Programms mit `file`-Objekten unerlässlich ist, Dateien kontrolliert zu öffnen und nach Verarbeitung wieder zu schließen, auch wenn zwischendurch etwas schief gehen sollte. Das ist quasi die Idee der `with`-Anweisung, deren Syntax wie folgt aufgebaut ist:

```
with objekt as name:
    anweisungsblock
```

oder im Zusammenhang mit Dateien:

```
with open(filename, mode) as dateiname:
    datei_anweisungsblock
```

Objekte können zwei besondere Methoden besitzen:

- `__enter__()`: öffnet eine Datei
- `__exit__()`: schließt die Datei

das `with`-Statement garantiert, dass *auf jeden Fall* die `__exit__()`-Methode aufgerufen wird, wenn zuvor die `__enter__()`-Methode erfolgreich ausgeführt werden konnte.

In [ ]:
```
# TODO Lesen mit with-Anweisung
with open("daten/LICENSE_PYTHON.txt") as daten:
    text = daten.readlines()

for t in text:
    print(t, end="")
```

In [ ]:
```
# TODO Lesen mit with-Anweisung alternativ
with open("daten/LICENSE_PYTHON.txt") as daten:
    for text in daten:
        print(text, end="")

daten.closed
```

5.1.7 Die Pseudofiles `sys.stdin` und `sys.stdout`

Eingaben über die Tastatur haben wir bisher bequem mithilfe der `input()`-Funktion abgefragt. Im Hintergrund laufen diese Eingaben über ein *Pseudofile*. Diese sind quasi ein `file`-Objekt mit eingeschränkten Zugriffsmöglichkeiten. Es besitzt keine Schreibmethoden, sondern lediglich die Methode `readline()`. Der Name dieses Pseudofiles lautet `sys.stdin`. Im Deutschen spricht mann auch von der *Standard-Eingabe*.

```
# input()-Funktion händisch programmiert
# funktioniert nur in der interaktiven Python-Shell
import sys
print("Eingabe: ", end = " ")
eingabe = sys.stdin.readline()
print("Ihre Eingabe war: ", eingabe)
```

Für die Ausgaben gibt es das Pseudofile `sys.stdout`, welches auch der Default-Parameter der Standard `print()`-Funktion ist. Dieses verhält sich wie ein `file`-Objekt, das nicht gelesen, sonder nur - mittels `write()` - beschreiben werden kann.

In [31]:
```
# Beispiel zur Ausgabe eines Strings ohne print()
import sys
sys.stdout.write("Hallo Welt!")
```

```
Hallo Welt!
```

Out[31]:
```
11
```

5.2 Objekte speichern mit `pickle`

pickle bedeutet im Deutschen *Essiggurke* und *to pickle* bedeutet *einlegen*. Das Python-spezifische Modul `pickle` stellt Funktionen bereit, mit denen man Programmdaten über das Programmende oder den Programmabbruch hinaus speichern kann. In der Informatik bezeichnet man diese Art von Daten als *persistente Daten*.

Folgende Objekt-Typen können mit dem `pickle`-Mechanismus gespeichert werden:

- Zahlen
- Strings
- Funktionen
- beliebige Sequenzen
- Dictionaries
- Instanzen selbst definierter Klassen (*spätere Vorlesung*)

5.2.1 Funktionen zum Speichern und Laden

Zum **Speichern** gibt es aus dem Modul die Funktion `dump(object, file)`, wobei:

- `object`: beliebiges Objekt aus der obigen Aufzählung

- `file` : Name eines `file` -Objekts, das im binären Schreibmodus geöffnet ist (`"wb"`)

In [32]:
```
# TODO Beispiel zum Speichern
import pickle
telefonbuch = [("Tim", "85675"), ("Jenny", "233325"), ("Max", "89923")]
with open("daten/tel.pkl", "wb") as datei:
    pickle.dump(telefonbuch, datei)
```

Zum **Laden** der Datei gibt es die Funktion `load(file)` , wobei:

- `file` : Name eines `file` -Objekts, das im binären Lesemodus geöffnet ist (`"rb"`)

In [33]:
```
# TODO Beispiel zum Laden
with open("daten/tel.pkl", "rb") as pik:
    geladenesTelefonbuch = pickle.load(pik)
print(geladenesTelefonbuch)
```

```
[('Tim', '85675'), ('Jenny', '233325'), ('Max', '89923')]
```

5.2.2 Wie funktioniert `pickle` ?

Beim Speichern wird aud dem übergebenen Objekt ein `bytestring` erzeugt, worin die Struktur des Objekts codiert ist. Insbesondere ist zu jedem elementaren Wert auch der Datentyp gespeichert.

Die Erzeugung einer Zeichenkette zur Repräsentation einer Datenstruktur nenn man auch *Serialisierung*.

In [34]:
```
# Rückgabewert der dumps()-Funktion
import pickle
s = pickle.dumps(telefonbuch)
s
```

Out[34]:
```
b'\x80\x04\x958\x00\x00\x00\x00\x00\x00\x00]\x94(\x8c\x03Tim\x94\x8c\x0585675\x
94\x86\x94\x8c\x05Jenny\x94\x8c\x06233325\x94\x86\x94\x8c\x03Max\x94\x8c\x05899
23\x94\x86\x94e.'
```

Beim Laden wird aus dem `bytestring` das Objekt wiedergewonnen, unter der Voraussetzung, dass der verwendete `bytestring` eine korrekte Repräsentation im Sinne des `pickle` -Protokolls darstellt

Warnung:

> The `pickle` module **is not secure**. Only unpickle data you trust.

5.3 CSV-Dateien

CSV steht für *comma separated values* und ist ein Datenformat, das uns besonders aus Excel bekannt ist. Es eignet sich daher für die Darstellung von Tabellen, deren Spalten durch Trennzeichen (i.d.R Kommata) voneinander getrennt sind.

Das Pythonmodul `csv` mit der Klasse `csv.reader` erlaubt es, Dateien im CSV-Format zu zerlegen

In [35]:
```
# TODO Datei mit CSV-Reader lesen
import csv
with open("daten/username.csv") as file:
    rdr = csv.reader(file, delimiter = ";")
    for line in rdr:
        print(line)
```

```
['Username', ' Identifier', 'First name', 'Last name']
['booker12', '9012', 'Rachel', 'Booker']
['grey07', '2070', 'Laura', 'Grey']
['johnson81', '4081', 'Craig', 'Johnson']
['jenkins46', '9346', 'Mary', 'Jenkins']
['smith79', '5079', 'Jamie', 'Smith']
```

In [1]:
```
# TODO Spaltenüberschriften herauslesen
import csv
with open("daten/username.csv") as file:
    rdr = csv.reader(file, delimiter = ";")
    header =  next(rdr) # "Überspringen der Spaltenüberschriften"
    for line in rdr:
        print(line)

print("Überschriften der Spalten sind ", header)
rdr
```

```
['booker12', '9012', 'Rachel', 'Booker']
['grey07', '2070', 'Laura', 'Grey']
['johnson81', '4081', 'Craig', 'Johnson']
['jenkins46', '9346', 'Mary', 'Jenkins']
['smith79', '5079', 'Jamie', 'Smith']
Überschriften der Spalten sind  ['Username', ' Identifier', 'First name', 'Last n
ame']
```

Out[1]:
```
<_csv.reader at 0x22ba1ed7340>
```