Netzwerkprogrammierung

Sockets mit Python: Grundlagen und Beispiele

S. Kaminski

Inhalt

Listings

1 Python und Sockets

Die Kommunikation im Internet wird heute als selbstverständlich angenommen. E-Mail, das World Wide Web und zahllose andere Dienste stehen rund um die Uhr zur Verfügung. Was geschieht dabei zwischen zwei Rechnern die miteinander Daten austauschen, was ist die Basis? Wie kann ich selber Daten übertragen?

Die heutzutage am häufigsten genutzte Technik basiert auf TCP/IP. Die beiden Abkürzungen TCP und IP stehen für „Transmission Control Protocol“ und „Internet Protocol". Über diese beiden Protokolle wird der Großteil der momentan verfügbaren Internetdienste realisiert und um deren Einsatz soll es im folgenden Text primär gehen. Im Text werden ein paar Beispielprogramme entwickelt, die untereinander Daten austauschen und die Kommunikation über diese Protokolle dargestellt.

Dieser Text führt Schritt für Schritt an die Entwicklung der zwei Kommunikationspartner in der Programmiersprache Python heran. Grundkenntnisse in Python werden nicht vorausgesetzt, die Bausteine werden im Text erläutert.

1.1 Konventionen in diesem Text

Zum Ausprobieren der Beispiele und für eigene Experimente werden nur ein Rechner und Python in der Version 3 benötigt. Python ist für alle gängigen Betriebssystem verfügbar. Aktuelle Versionen und Anleitungen für die Installation sind auf `www.python.org` zu finden.

Die Beispielausgaben der Programme in diesem Text stammen aus einer Linux-Shell[1], sollten aber auf jedem andereren Python-System reproduzierbar sein. Im Text werden diese in nicht proportionaler Schrift (Schreibmaschinenschrift) dargestellt, z. B.:

```
>>> print('hallo welt')
hallo welt
```

Dies gilt auch für alle Schlüsselwörter, Funktions- und Variablennamen aus den Programmen im Text, z. B. Variable `foo` muss an die Funktion `bar()` übergeben werden.

Listings sind durch einen Rahmen vom übrigen Text abgesetzt und nur zur besseren Referenzierung mit Zeilennummern versehen (Python verwendet keine Zeilennummern).

```
1 #!/bin/env python
2 print('Hallo Welt')
```

Listing 1.1. Beispiellisting: Darstellung von Quelltext

HINWEIS: In allen Listings wird für eine bessere Lesbarkeit auf eine Fehlerbehandlung verzichtet.

[1] Auf MacOS und Windows gerne als Terminal bezeichnet.

1.2 Python

Python arbeitet mit einem Interpreter in dem einzelne Anweisungen eingegeben und ausgeführt werden können. Nach dem Start zeigt das Programm seine Eingabebereitschaft mit drei „größer-als“ Zeichen an:

```
$ python
Python 3.11.8 (main, Feb 28 2024, 00:00:00) [GCC 13.2.1 20231011
(Red Hat 13.2.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

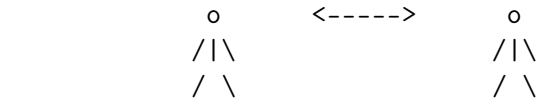
Der Python-Interpreter wird durch `python`[2] in der ersten Zeile gestartet (Das Dollarzeichen stellt die Aufforderung zur Eingabe der Shell dar). Nach Programminformationen erscheint der Eingabeprompt von Python. Hier können nun Anweisungen eingegeben werden.

[2] Je nach System kann/muss der Programmname `python` um eine Versionsnummer ergänzt werden. Auf älteren Systemen startet so Python in der Version 2.

2 Grundlagen

2.1 Kommunikation

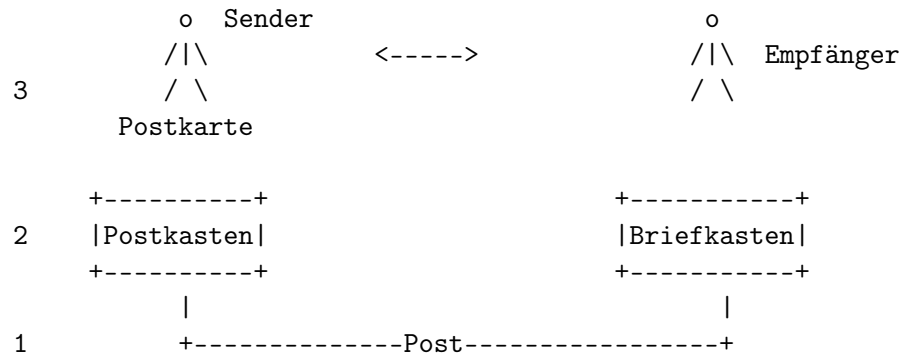
Wie kommunizieren wir? Im einfachsten Fall reden wir miteinander von Mensch zu Mensch. Eine einfache Darstellung könnte wie folgt aussehen.



Wenn die Entfernung etwas größer wird kommen Hilfsmittel ins Spiel, angefangen mit Brief und Postkarte über Telefon bis zur E-Mail über das Internet.

In technischen Beschreibungen wird Kommunikation für jeden Teilnehmer als ein Stapel von mehreren übereinanderliegenden Schichten dargestellt. Jede Schicht nutzt die Dienste der jeweils darunterliegende Schicht zum Versenden. Eingehende Daten werden an die darüberliegende Schicht weitergegeben.

Als einfaches Beispiel kann der Versand einer Postkarte dienen. Der Text auf der Karte stellt die zu übertragenden Daten dar. Auf der Karte gebe ich die Adresse des Empfängers an.



Meine Schnittstelle zum Nachrichtenversand ist der Briefkasten im Ort. Wenn ich die Karte einwerfe ist die Nachricht für mich versendet und ich kann nur hoffen, dass sie möglichst schnell den Empfänger erreicht.

Die Post holt die Karte aus dem Briefkasten und stellt sie der darauf angegebenen Adresse zu. Wie das geschieht braucht mich nicht zu interessieren.

Irgendwann landet die Karte im Briefkasten des Empfängers, aus dem sie dann entnommen werden kann. Die Nachricht ist übertragen und damit für den Empfänger lesbar. Das folgende Bild versucht ein Schichtenmodell für den beschriebenen Postweg darzustellen:

Die Zahlen links stellen die schon erwähnten Schichten dar (Layer, hier 1-3, es wird immer von unten gezählt). Ein häufig genutztes Modell von der ISO/OSI verwendet sieben Schichten.

2.2 TCP/IP

In Computernetzen gibt es keine Briefkästen. Im Fall von TCP/IP ist die Schnittstelle für den Programmierer das Socket-API[1]. Dies ermöglicht mir den Zugriff auf Funktionen zum Herstellen einer Verbindung und zum Versand von Daten über das Internet.

Die Bezeichnung TCP/IP nennt zwei aufeinander aufbauende Protokolle. TCP setzt auf IP auf. Auch andere Protokolle verwenden IP als darunter liegende Schicht. UDP[2] ist eine solche Alternative.

TCP und UDP unterscheiden sich in der Verbindungsqualität. TCP garantiert die korrekte Übermittlung der Daten in der richtigen Reihenfolge und treibt dafür einen erheblichen Aufwand. UDP ist mehr auf schnellen Verbindungsaufbau und hohen Durchsatz getrimmt. Zuverlässige Zustellung ist nicht so wichtig, Datenverlust ist möglich.

TCP ist ähnlich einem Telefonanruf. Nach dem Wählen und dem Verbindungsaufbau höre ich meinen Gegenüber, melde mich und danach beginnt das Gespräch. Häufig wird im Gespräch ein „ja“ oder „hm“ als Bestätigung (Quittung) verwendet.

UDP ist mehr das Modell „Postkarte“: Adresse auf die Karte schreiben und ab damit in den Postkasten - sie wird schon irgendwie ankommen, eine Anwort ist nicht nötig bzw. wird nicht unbedingt erwartet.

2.3 Adressen im Internet

Jeder hat sicher schon mal Adressen der Form 1.2.3.4 gesehen. Dies sind IP-Adressen[3] in der sogenannten „dotted notation“. Hier sind jeweils acht Bit der Adresse als Dezimalzahl dargestellt und diese sind durch Punkte getrennt. Jede Stelle kann Werte im Bereich von 0 bis 255 enthalten. Diese Darstellung ist schon deutlich besser als z. B. 2130706433 für die Adresse 127.0.0.1.

Um sich nicht viele dieser Zahlenkolonnen merken zu müssen wurde eine Möglichkeit geschaffen den Zahlen Namen zuzuordnen. Kaum jemand verwendet eine IP-Adresse in der URL-Zeile seines Browsers. Stattdessen wird ein Name im „Telefonbuch des Internet“ nachgeschlagen. Das Ergebnis einer Suche ist die IP-Adresse eines Rechners[4]. Das Protokoll dafür heißt DNS – Domain Name System. Die Suche läuft für den Programmierer unsichtbar bei einem Verbindungsaufbau mit einen Namen ab. Nur im Fall eines Fehlers wird die Wichtigkeit dieses sonst unsichtbaren Dienstes offenbar.

Für die Adressierung reicht die IP-Adresse alleine nicht aus. Als weitere Angabe ist der Port erforderlich. Diese Zahl ist sozusagen die Wohnungsnummer im Hochhaus und liegt im Bereich 1 - 65535.

[1] API – Application Programming Interface: Die Beschreibung der nutzbaren Funktionen einer Programmsammlung.

[2] User Datagram Protocol

[3] Genaugenommen Adressen von IPv4, der Version 4 des Internet Protokolls. Adressen der Version 6 sind erheblich länger und noch viel schlechter zu merken.

[4] Eine IP-Adresse gehört zu einem Netzwerkinterface eines Rechners. Auf einem Interface können mehrere, auch von unterschiedlichen IP-Versionen, eingerichtet sein. Es ist durchaus üblich, Zugang zu einem Rechner über IPv4- und IPv6-Adressen zu haben.

Für viele Dienste sind die Ports vorgegeben, z. B. 25 für `SMTP`, 80 für `http` und 443 für `https`.[5] Eine Besonderheit stellen die Nummern bis 1024 bei der Verwendung in einem Server dar. Diese Nummern erfordern `root`-Rechte um sie zu verwenden.

2.4 telnet

Mit seiner unverschlüsselten Datenübertagung gilt `telnet` inzwischen als nicht mehr zeitgemäß und sollte nicht mehr verwendet werden. Allerdings hat es eine Eigenschaft, die es zu einem praktischen Tool im Werkzeugkasten macht: Damit kann zum Testen schnell eine Verbindung zu einer beliebigen Gegenstelle aufgebaut werden.[6]

Ein lokal laufender Webserver kann damit angesprochen werden. Die Adresse ist entweder „127.0.0.1“, „::1“ oder „localhost“, der Port 80. Mit diesen Werten kann `telnet` aufgerufen werden:

```
$ telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET / HTTP/1.1
HOST: www.example.com

HTTP/1.1 200 OK
Date: Wed, 13 Mar 2024 17:51:03 GMT
Server: Apache/2.4.58 (Fedora Linux)
Last-Modified: Sun, 14 Jan 2024 07:58:28 GMT
ETag: "e78-60ee34336d710"
Accept-Ranges: bytes
Content-Length: 3704
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html>
...
</body></html>
Connection closed by foreign host.
```

Ein Webserver wartet nach dem Verbindungsaufbau auf eine Eingabe. Diese ist hier zwei Zeilen gefolgt von einer Leerzeile:

```
GET / HTTP/1.1
```

in der ersten Zeile. In der zweiten Zeile wird mit `HOST:` der gewünschte Server angegeben.

[5] Die Liste der *well known ports* ist auf UNIX-Systemen in `/etc/services`. Hier ist auch vermerkt welches Protokoll (TCP/UDP) verwendet wird.

[6] Telnet basiert auf TCP, also können nur Server mit diesem Protoll angesprochen werden.

2.5 ss – socket statistics

Anzeige von Netzwerkinfonrmationen: Welche Prozesse nutzen Sockets, welche Verbindungen bestehen, in welchem Zustand sind sie?

```
$ ss -tpln | grep LISTEN
LISTEN 0      4096         127.0.0.1:631          0.0.0.0:*
LISTEN 0      4096           0.0.0.0:5355         0.0.0.0:*
LISTEN 0      4096         127.0.0.1:8384         0.0.0.0:*
users:(("syncthing",pid=1957,fd=18))
LISTEN 0      4096     127.0.0.53%lo:53           0.0.0.0:*
LISTEN 0      4096        127.0.0.54:53           0.0.0.0:*
LISTEN 0      4096             [::1]:631             [::]:*
LISTEN 0      4096              [::]:5355            [::]:*
LISTEN 0      4096                 *:22000              *:*
users:(("syncthing",pid=1957,fd=12))
```

Die Parameter bei dem Aufruf von `ss` sind:

- t – TCP
- u – UDP
- p – zugehöriger Prozess
- l – Sockets im Zustand `LISTEN`
- n – Service nur numerisch, nicht als Name

Mit Namensauflösung (also ohne Parameter `n`):

```
$ ss -tpl | grep LISTEN
LISTEN 0      4096         127.0.0.1:ipp              0.0.0.0:*
LISTEN 0      4096           0.0.0.0:hostmon          0.0.0.0:*
LISTEN 0      4096         127.0.0.1:8384             0.0.0.0:*
LISTEN 0      4096    127.0.0.53%lo:domain           0.0.0.0:*
LISTEN 0      4096       127.0.0.54:domain           0.0.0.0:*
LISTEN 0      4096            [::1]:ipp                [::]:*
LISTEN 0      4096             [::]:hostmon            [::]:*
LISTEN 0      4096                *:snapenetio            *:*
```

Mit `root`-Rechten entfaltet das Programm seine volle Wirkung, normale Benutzer sehen häufig nur allgemeine Informationen oder die der eigenen Prozesse. Im Zweifelsfall hilft ein Blick in die Manpage des Programms.

2.6 tcpdump

Ein unheimlich praktisches Tool zur Anzeigen oder Mitschneiden des Netzwerkverkehrs. Die Darstellung findet in einer Shell statt, ist also nicht besonders komfortabel. Ein Mitschnitt der Anfrage an den Webserver von vorhin geht wie folgt (Abbruch der Aufzeichnung mit `Ctrl + C`):

```
# tcpdump -i lo -p tcp port 80
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
18:26:51.677820 IP6 localhost.54274 > localhost.http: Flags [S],
seq 2316672992, win 33280, options [mss 65476,sackOK,TS
val 3932261990 ecr 0,nop,wscale 7], length 0
18:26:51.677846 IP6 localhost.http > localhost.54274: Flags [S.], ...
18:26:51.677863 IP6 localhost.54274 > localhost.http: Flags [.], ...
18:27:04.512241 IP6 localhost.54274 > localhost.http: Flags [P.], ...
18:27:04.512307 IP6 localhost.http > localhost.54274: Flags [.], ...
18:27:11.296411 IP6 localhost.54274 > localhost.http: Flags [P.], ...
18:27:11.296449 IP6 localhost.http > localhost.54274: Flags [.], ...
18:27:11.779644 IP6 localhost.54274 > localhost.http: Flags [P.], ...
18:27:11.779681 IP6 localhost.http > localhost.54274: Flags [.], ...
18:27:11.780118 IP6 localhost.http > localhost.54274: Flags [P.], ...
18:27:11.780138 IP6 localhost.54274 > localhost.http: Flags [.], ...
18:27:16.786113 IP6 localhost.http > localhost.54274: Flags [F.], ...
18:27:16.786312 IP6 localhost.54274 > localhost.http: Flags [F.], ...
18:27:16.786358 IP6 localhost.http > localhost.54274: Flags [.], ...
^C
14 packets captured
28 packets received by filter
0 packets dropped by kernel
```

Um den Vorgang in eine Datei zu speichern:

```
# tcpdump -i lo -s 65535 -p tcp port 80 -w http.dump
dropped privs to tcpdump
tcpdump: listening on lo, link-type EN10MB (Ethernet), snapshot \
length 65535 bytes
^C11 packets captured
22 packets received by filter
0 packets dropped by kernel
```

2.7 wireshark

`wireshark` ist `tcpdump` mit einer GUI. Neben der Capture-Funktion und hübschen Darstellung der Daten sind noch eine Vielzahl von Tools zur Analyse des Netzwerkverkehrs eingebaut.

Wireshark kann die Dateien von `tcpdump` lesen. Auch die so beliebten Fritzboxen können den Netzwerkverkehr in diesem Format aufzeichnen. Auch hier bietet sich dann die Darstellung und Analyse mit `wireshark` an.

3 Die Praxis

3.1 Der erste Server

An der Kommunikation sind immer zwei Parteien beteiligt. In der Programmierung führt das zu kleinen Unterschieden, je nachdem welche Seite ein Programm darstellt. Der sogenannte Server wartet auf eingehende Anfragen eines Clients[1]. Die grundlegenden Funktionen eines Server sind in Python mit ein paar Zeilen Code schnell realisiert:

```
1  import socket
2
3  host = 'localhost'
4  port = 50000
5
6  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7  s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
8  s.bind((host, port))
9  s.listen(1)
10 con, addr = s.accept()
```

Listing 3.1. socketserver_01.py: Der erste Server-Socket

Listing 3.1 zeigt alles erforderliche um auf Empfang zu gehen.

Zeile 1: `import socket` bindet die Schnittstelle zum Socket-API ein[2].

Zeilen 3 und 4: Zwei Variablen[3] werden mit einem Wert belegt: `host` und `port` bilden zusammen die Adresse des Servers. Diese Daten benötigt ein Client zur Kontaktaufnahme.

Zeilen 6–9: Dies ist für jeden Server-Socket erforderlich. Dies richtet den Kommunikationsendpunkt auf dem Rechner ein.

Zeile 10: Startet den Empfang.

Das Programm blockiert an dieser Stelle und wartet auf Daten (Der Python-Prompt >>> verschwindet, Eingaben werden nicht mehr sofort dargestellt).

[1] In der Kommunikationstheorie wird dieses Modell als Client-Server-Architektur bezeichnet.
[2] Mehr zum `import` in Kapitel 4.1 auf Seite 27.
[3] Variable: Ein Name für einen Wert.

Die schmutzigen Details

Die Zeilen 6–10 des Programms 3.1 wurden zunächst nur kurz erläutert. Wer an den Details interessiert ist kann jetzt gerne weiterlesen, alle anderen können bis zum Kapitel 3.1.1 auf Seite 15 weiterblättern.

Um einem Server-Socket für die Kommunikation zu erzeugen sind einige Schritte erforderlich, gehen wir die Zeilen 6 bis 10 einzeln durch.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Dies ist die Initialisierung eines Socket-Objekts durch den Aufruf der Funktion[4] `socket()` aus dem Modul `socket`, das in Zeile 1 importiert wurde.

Dem Funktionsnamen folgt ein Paar runde Klammern. Darin befinden sich evtl. nötige Parameter. Im Fall der Funktion `socket()` sind es zwei:

- Das „Adressformat“ – Eine Angabe in welchem Adressbereich dieser Socket arbeiten soll. In unserem Fall ist dies ein Wert für die globale Adressierung. Alternativ könnte ein lokaler Socket mit `AF_UNIX` erzeugt werden.
- Die Art des Socket – Dies ist die Wahl des Protokolls. Der genutzte Wert sorgt für die Verwendung von TCP. Eine andere Option ist `SOCK_DGRAM` für das Protokoll UDP.

Alle möglichen Werte für die beiden Parameter sind im Modul `socket` definiert. Die Auswahl hier entscheidet auch über einige der nachfolgend erforderlichen Funktionsaufrufe. Beispiele dazu in Kapitel 3.3. Das Ergebnis ist ein Socket-Objekt in der Variablen `s` mit dem alle weiteren Aktionen ausgeführt werden.

```
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Zeile 7 ist eine vorbeugende Maßnahme und nicht unbedingt erforderlich. Bei der Freigabe einer Socket-Adresse halten Betriebssysteme gewisse Wartefristen ein, um eventuell später eintreffende Daten behandeln zu können. Während dieser Zeit kann diese Adresse normalerweise nicht erneut genutzt werden. Nun ist es aber vollkommen normal ein Programm, z. B. wegen einer Änderung in der Konfiguration, neu zu starten. Dies wäre ohne die gezeigte „Socket-Option“ nicht möglich. `SO_REUSEADDR` zeigt dem System die gewünschte Wiederverwendung an und verhindert eine Fehlermeldung bei zu schnellem Neustart.

```
s.bind((host, port))
```

Die Funktion `bind()` teilt dem Betriebssystem mit, auf welcher Adresse und welchem Port der Socket eingerichtet werden soll. Hier kommen die zuvor definierten Variablen `host` und `port` zum Einsatz.

Als Adresse kann eine oder mehrere Adressen des Rechners genutzt werden. Diese können als Liste von Zeichenketten angegeben werden.

Eine besondere Adresse stellt `localhost` dar. Diese ist auf jedem Rechner mit funktionierendem TCP/IP vorhanden. Bei IPv4 ist es `127.0.0.1`, bei IPv6 `::1`.

[4] Funktion: Ein Stück Programm, welches mit ein paar Informationen versorgt eine bestimmte Aufgabe erledigt. Der Name dient wie bei Variablen als Referenz.

Um einen Socket auf allen vorhandenen Adressen zur Verfügung zu stellen gibt man eine leere Zeichenkette als `host` an.

Ports können im Bereich 1-65355 liegen, allerdings sind sie bis 1024 dem Superuser vorbehalten. Programme ohne die erforderlichen Berechtigungen erhalten eine Fehlermeldung.

```
s.listen(1)
```

Die Zahl beim Aufruf von `listen()` beeinflusst die maximal gleichzeitig vom Betriebssystem zu verwaltenden Verbindungsanfragen. Dies ist dabei nur ein Parameter von vielen im Betriebssystem für diese Funktion. Eine kleine Zahl sollte hier für die meisten Anwendungen ausreichen.

Dies waren schon die Vorbereitungen für einen Serversocket.

```
con, addr = s.accept()
```

Mit dem Aufruf von `accept()` werden eingehende Verbindungen erwartet. Die Funktion liefert die Daten der Verbindung oder leere Werte im Fehlerfall.

3.1.1 Der erste Kontakt

Um nicht gleich einen Client zu programmieren zu müssen reicht für einen ersten Test ein Webbrowser oder das Programm „telnet“:

- Webbrowser: In der Adresszeile „localhost:50000“ eingeben und abschicken.
- Telnet: In einem Terminal „telnet localhost 50000“ eingeben und ausführen.

Das Ergebnis auf Serverseite ist die Fortsetzung des Programms, d. h. der Python-Interpreter zeigt wieder den Eingabeprompt und kann weitere Anweisungen entgegennehmen. An dieser Stelle geht es gleich weiter mit dem Code zum Empfang von Daten und deren Verarbeitung.

Je nachdem welcher Client beim Test zum Einsatz (Browser oder telnet) kam ist nun etwas Aufräumarbeit nötig.

- Der Browser zeigt „waiting for localhost“ weil unser Programm noch nichts sendet. Dies kann mit dem „Stop“-Button abgebrochen werden. Alternativ kann auf den Timeout gewartet werden, dieser beträgt normalerweise eine Minute.
- Die Telnet-Verbindung ist zu stande gekommen und wartet nun auf weitere Eingaben. In einer Linux-Shell kann die Verbindung durch Drücken von `ctrl + 5` mit der anschließenden Eingabe von `quit` beendet werden.

3.1.2 Mit wem spreche ich?

Die Funktion `accept()` kehrt nach einem erfolgreichen Verbindungsaufbau mit Informationen über den Kommunikationspartner zurück. Die Rückgabewerte stellen ein Objekt für die gerade zustande gekommene Verbindung und Adressinformationen zur Gegenstelle dar:

```
>>> con
<socket.socket fd=4, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 50000),
raddr=('127.0.0.1', 40390)>
>>> addr
('127.0.0.1', 40390)
```

In diesem Beispiel besteht die Verbindung zur IP-Adresse 127.0.0.1 und Port 40390. Diese Information ist sowohl in `con` als auch in `addr` enthalten (`raddr` in `con`).

3.1.3 Daten empfangen: Telnet

Das `con`-Objekt verfügt über die Methoden `recv()` und `send()` zum Empfangen und Senden.

Der Client kann über die Telnet-Verbindung eine Zeichenfolge, z. B. `abc`, an den Server senden (Die Daten werden mit einem `return` abgeschickt):

```
$ telnet localhost 50000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
abc
```

Daten auf der Verbindung werden mit `recv()` vom Socket gelesen (`recv()` blockiert wenn keine Daten verfügbar sind). Dabei ist die maximal zu lesende Zahl von Bytes anzugeben. Auf der Serverseite geht es also wie folgt weiter:

```
>>> data = con.recv(10)
>>> data
b'abc\r\n'
```

Die empfangenen Daten werden als Bytefolge (zu erkennen an dem `b` vor der Zeichenkette) geliefert. Die Steuerzeichen[5] für das `return` ist in den Daten enhalten: `\r\n`.

3.1.4 Daten empfangen: Browser

Der Webbrowser sendet schon etwas mehr Daten nachdem die Verbindug steht:

```
>>> data = con.recv(4096)
>>> data
b'GET / HTTP/1.1\r\nHost: 127.0.0.1:50000\r\nUser-Agent: Mozilla/5.0
(X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
SeaMonkey/2.49.4\r\nAccept: text/html,application/xhtml+xml,
application/xml;q=0.9,*/*;q=0.8\r\nAccept-Language: en-US,en;q=0.5\r\n
Accept-Encoding: gzip, deflate\r\nDNT: 1\r\nConnection:
keep-alive\r\nUpgrade-Insecure-Requests: 1\r\n\r\n'
```

[5] Diese Zeichen sind normalerweise nicht sichtbar.

3.1.5 Bytes, Zeichen und deren Darstellung

Die Bytefolge vom Socket kann direkt weiterverarbeitet werden. Da es sich in beiden Fällen um Textdaten handelt empfiehlt sich eine Konvertierung in eine Zeichenkette. Dies erledigt die Methode `decode()` mit der Angabe der gewünschten Darstellung. Die heutzutage am häufigsten genutzte Codierung von Text ist UTF-8 (Unicode):

```
>>> data.decode('utf-8')
'abc\r\n'
```

Eine Zeichenkette wird in Python durch Anführungszeichen eingeschlossen. Dies können Einfache oder Doppelte sein.

Das Gegenstück zu `decode()` heisst `encode()`. Mehr dazu beim Senden von Daten.

3.1.6 Zerlegen der Daten

Die Anfrage des Browsers ist ein HTTP-Request (der Abruf einer Webseite). Dieses Protokoll verwendet Textnachrichten. Einzelne Zeilen sind durch Return- und Newline-Zeichen getrennt. Nach der Konvertierung in Text können die einzelnen Zeilen leicht getrennt werden:

```
>>> sdata = data.decode('utf-8')
>>> sdl = sdata.split('\r\n')
>>> sdl
['GET / HTTP/1.1', 'Host: 127.0.0.1:50000',
 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0)
  Gecko/20100101 Firefox/52.0 SeaMonkey/2.49.4',
 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
 'Accept-Language: en-US,en;q=0.5',
 'Accept-Encoding: gzip, deflate', 'DNT: 1',
 'Connection: keep-alive',
 'Upgrade-Insecure-Requests: 1', '', '']
```

Nach der Umwandlung der empfangenen Daten in eine Zeichenkette wird durch `split()` eine Trennung in einzelne Zeilen durchgeführt. Der Funktion erhält die zu nutzenden Trennzeichen. Das Ergebnis der Funktion ist eine Liste[6].

Die erste Zeile eines HTTP-Requests muss die verwendete Methode, einen URL und die Protokollversion enthalten. Dies kann leicht durch einen Regulären Ausdruck (Regex[7]) überprüft werden. Die erste Zeile im Beispiel besteht aus:

```
GET / HTTP/1.1
```

Um den Regulären Ausdruck einfach zu halten wird nur die GET-Methode zugelassen. Der URL soll aus beliebigen Buchstaben, Ziffern, dem Binde- und Unterstrich sowie dem / bestehen. Das Protokoll wird in der Form `HTTP/1.x` angegeben.

[6] Liste in Python: Durch Komma getrennte Werte umgeben von eckigen Klammern. Die einzelnen Elemente können über einen Index angesprochen werden. Das erste Element hat den Index Null.

[7] Reguläre Ausdrücke: Damit werden Zeichenfolgen beschrieben um Vergleiche, das sogenannte „Pattern Matching“, durchzuführen.

```
>>> import re
>>> rec_REQ_CHK = re.compile('^GET /[a-zA-Z0-9\-_/]* HTTP/1\.[01]$')
>>> m = rec_REQ_CHK.match(sdl[0])
>>> m
<re.Match object; span=(0, 14), match='GET / HTTP/1.1'>
```

Wenn das erste Element der Liste `sdl` dem Regex entspricht enthält die Variable `m` das Ergebnis des Pattern Matching. Falls die Zeichenkette den Test nicht besteht ist die Variable leer. Die weiteren Aktionen des Programms können also durch einen Test der Variablen `m` gesteuert werden.

3.1.7 Daten senden: An einen Browser

Die Methode `send()` eines Verbindungs-Objekts dient zum Versand von Daten über einen Socket.

Ein Webbrowser erwartet nach seinem Request eine Antwort in einem bestimmten Format. Diese wird als HTTP Response bezeichnet.

Die Antwort besteht aus einer Statuszeile, beliebigen Headerzeilen, zwei Leerzeilen und den eigentlichen Daten.

Die Statuszeile besteht aus der HTTP-Version, dem Status-Code, einer Textbeschreibung des Statuscode gefolgt von Return und Newline. Damit der Browser den Inhalt der Antwort richtig interpretieren kann wird noch der „content-type“-Header in die Antwort aufgenommen:

```
status = 'HTTP/1.0 200 OK\r\ncontent-type: text/html\r\n\r\n'
```

Der Typ „text/html“ beschreibt die Daten als Text mit der genaueren Bezeichnung als HTML-Text. Als Daten der Antwort wird eine minimale Webseite mit einer kurzen Nachricht definiert:

```
html = '<!doctype html><html><head><title>Antwort</title></head>'
html += '<body><h1>Hallo Welt!</h1></body></html>'
```

Die beiden Teile werden zusammengefügt, in eine Bytefolge umgewandelt und an den Client gesendet. Da HTTP ein zustandsloses Protokoll ist, kann nach dem Senden die Verbindung geschlossen werden[8]:

```
data = status + html
bdata = data.encode('utf-8')
bytes_sent = con.send(bdata)
con.close()
```

Der Browser sollte nach dem Senden den Text „Hallo Welt!“ anzeigen. Die Methode `send()` liefert die Anzahl der gesendeten Bytes zurück.

[8] In der Praxis wird die Verbindung offen gehalten und zum Senden der nächsten Anfragen genutzt – Stichwort „keepalive“.

3.1.8 Beenden einer Verbindung

Wenn alle Daten ausgetauscht wurden kann die Verbindung geschlossen werden. Dies erledigt die Methode `close()` auf dem aktiven Verbindungsobjekt.

```
s.close()
```

Danach gibt das Betriebssystem die Resourcen frei und nimmt keine neue Verbindungen mehr auf dieser Adresse an.

3.1.9 Komplettes Serverprogramm

Hier ist noch einmal das komplette Server-Programm:

```
1  import re
2  import socket
3
4  rec_REQ_CHK = re.compile('^GET .+ HTTP/[0-9.]+$')
5
6  host = "localhost"
7  port = 50000
8
9  status = 'HTTP/1.0 200 OK\r\ncontent-type: text/html'
10 html = '<!doctype html><html><head><title>Antwort</title></head>'
11 html += '<body><h1>Hallo Welt!</h1></body></html>'
12 data = status + '\r\n\r\n' + html
13 bdata = data.encode('utf-8')
14
15 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
16 s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
17 s.bind((host, port))
18 s.listen(1)
19 con, addr = s.accept()
20 data = con.recv(4096)
21
22 data = data.decode('utf-8')
23 dl = data.split('\r\n')
24 request = dl[0]
25
26 m = rec_REQ_CHK.match(request)
27 if m:
28     bytes_sent = con.send(bdata)
29
30 con.close()
31 s.close()
```

Listing 3.2. socketserver_02.py: Der komplette Code für den Server

3.2 Ein Client

Nun zur anderen Seite der Kommunikation. Ein Webbrowser oder Telnet als Client ist für einfache Testfälle und die schnelle Prüfung von Verbindungen vielleicht ausreichend, langfristig ist ein individueller Kommunikationsteilnehmer klar im Vorteil.

Der Client kann bei Kenntnis der Serverdaten (Name oder IP-Adresse und Port[9]) eine Verbindung aufbauen.

```
1 import socket
2
3 host = 'localhost'
4 port = 50000
5
6 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7 s.connect((host, port))
8 s.send('GET / HTTP/1.0\r\n'.encode('utf-8'))
```

Listing 3.3. TCP-Socket Client

Listing 3.3 zeigt die nötigen Schritte um einen Socket zu erzeugen (Zeile 6) und die Verbindung zum Server aufzubauen. Nach dem Aufruf von `connect()` (Zeile 7) steht die Verbindung und Daten können mit `send()` gesendet werden (Zeile 8). Die Nachricht ist ein HTTP-Request. Ein Browser macht nichts anderes.

Natürlich könnte der Client auch einfach Daten vom Server erwarten, dies hängt ganz vom verwendeten Protokoll ab – Client und Server müssen die gleiche Sprache sprechen.

Zum Lesen von Daten verwendet der Client die gleiche Funktion wie der Serversocket: `recv()`. Für die gezeigten Programme erwartet der Client die Antwort des Servers wie folgt:

```
bdata = s.recv(4096)
data = bdata.decode('utf-8')
```

Die Antwort ist die minimale Webseite des Servers:

```
>>> data
'HTTP/1.0 200 OK\r\ncontent-type: text/html\r\n\r\n<!doctype
html><html><head><title>Antwort</title></head><body><h1>Hallo
Welt!</h1></body></html>'
```

Nachdem alle eroforderlichen Daten ausgetauscht sind muss der Socket vom Client mit `close()` geschlossen werden.

3.3 Andere Socketvarianten

Sockets können für unterschiedliche Adressformate und Übertragungsqualitäten erzeugt werden. Das Adressformat entscheidet z. B. ob der Socket im Internet (`AF_INET`) oder nur auf dem Rechner (`AF_UNIX`) erreichbar ist.

[9] Bei UNIX-Sockets ist der Pfad die vollständige Adresse.

Die Qualität der Datenübertragung kann zwischen „alle Pakete kommen garantiert in der richtigen Reihenfolge an“ (`TCP/SOCK_STREAM`) und „Reihenfolge ist nicht so wichtig, ein Verlust ist auch nicht schlimm“ (`UDP/SOCK_DGRAM`) gewählt werden.

3.3.1 UDP-Socket Server

UDP (User Datagram Protocol) bietet keine Transportsicherung, Pakete können in beliebiger Reihenfolge ankommen und können sogar verloren gehen. Dafür ist der Verbindungsaufbau einfach und der Aufwand bei der Verarbeitung der Pakete ist geringer.

```
1  import socket
2  host = 'localhost'
3  port = 50000
4
5  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
6  s.bind( (host, port) )
7  data, addr = s.recvfrom(256)
8  if data:
9      print('\nDaten', data.decode('utf-8'), 'von', addr[0], addr[1])
10 else:
11     print('no data', addr[0], addr[1])
12
13 s.close()
```

Listing 3.4. Minimaler UDP-Server

Hier wird direkt nach dem Einstellen der Adresse mit `bind()` in den Empfangsmodus geschaltet. `recvfrom()` ist hier die Funktion zum Empfangen von Daten auf dem Socket.

3.3.2 UDP-Socket Client

```
1  import socket
2  host = 'localhost'
3  port = 50000
4
5  data = '1' * 18
6  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
7  bytes_sent = s.sendto(data.encode('ascii'), (host, int(port)))
8  s.close()
```

Listing 3.5. Minimaler UDP-Client

Auch im Client ist der Code kürzer. Direkt nach der Erzeugung des Socket können mit `sendto()` Daten verschickt werden.

3.3.3 UNIX-Socket Server mit TCP

Diese Sockets sind nur auf dem Rechner erreichbar auf dem sie erzeugt wurden. Es handelt sich dabei um Dateien im Filesystem[10]. Zeile 2 in Listing 3.6 definiert den Pfad für den Socket. Der Programmierer muss ein für ihn schreibbares Verzeichnis wählen.

```
1  import socket
2  socket_pfad = '/tmp/unixsocket_tcp'
3
4  server = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
5  server.bind(socket_pfad)
6  server.listen(1)
7  con, client_addr = server.accept()
8  data = con.recv(256)
9  print(data)
10 con.close()
11 server.close()
12
13 import os
14 os.unlink(socket_pfad)
```

Listing 3.6. Stream-UNIX-Socket-Server

Socketdateien sind in der Langform eines Listing in der Shell an einem `s` zu Beginn der Berechtigungen zu erkennen:

```
$ ls -l /tmp/
srwxrwxr-x 1 501 501  0 Jan 1 19:40 unixsocket_tcp
```

Die Datei wird bei Programmende im Gegensatz zu Stream-Sockets nicht vom Betriebssystem entfernt (siehe `SO_REUSEADDR` im Kapitel 3.1 auf Seite 14). Die Socketoption kann auf einen UNIX-Socket gesetzt werden, es bleibt aber ohne Effekt.

Die evtl. vorhandene Datei wird zum Problem beim erneuten Programmstart:

```
>>> import socket
>>> socket_pfad = '/tmp/unixsocket_tcp'
>>> server = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
>>> server.bind(socket_pfad)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 98] Address already in use
```

Bei der Nutzung eines UNIX-Sockets bieten sich zwei Wege für die Initialisierung an:

- Strukturierte Programmierung – Auswertung der Rückgabewerte der einzlnen Funktionen und Entscheidung je nach erhaltenen Informationen.

[10] Die Dateien können ganz normal bearbeitet und gelöscht werden.

- Aktionen mit Fehlerbehandlung – Einfach erstmal machen und ggf. in der Fehlerbehandlung reagieren. Kapitel 4.2 auf Seite 28 geht detailierter auf das Exception-Handling von Python ein.

Strukturierter Ansatz

```
1  import os
2  import stat
3  import sys
4
5  socket_pfad = '/tmp/unixsocket_tcp'
6  if os.path.exists(socket_pfad):
7      s = os.stat(socket_pfad)
8      if stat.S_ISSOCK(s.st_mode):
9          print('removing old socket {}'.format(socket_pfad) )
10         os.unlink(socket_pfad)
11     else:
12         print( '{} - no socket, please check!'.format(socket_pfad) )
13         sys.exit()
```

Listing 3.7. UNIX-Socket strukturiert erzeugen

Das Programm prüft, ob die Socketdatei existiert. Anschließend wird der Dateityp geprüft und dann gelöscht. Der ganze Block wird nicht ausgeführt wenn die Socket-Datei nicht vorhanden ist, das Programm kann dann einfach den Socket erzeugen.

Zur Sicherheit sollte der Aufruf von `os.unlink()` mit einem Exception-Handling versehen werden. Allerdings schadet ein Programmabbruch an dieser Stelle auch nicht, da der Socket vom Programm benötigt wird.

Erstmal machen

Dieser Weg löscht einfach den Socket – Zumindest versucht er es:

```
1  import stat
2  import os
3
4  socket_pfad = '/tmp/unixsocket_tcp'
5  try:
6      s = os.stat(socket_pfad)
7      if stat.S_ISSOCK(s.st_mode):
8          print('removing old socket {}'.format(socket_pfad) )
9          os.unlink(socket_pfad)
10 except FileNotFoundError:
11     pass
12 except EXCEPTION as e:
13     print( '{}'.format(e) )
```

Listing 3.8. UNIX-Socket mit Exception-Handling erzeugen

Beim Aufruf von `os.stat()` könnte ein Fehler bei nicht vorhandener Datei ausgelöst werden. Für die Programmlogik wäre dies aber kein Fehler, da die Datei sonst sowieso gleich gelöscht würde. Deshalb führt dieser Zweig der Fehlerbehandlung auch keine Aktion aus (`pass` nach `FileNotFoundError` in Zeile 11). Alle anderen Fehler werden einfach ausgegeben.

3.3.4 UNIX-Socket Client mit TCP

```
1 import socket
2 socket_pfad = '/tmp/unixsocket_tcp'
3
4 client = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
5 client.connect(socket_pfad)
6 bytes_sent = client.send(b'Hallo Welt!')
7 client.close()
```

Listing 3.9. Stream UNIX-Socket-Client

3.3.5 UNIX-Socket Server mit UDP

```
1  import socket
2  socket_pfad = '/tmp/unixsocket_udp'
3
4  server = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
5  server.bind(socket_pfad)
6  data, addr = server.recvfrom(1024)
7  server.close()
8
9  import os
10 os.unlink(socket_pfad)
```

Listing 3.10. UNIX-Socket Server mit UDP

3.3.6 UNIX-Socket Client mit UDP

```
1 import socket
2 socket_pfad = '/tmp/unixsocket_udp'
3
4 client = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
5 client.sendto(b'Hallo Welt!', socket_pfad)
```

Listing 3.11. UNIX-Socket Client mit UDP

3.4 Multiplexen

Nur eine Verbindung zur Zeit ist eine erhebliche Einschränkung, Zur Bearbeitung von mehreren Verbindungen muss der vorgestellte Code etwas geändert werden. Die folgenden vier Listings sind das gesamte Programm.

```
1 import socket
2
3 host = 'localhost'
4 port = 50000
5
6 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7 s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
8 s.bind((host, port))
9 s.listen(10)
```

Listing 3.12. Muliplex 1: Initialisierung Server-Socket

Soweit ist bis auf den geänderten Wert beim `listen()`-Aufruf alles bekannt.

Als nächstes kommt ein neuer Systemaufruf in Spiel: `select()`. Diese Funktion erwartet vier Parameter: Jeweils eine Liste mit Objekten von den Daten gelesen, geschrieben oder ein Fehler gemeldet werden soll und ein Wert für einen Timeout.

In die Leseliste wird der gerade erzeugt Socket eingefügt, die anderen Listen bleiben leer. Der Timeout wird auf fünf Sekunden gesetzt und dient der Kontrolle.

```
1 rlist = [s]
2 wlist = []
3 elist = []
4 timeout = 5
```

Listing 3.13. Muliplex 2: Listen der Sockets und Timeout

Dann kommen noch zwei Module zum Programm.

```
1 import time
2 import select
```

Listing 3.14. Muliplex 3: Erforderliche Module

Die eigentliche Funktion läuft in einer Endlosschleife. `select()` wartet auf ein Ereignis in einer der Listen oder den Timeout.

Der Rückgabewert in `readable` wird mit dem vorhandenen Socket verglichen. Wenn dieser Daten empfangen kann muss mit `accept()` eine neue Verbindung aufgebaut werden. Die Daten der Gegenstelle werden ausgegeben und an die Liste `rlist` angehängt damit im nächsten Durchlauf hier Daten empfangen werden können.

Jeder andere Socket hat Daten zum Lesen bereit. Diese werden mit `recv()` abgeholt. Wenn keine Daten verfügbar sind wurde dieser Socket geschlossen und er kann aus der Liste der Empfänger entfernt werden.

```
1  while True:
2      readable, writable, errored = select.select(rlist, wlist, elist, timeout)
3      if readable:
4          for sock in readable:
5              if sock is s:
6                  con, addr = s.accept()
7                  rlist.append(con)
8                  rlist.index(con)
9                  print('cnct from', addr)
10             else:
11                 data = sock.recv(1024)
12                 if data:
13                     pass  # keine Aktion
14                     # sock.send(data)
15                 else:
16                     sp = rlist.index(sock)
17                     sock.close()
18                     print('clsd', sp, sock)
19                     rlist.remove(sock)
20     else:  # not readable and not writeable and not errored
21         print( int( time.time() ) )
```

Listing 3.15. Muliplex 4: Senden und Empfangen mit Multiplex

Für eine tatsächliche Funktion muss in Zeile 13 und 14 das Kommentarzeichen `#` gesetzt bzw. entfernt werden. So gibt es keine Antwort für einen CLient.

Wenn kein Socket Daten verfügbar hat ist `select()` in den Timeout gelaufen. Hier wird zur Kontrolle die aktuelle Zeit als UNIX-Timestamp ausgegeben.

Neben `select()` gibt es noch eine Menge andere Funktionen zum Multiplexen von Verbindungen: `poll()`. `epoll()`, `kqueue()` um nur einige zu nennen. Diese sind zum Teil systemspezifisch und nicht auf jeder Plattform verfügbar.

Python bietet auch noch ein Modul mit einem höheren Abstraktionslevel `selektors`.

4 Python Referenz

Hier wird auf einige Besonderheiten von Python eingegangen, die für die Ziele des Textes nicht unbedingt erläutert werden müssen.

4.1 Importieren von Modulen

Python bietet eine Vielzahl von Modulen mit fertigen Funktionen für die unterschiedlichsten Einsatzgebiete. Es gibt zwei Wege Module mit ihren Funktionen und Daten zu importierten. Dies wirkt sich auf die Nutzung aus:

- `import socket` – Funktionen müssen bei diesem Vorgehen mit dem vollständigen Pfad aufgerufen werden: `modulname.funktionsname()`
- `from socket import socket` oder `from socket import *` – Dies lädt einzelne oder alle Funktionen eines Moduls. Diese können dann direkt mit ihrem Namen aufgerufen werden. Gerade die letzte Form sollte nur mit äußerster Vorsicht genutzt werden, da hier bereits existierende Funktionen ohne Warnung überschrieben werden können.

Punkt 1 ist der in den Listings beschriebene Weg:

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Die Funktion `socket` und die Konstanten `AF_INET` und `SOCK_STREAM` aus dem Modul werden mit vorangestelltem `socket` verwendet.

Ein Blick mit `dir()` in den Namensraum des Interpreters zeigt nur das Modul `socket` im Namensraum.

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'socket']
```

Punkt 2a ist eine Möglichkeit wenn wenige Funktionen eines Moduls benötigt werden und keine Probleme mit gleich benannten Funktionen und Konstanten erwartet werden. Dadurch kann einige Tipparbeit vermieden werden.

```
>>> from socket import socket
>>> s = socket(AF_INET, SOCK_STREAM)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'AF_INET' is not defined
```

Leider muss man in diesem Fall an alle benötigten Funktionen und Konstanten denken bzw. später ergänzen:

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> dir()
['AF_INET', 'SOCK_STREAM', '__annotations__', '__builtins__',
'__doc__', '__loader__', '__name__', '__package__', '__spec__',
's', 'socket']
```

Jetzt zeigen sich zusätzlich zur Variablen `s` schon drei Elemente im Namensraum.

Die letzte Form

```
>>> from socket import *
>>> dir()
['AF_ALG', 'AF_APPLETALK', 'AF_ASH', 'AF_ATMPVC', 'AF_ATMSVC',
'AF_AX25', 'AF_BLUETOOTH', 'AF_BRIDGE', 'AF_CAN',...
```

ist die allgemeinste und birgt die Gefahr bereits vorhandene Namen durch den Import zu verdecken. Das kann zu unerwartetem Verhalten des Programms führen.

Jedes Python-Programm kann als Modul geladen werden.

4.2 Fehlerbehandlung

Python behandelt Fehler durch das Fangen von Ausnahmen (Exceptions). Codeblöcke werden in die Anweisungen `try` und `except` eingeschlossen. In diesem Block auftretende Fehler können im `except`-Zweig behandelt werden.

```
>>> 42 / '7'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Python liefert bei einem Laufzeitfehler einen sogenannten Traceback und beendet das Programm. Neben der Position des Fehlers (Modul und Zeilennummer) enthält die Ausgabe den Fehlertyp und eine Erklärung.

Um den Programmabbruch zu verhindern kann `try .. except` genutzt werden.

```
1 try:
2     42 / '7'
3 except TypeError as e:
4     print( 'Fehler: {}'.format(e) )
```

Listing 4.1. Fehler abfangen

Das Programm läuft mit der eigenen Meldung als Ausgabe:

```
>>> try:
...     42 / '7'
... except TypeError as e:
...     print( 'Fehler: {}'.format(e) )
...
Fehler: unsupported operand type(s) for /: 'int' and 'str'
```

Mehrere Fehlerbehandlungen können in einer Liste hintereinander angegeben werden (Listing 4.2 Zeile 3).

```
1 try:
2     42 / '7'
3 except (TypeError, NameError) as e:
4     print( 'Fehler: {} ({})'.format(e, type(e)) )
```

Listing 4.2. Verschiedene Fehler auf einmal

Fehler sollten konkret um „gefährliche“ Aktionen abgefangen werden. Zur Sicherheit sollte auf oberster Programmebene ein Handler für alle Laufzeitfehler platziert werden (Listing 4.3).

```
1 try:
2     # beliebiger code...
3     try:
4         # mehr code...
5     except TypeError:
6         # Behandlung TypeError
7     # noch mehr code...
8 except Exception as e:
9     print( 'Fehler: {} ({})'.format(e, type(e)) )
```

Listing 4.3. Beliebige Fehler abfangen

Die Klasse `Exception` ist die Basisklasse für alle Laufzeitfehler.

4.3 Bytes und Zeichenketten

Python unterscheidet seit Version 3 Zeichenketten und Bytefolgen. Zeichenketten werden im Programm verwendet und dienen zur Ausgabe an den Anwender. Bytefolgen kommen aus Datei-Objekten, also auch Sockets und sind die rohe Form der Daten.

Eine Zeichenkette ist eine interpretierte Bytefolge. Dafür wurde die Bytefolge unter Angabe eines Encodings decodiert, z. B.:

```
bdata = s.recv(4096)
data = bdata.decode('utf-8')
```

Natürlich können auch Bytefolgen bearbeitet und ausgegeben werden. Dies hat aber einige Einschränkungen gegenüber Zeichenketten.

Der Versand einer Zeichenkette über einen Socket oder das Speichern in eine Binärdatei[1] erfordern eine Codierung der Daten mit der Methode `encode()`, z. B.:

```
s.send('GET / HTTP/1.0\r\n'.encode('utf-8'))
```

4.4 Python-Module für Netzwerkprogrammierung

Eine kurze Liste von Python-Modulen.

Tab. 4.1. Module für Kommunikation

Modul	Beschreibung
`http`	HTTP-Module.
`http.client`	HTTP-Client.
`http.server`	HTTP-Server.
`ftplib`	FTP.
`imaplib`	IMAP4.
`ipaddress`	Bearbeitung von IP-Adressen.
`nntplib`	NNTP.
`poplib`	POP3.
`smtplib`	SMTP Client.
`smtpd`	SMTP Server.
`select`	I/O-Multiplexing mit `select`.
`selectors`	Python-Interpretation von I/O-Multiplexing.
`socket`	Socket-Schnittstelle.
`socketserver`	Basisklasse für Netzwerkserver.
`ssl`	TLS/SSL-Funktionen für Sockets.
`telnetlib`	Telnet Client.
`urllib`	Bearbeitung von URL.
`urllib.request`	Abfragen einer URL.
`urllib.response`	Klassen für das Ergebnis eines Requests.
`urllib.error`	Fehlerklassen beim Zugriff auf eine URL.
`urllib.parse`	Zerlegen einer URL.
`xmlrpc`	XML RPC.
`xmlrpc.client`	XML RPC Client
`xmlrpc.server`	XML RPC Server

Python Referenz: `https://docs.python.org/`

[1] Python bietet eine Ausnahme zum Schreiben und Lesen von Textdateien. Dies ist die Standardeinstellung bei der Funktion `open()` und die Erstellung von Binärdateien muß hier explizit angegeben werden.