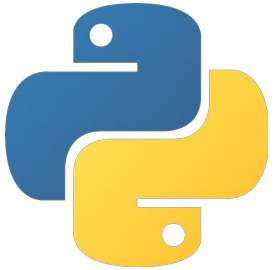
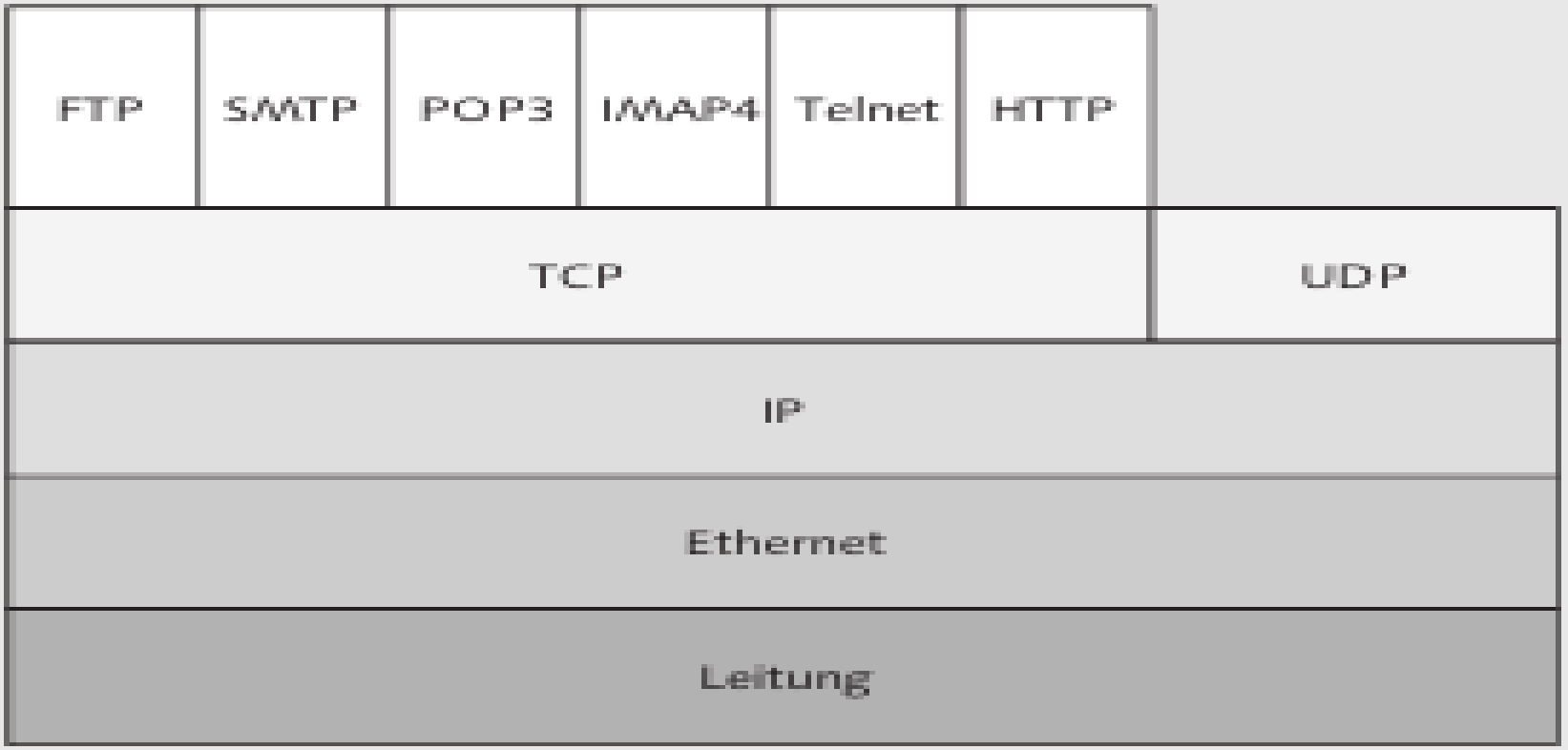
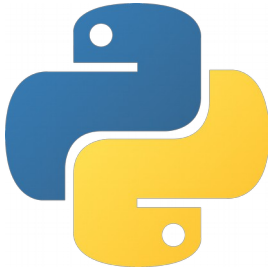
Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Für Anwendungsprogrammierer eigentlich interessante Teil fängt erst oberhalb des IP-Protokolls an, nämlich bei den Transportprotokollen TCP und UDP



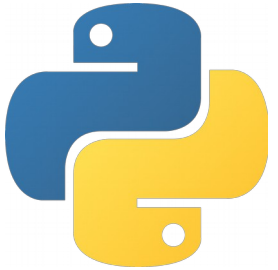
Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Die Protokolle, die auf TCP aufbauen, sind am weitesten abstrahiert und deshalb für uns ebenfalls interessant

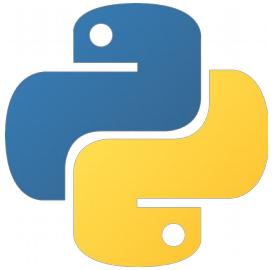
Protokoll	Beschreibung	Modul
UDP	grundlegendes verbindungsloses Netzwerkprotokoll	*socket*
TCP	grundlegendes verbindungsorientier-tes Netzwerkprotokoll	*socket*
HTTP	Übertragen von Textdateien, beispiels-weise Webseiten	*urllib*
FTP	Dateiübertragung	*ftplib*
SMTP	Versenden von E-Mails	*smtplib*
POP3	Abholen von E-Mails	*poplib*
IMAP4	Abholen und Verwalten von E-Mails	*imaplib*
Telnet	Terminalemulation	*telnetlib*

Python Netzwerkkommunikation



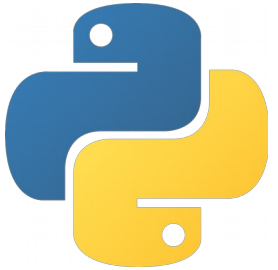
- Netzwerkkommunikation
  - Es gibt auch abstrakte Protokolle, die auf UDP aufbauen, beispielsweise NFS (Network File System)
  - Wir werden aber ausschließlich auf TCP basierende Protokolle behandeln, da diese die am häufigsten verwendeten sind
  - Nachdem wir uns mit der Socket API beschäftigt haben, folgen einige spezielle Module, die beispielsweise mit bestimmten Protokollen wie HTTP oder FTP umgehen können

Python Netzwerkkommunikation



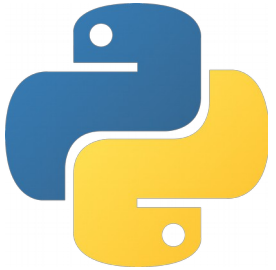
- Netzwerkkommunikation
  - Socket API
    - Das Modul socket der Standardbibliothek bietet grundlegende Funktionalität zur Netzwerkkommunikation
    - Es bildet dabei die standardisierte Socket API ab, die so oder in ähnlicher Form auch für viele andere Programmiersprachen implementiert ist
    - Jeder Rechner besitzt in einem Netzwerk, auch dem Internet, eine eindeutige sogenannte IP-Adresse, über die er angesprochen werden kann

Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API
    - Eine IP-Adresse ist ein String der folgenden Struktur "192.168.1.23"
    - Dabei repräsentiert jeder der vier Zahlenwerte ein Byte und kann somit zwischen 0 und 255 liegen
    - In diesem Fall handelt es sich um eine IP-Adresse eines lokalen Netzwerks, was an der Anfangssequenz 192.168 zu erkennen ist

Python Netzwerkkommunikation



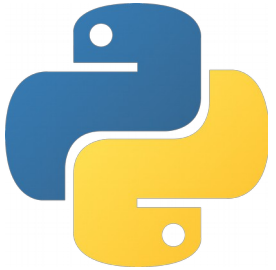
- Netzwerkkommunikation
  - Socket API

    - Damit ist es jedoch noch nicht getan, denn auf einem einzelnen Rechner können mehrere Programme laufen, die gleichzeitig Daten über die Netzwerkschnittstelle senden und empfangen möchten

    - Aus diesem Grund wird eine Netzwerkverbindung zusätzlich an einen sogenannten Port gebunden

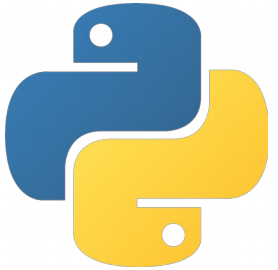
    - Der Port ermöglicht es, ein bestimmtes Programm anzusprechen, das auf einem Rechner mit einer bestimmten IP-Adresse läuft

Python Netzwerkkommunikation

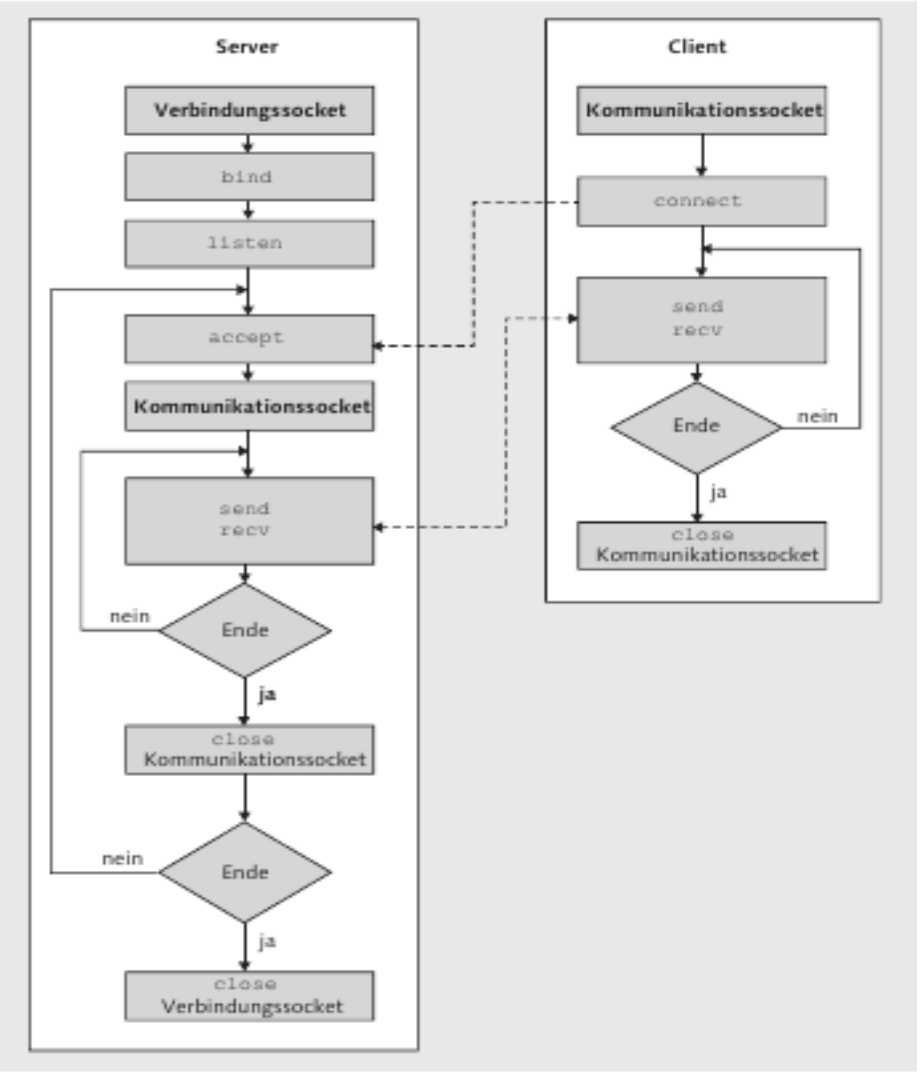


- Netzwerkkommunikation
  - Socket API
    - Bei einem Port handelt es sich um eine 16-Bit-Zahl – grundsätzlich sind also 65.535 verschiedene Ports verfügbar
    - Allerdings sind viele dieser Ports für Protokolle und Anwendungen registriert und sollten nicht verwendet werden
    - Beispielsweise sind für HTTP- und FTP-Server die Ports 80 bzw. 21 registriert
    - Grundsätzlich können Sie Ports ab 1024 verwenden, da hört die Systemverwaltung auf, ab 49152 bedenkenlos verwenden

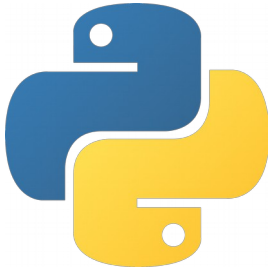
Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API



Python Netzwerkkommunikation



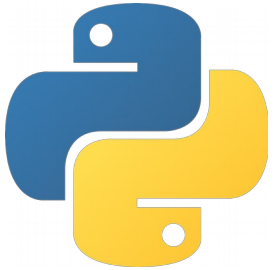
- Netzwerkkommunikation
  - Socket API
  - UDP

    - Das Netzwerkprotokoll UDP (User Datagram Protocol) wurde 1977 als Alternative zu TCP für die Übertragung menschlicher Sprache entwickelt

    - Charakteristisch ist, dass UDP verbindungslos und nicht-zuverlässig ist

    - Diese beiden Begriffe gehen miteinander einher und bedeuten zum einen, dass keine explizite Verbindung zwischen den Kommunikationspartnern aufgebaut wird, und zum anderen, dass UDP weder garantiert, dass gesendete Pakete in der Reihenfolge ankommen, in der sie gesendet wurden, noch dass sie überhaupt ankommen

Python Netzwerkkommunikation



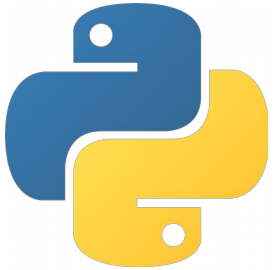
- Netzwerkkommunikation
  - Socket API
  - UDP

    - Aufgrund dieser Einschränkungen können mit UDP jedoch vergleichsweise schnelle Übertragungen stattfinden, da beispielsweise keine Pakete neu angefordert oder gepuffert werden müssen

    - Damit eignet sich UDP insbesondere für Multimedia-Anwendungen wie VoIP, Audio- oder Videostreaming, bei denen es auf eine schnelle Übertragung der Daten ankommt und kleinere Übertragungsfehler toleriert werden können

    - Auf den folgenden Seiten ein Client-Server Beispiel mit UDP

Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API
  - UDP

    - Client

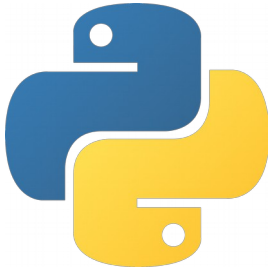
```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

ip = input("IP-Adresse: ")
nachricht = input("Nachricht: ")

s.sendto(nachricht.encode(), (ip, 50000))
s.close()
```

Python Netzwerkkommunikation

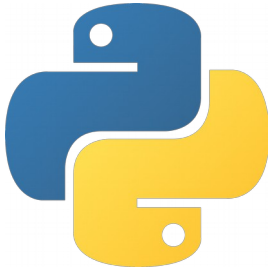


- Netzwerkkommunikation
  - Socket API
  - UDP
    - Server

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
try:
   s.bind(("", 50000))
   while True:
      daten, addr = s.recvfrom(1024)
      print("[{}] {}".format(addr[0], daten.decode()))
finally:
   s.close()
```

Python Netzwerkkommunikation



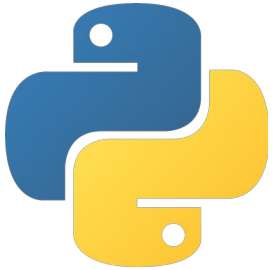
- Netzwerkkommunikation
  - Socket API
  - TCP

    - TCP (Transmission Control Protocol) ist kein Konkurrenzprodukt zu UDP, sondern füllt mit seinen Möglichkeiten die Lücken auf, die UDP offen lässt

    - So ist TCP vor allem Verbindungsorientiert und zuverlässig

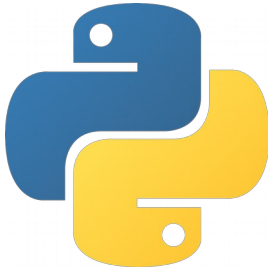
    - Verbindungsorientiert bedeutet, dass nicht, wie bei UDP, einfach Datenpakete an bestimmte IP-Adressen geschickt werden, sondern dass zuvor eine Verbindung aufgebaut wird und auf Basis dieser Verbindung weitere Operationen durchgeführt werden

Python Netzwerkkommunikation



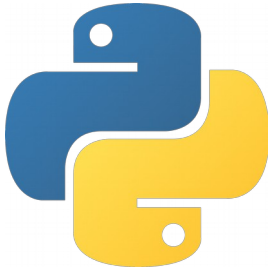
- Netzwerkkommunikation
  - Socket API
  - TCP
    - Zuverlässig bedeutet, dass es mit TCP nicht, wie bei UDP, vorkommen kann, dass Pakete verloren gehen, fehlerhaft oder in falscher Reihenfolge ankommen
    - Solche Vorkommnisse korrigiert das TCP-Protokoll intern, indem es beispielsweise unvollständige oder fehlerhafte Pakete neu anfordert
    - Aus diesem Grund ist TCP zumeist die erste Wahl, wenn es um eine Netzwerkschnittstelle geht

Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API
  - TCP
    - Bedenken Sie aber unbedingt, dass jedes Paket, das neu angefordert werden muss, Zeit kostet und die Latenz der Verbindung somit steigen kann
    - Außerdem sind fehlerhafte Übertragungen in einem LAN äußerst selten, weswegen Sie gerade dort die Performance von UDP und die Verbindungsqualität von TCP gegeneinander abwägen sollten
    - Auf den folgenden Seiten ein Client-Server Beispiel mit TCP

Python Netzwerkkommunikation

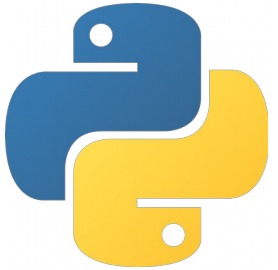


- Netzwerkkommunikation
  - Socket API
  - TCP

    - Client

```
import socket
ip = input("IP-Adresse: ")
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, 50000))
try:
   while True:
      nachricht = input("Nachricht: ")
      s.send(nachricht.encode())
      antwort = s.recv(1024)
      print("[{}] {}".format(ip, antwort.decode()))
finally:
   s.close()
```

Python Netzwerkkommunikation



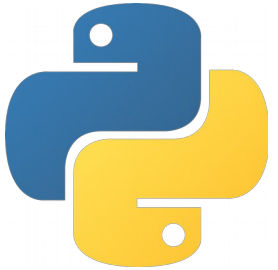
- Netzwerkkommunikation
  - Socket API
  - TCP

    - Server

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("", 50000))
s.listen(1)
try:
whileTrue:
   komm, addr = s.accept()

   ...
```

Python Netzwerkkommunikation



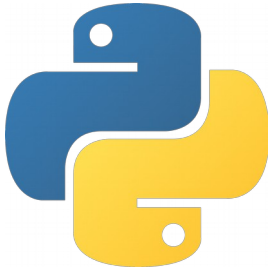
- Netzwerkkommunikation
  - Socket API
  - TCP

    - Server

```
…

while True:
    data = komm.recv(1024)
    if not data:
        komm.close()
        break
    print("[{}] {}".format(addr[0], data.decode()))
    nachricht = input("Antwort: ")
    komm.send(nachricht.encode())
finally:
  s.close()
```

Python Netzwerkkommunikation



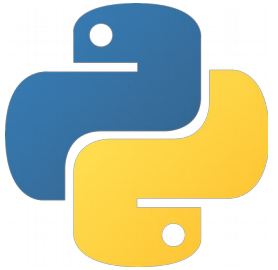
- Netzwerkkommunikation
  - Socket API
  - TCP

    - Server

```
…

while True:
    data = komm.recv(1024)
    if not data:
        komm.close()
        break
    print("[{}] {}".format(addr[0], data.decode()))
    nachricht = input("Antwort: ")
    komm.send(nachricht.encode())
finally:
  s.close()
```

Python Netzwerkkommunikation



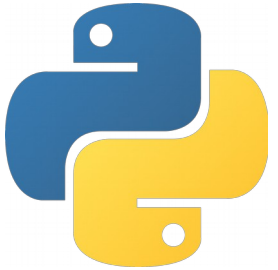
- Netzwerkkommunikation
  - Socket API
  - Blockierende und nicht-blockierende Sockets

    - Wenn ein Socket erstellt wird, befindet er sich standardmäßig im sogenannten blockierenden Modus (engl. blocking mode)

    - Das bedeutet, dass alle Methodenaufrufe warten, bis die von ihnen angestoßene Operation durchgeführt wurde

    - So blockiert ein Aufruf der Methode recv eines Sockets so lange das komplette Programm, bis tatsächlich Daten eingegangen sind und aus dem internen Puffer des Sockets gelesen werden können

Python Netzwerkkommunikation



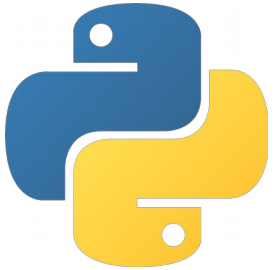
- Netzwerkkommunikation
  - Socket API
  - Blockierende und nicht-blockierende Sockets

    - Ein Socket lässt sich durch Aufruf seiner Methode setblocking in den nicht-blockierenden Zustand versetzen s.setblocking(False)

    - Dies wirkt sich folgendermaßen auf diverse Socket-Operationen aus

      - Die Methoden recv und recvfrom des Socket-Objekts geben nur noch ankommende Daten zurück, wenn sich diese bereits im internen Puffer des Sockets befinden
      - Sobald die Methode auf weitere Daten zu warten hätte, wirft sie eine socket.error-Exception und gibt damit den Kontrollfluss wieder an das Programm ab

Python Netzwerkkommunikation

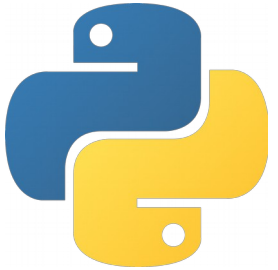


- Netzwerkkommunikation
  - Socket API
  - Blockierende und nicht-blockierende Sockets

    - Dies wirkt sich folgendermaßen auf diverse Socket-Operationen aus

      - Die Methoden send und sendto versenden die angegebenen Daten nur, wenn sie direkt in den Ausgangspuffer des Sockets geschrieben werden können
      - Gelegentlich kommt es vor, dass dieser Puffer voll ist und send bzw. sendto zu warten hätten, bis der Puffer weitere Daten aufnehmen kann
      - In einem solchen Fall wird im nicht-blockierenden Modus eine socket.error-Exception geworfen und der Kontrollfluss damit an das Programm zurückgegeben

Python Netzwerkkommunikation

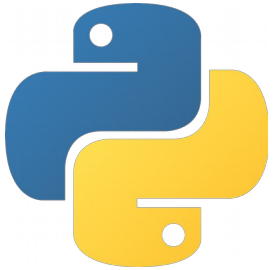


- Netzwerkkommunikation
  - Socket API
  - Blockierende und nicht-blockierende Sockets

    - Dies wirkt sich folgendermaßen auf diverse Socket-Operationen aus

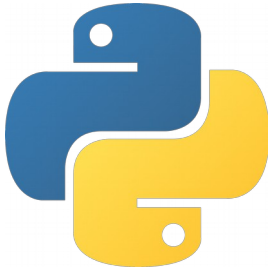
      - Die Methode connect sendet eine Verbindungsanfrage an den Zielsocket und wartet nicht, bis diese Verbindung zustande kommt
      - Durch mehrmaligen Aufruf von connect lässt sich feststellen, ob die Operation immer noch durchgeführt wird
      - Wenn connect aufgerufen wird und die Verbindungsanfrage noch läuft, wird eine socket.error-Exception mit der Fehlermeldung »Operation now in progress« geworfen

Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API
  - Verwendung des Moduls
    - create_connection(address[, timeout[, source_address]])
      - Diese Funktion verbindet über TCP zu der über das Adressobjekt address identifizierten Gegenstelle und gibt das zum Verbindungsaufbau verwendete Socket-Objekt zurück
      - Für den Parameter timeout kann ein Timeout-Wert übergeben werden, der beim Verbindungsaufbau berücksichtigt wird
      - Wenn dem Parameter source_address ein Adressobjekt übergeben wird, wird das Socketobjekt vor dem Aufbau an diese Adresse gebunden

Python Netzwerkkommunikation

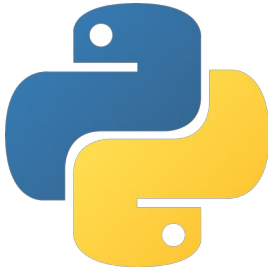


- Netzwerkkommunikation
  - Socket API
  - Verwendung des Moduls
    - create_connection(address[, timeout[, source_address]])

      import socket
      s = socket.create_connection((ip1,port1), timeout, (ip2, port2))

    - Äquivalent
      import socket
      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
      s.settimeout(timeout)
      s.bind((ip2, port2))
      s.connect((ip1, port1))

Python Netzwerkkommunikation



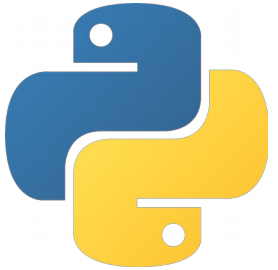
- Netzwerkkommunikation
  - Socket API
  - Verwendung des Moduls

    - create_connection(address[, timeout[, source_address]])

    - Alternative mit 'with'

```
with socket.create_connection(("ip", port)) as s:
    s.send(b"Hallo Welt")
```

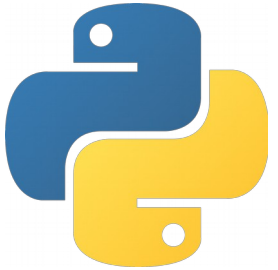
Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API
  - Verwendung des Moduls
    - gethostbyname_ex(hostname)
      - Die Funktion gethostbyname_ex gibt für einen gegebenen Hostnamen ein Tupel zurück, das den primären Hostnamen, eine Liste der alternativen Hostnamen und eine Liste der IPv4-Adressen, unter denen der Server erreichbar ist

        >>> socket.gethostbyname_ex("www.google.de")
        ('www.l.google.com',
        ['www.google.de', 'www.google.com'],
        ['74.125.39.99', '74.125.39.104', '74.125.39.105',
        '74.125.39.103', '74.125.39.106', '74.125.39.147'])

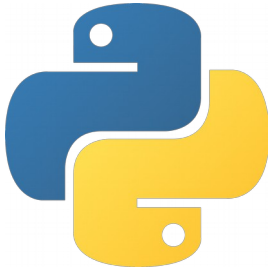
Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API
  - Verwendung des Moduls
    - getservbyname(service[, protocol])
      - Diese Funktion gibt den Port für den Service service mit dem Netzwerkprotokoll protocol zurück
      - Bekannte Services sind beispielsweise "http" oder "ftp" mit den Portnummern 80 bzw. 21
      - Der Parameter protocol sollte entweder "tcp" oder "udp" sein

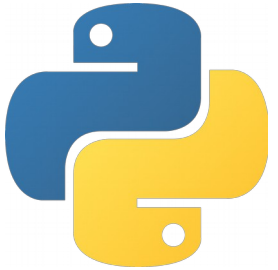
        >>> socket.getservbyname("http", "tcp")
        80

Python Netzwerkkommunikation



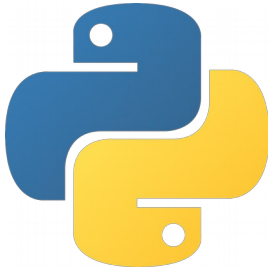
- Netzwerkkommunikation
  - Socket API
  - Verwendung des Moduls
    - socket([family[, type[, proto]]])
      - Diese Funktion erzeugt einen neuen Socket
      - Der erste Parameter family kennzeichnet dabei die Adressfamilie und sollte entweder socket.AF_INET für den IPv4-Namensraum oder socket.AF_INET6 für den IPv6-Namensraum sein
      - Der zweite Parameter type kennzeichnet das zu verwendende Netzwerkprotokoll und sollte entweder socket.SOCK_STREAM für TCP oder socket.SOCK_DGRAM für UDP sein

Python Netzwerkkommunikation



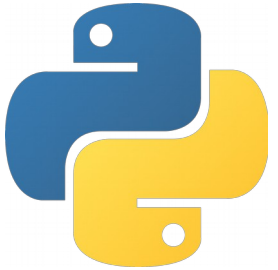
- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - accept()
      - Diese Methode wartet auf eine eingehende Verbindungsanfrage und akzeptiert diese
      - Die Socket -Instanz muss zuvor durch Aufruf der Methode bind an eine bestimmte Adresse und einen Port gebunden worden sein und Verbindungsanfragen erwarten
      - Letzteres geschieht durch Aufruf der Methode listen

Python Netzwerkkommunikation



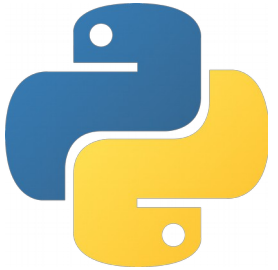
- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - accept()
      - Die Methode accept gibt ein Tupel zurück, dessen erstes Element eine neue Socket-Instanz, auch Connection-Objekt genannt, ist, über die die Kommunikation mit dem Verbindungspartner erfolgen kann
      - Das zweite Element des Tupels ist ein weiteres Tupel, das IP-Adresse und Port des verbundenen Sockets enthält

Python Netzwerkkommunikation



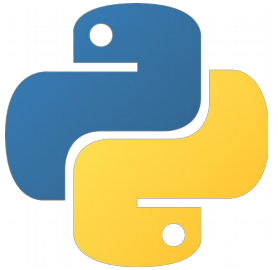
- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - bind(address)
      - Diese Methode bindet den Socket an die Adresse address
      - Der Parameter address muss ein Tupel der Form sein, wie es accept zurückgibt
      - Nachdem ein Socket an eine bestimmte Adresse gebunden wurde, kann er, im Falle von TCP, in den passiven Modus geschaltet werden und auf Verbindungsanfragen warten oder, im Falle von UDP, direkt Datenpakete empfangen

Python Netzwerkkommunikation



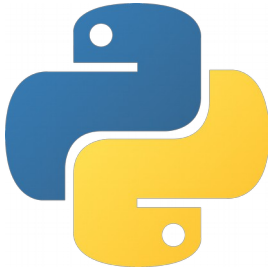
- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - connect(address)
      - Diese Methode verbindet zu einem Server mit der Adresse address
      - Beachten Sie, dass dort ein Socket existieren muss, der auf dem gleichen Port auf Verbindungsanfragen wartet, damit die Verbindung zustande kommen kann.
      - Der Parameter address muss im Falle des IPv4-Protokolls ein Tupel sein, das aus der IP-Adresse und der Portnummer besteht

Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - connect_ex(address)
      - Diese Methode unterscheidet sich von connect nur darin, dass im nicht-blockierenden Modus keine Exception geworfen wird, wenn die Verbindung nicht sofort zustande kommt
      - Der Verbindungsstatus wird über einen ganzzahligen Rückgabewert angezeigt. Ein Rückgabewert von 0 bedeutet, dass der Verbindungsversuch erfolgreich durchgeführt wurde
      - Beachten Sie, dass bei echten Fehlern, die beim Verbindungsversuch auftreten, weiterhin Exceptions geworfen werden, beispielsweise wenn der Zielsocket nicht erreicht werden konnte

Python Netzwerkkommunikation

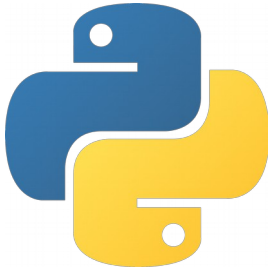


- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse

    - listen(backlog)

      - Diese Methode versetzt einen Serversocket in den sogenannten Listen-Modus, das heißt, der Socket achtet auf Sockets, die sich mit ihm verbinden wollen. Nachdem diese Methode aufgerufen wurde, können eingehende Verbindungswünsche mit accept akzeptiert werden
      - Der Parameter backlog legt die maximale Anzahl an gepufferten Verbindungsanfragen fest und sollte mindestens 1 sein
      - Den größtmöglichen Wert für backlog legt das Betriebssystem fest, meistens liegt er bei 5

Python Netzwerkkommunikation

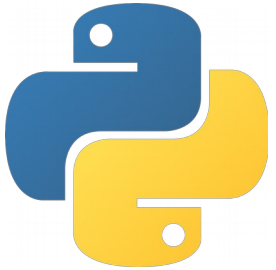


- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse

    - recv(bufsize[, flags])

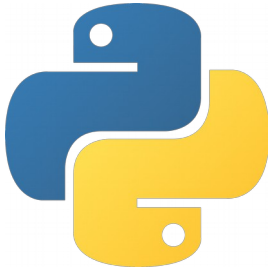
      - Diese Methode liest beim Socket eingegangene Daten
      - Durch den Parameter bufsize wird die maximale Anzahl von zu lesenden Bytes festgelegt
      - Die gelesenen Daten werden in Form eines Strings zurückgegeben
      - Über den optionalen Parameter flags kann das Standardverhalten von recv geändert werden
      - Diese Veränderungen werden allerdings nur in seltenen Fällen benötigt, weswegen wir sie hier nicht besprechen möchten, gilt auch für folgende Methoden

Python Netzwerkkommunikation



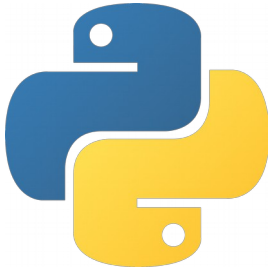
- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - recvfrom(bufsize[, flags])
      - Diese Methode unterscheidet sich von recv in Bezug auf den Rückgabewert
      - Dieser ist bei recvfrom ein Tupel, das als erstes Element die gelesenen Daten als String und als zweites Element das Adressobjekt des Verbindungspartners enthält

Python Netzwerkkommunikation



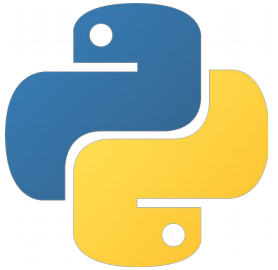
- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - send(string[, flags])
      - Diese Methode sendet den String string zum verbundenen Socket
      - Die Anzahl der gesendeten Bytes wird zurückgegeben
      - Beachten Sie, dass unter Umständen die Daten nicht vollständig gesendet wurden
      - In einem solchen Fall ist die Anwendung dafür verantwortlich, die verbleibenden Daten erneut zu senden

Python Netzwerkkommunikation



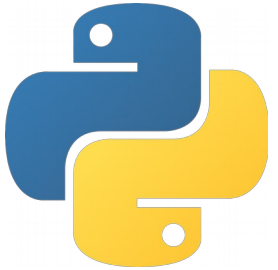
- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - sendall(string[, flags])
      - Diese Methode unterscheidet sich von send darin, dass sendall so lange versucht, die Daten zu senden, bis entweder der vollständige Datensatz string versendet wurde oder ein Fehler aufgetreten ist
      - Im Fehlerfall wird eine entsprechende Exception geworfen

Python Netzwerkkommunikation



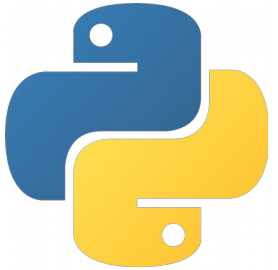
- Netzwerkkommunikation
  - Socket API
  - Die Socket-Klasse
    - settimeout(value), gettimeout()
      - Diese Methode setzt einen Timeout-Wert für diesen Socket
      - Dieser Wert bestimmt im blockierenden Modus, wie lange auf das Eintreffen bzw. Versenden von Daten gewartet werden soll
      - Dabei können Sie für value die Anzahl an Sekunden in Form einer Gleitkommazahl oder None übergeben
      - Über die Methode gettimeout kann der Timeout-Wert ausgelesen werden

Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Socket API
  - Netzwerk-Byte-Order
    - Wird nicht weiter erläutert, bei Interesse nachlesen
  - Multiplexende Server – Das Modul select
    - Für die Verarbeitung mehrerer Client's
    - Bei Interesse bitte nachlesen
  - Multiclient Server – Das Modul socketserver
    - Ist in der Lage ist, mehrere Clients zu bedienen
    - Bei Interesse bitte nachlesen

Python Netzwerkkommunikation



- Netzwerkkommunikation
  - Weitere Thematiken, deren Umgang nicht erläutert wird
  - Umgang mit FTP – Das Modul ftplib
    - File Transfer Protocol
    - Dateiübertragungen in TCP/IP-Netzwerken
  - Umgang mit Telnet – Das Modul telnet
    - Auch weil es ein altes Protokoll ist, was nur noch selten verwendet wird, auch selten aktiviert ist
  - XML-RPC – Extensible Markup Language Remote Procedure Call
    - Dient dem entfernten Funktions- und Methodenaufruf über eine Netzwerkschnittstelle
    - Modul xmlrpc.server