<http://andreymal.org/socket3/>

«Со́кеты (англ. socket — разъём) — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.» © Википедия  
  
Суть работы: на одном компьютере программа открывает сокет, слушает какой-то порт (в случае с TCP и UDP), другая программа на другом (или том же) компьютере, указав IP и этот самый порт, подключается к слушающей порт программе, и дальше они обмениваются какими надо данными, после чего закрывают соединение.  
  
Для работы с сокетами нам нужно импортировать соответствующий модуль.

import socket

Теперь нужно создать сам сокет.

sock = socket.socket()

Теперь у нас есть сокет в переменной sock, и мы можем работать с ним дальше.

**Сервер TCP**

Суть TCP-соединения: одна программа устанавливает соединение с другой, и они обмениваются данными, причём их потери не происходит. После завершения работы соединение должно быть закрыто.  
  
Данные в TCP — это поток байтов. Разделять его на отдельные сообщения придётся самой программе.  
  
Сокет для TCP-соединения создаётся как обычно. Для создания сервера нужно связать сокет с одним или всеми из имеющихся у компьютера хостов (IP-адресов) и каким-либо свободным портом. Если не указать хост или указать "0.0.0.0", сокет будет прослушивать все хосты. Если указать "127.0.0.1", то подключиться можно будет только с этого же компьютера.  
  
Для привязки используется функция bind сокета, которая принимает массив, содержащий два элемента: хост и порт.

sock.bind( ("", 14900) )

Теперь можно заняться прослушкой, это можно сделать с помощью функции listen. Она принимает в качестве аргумента максимальное число соединений.

sock.listen(10)

Теперь принимаем соединения с помощью функции accept. Она ждёт появление входящего соединения и возвращает связанный с ним сокет и адрес подключившегося. Адрес — массив, состоящий из IP-адреса и порта.

conn, addr = sock.accept()

В объекте conn теперь у нас сокет, через который мы можем обмениваться данными с клиентом, в addr[0] — IP-адрес подключившегося клиента. Чтобы получить следующего клиента, нужно вызвать функцию accept ещё раз, при этом необязательно закрывать соединение с предыдущим клиентом: можно держать столько подключенных клиентов, сколько было указано в listen.  
  
Для чтения данных используется функция recv, которой первым параметром нужно передать количество получаемых байт данных. Если столько байт, сколько указано, не пришло, а какие-то данные уже появились, она всё равно возвращает всё, что имеется, поэтому надо контролировать размер полученных данных.

data = conn.recv(16384)

Тип возвращаемых данных — bytes. У этого типа есть почти все методы, что и у строк, но для того, чтобы использовать из него текстовые данные с другими строками (складывать, например, или искать строку в данных, или печатать), придётся декодировать данные (или их часть, если вы обработали байты и выделили строку) и использовать уже полученную строку. *(Здесь и далее используется кодировка utf-8, если вы вдруг по какой-то глупости используете другую кодировку — указывайте свою.)*

udata = data.decode("utf-8")  
print("Data: " + udata)

Если вы попытаетесь использовать байты вместо строк, вы получите ошибку:

>>> print("Data: "+data)  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
TypeError: Can't convert 'bytes' object to str implicitly

Для отправки данных в сокет используется функция send. Принимает она тоже bytes, поэтому для отправки строки вам придётся её закодировать.

conn.send(b"Hello!\n")  
conn.send(b"Your data: " + udata.encode("utf-8"))

Вот так с помощью функций recv и send и осуществляется весь обмен данными в TCP-соединении.  
  
После всего и клиенту, и серверу необходимо закрыть сокет с помощью функции close.

conn.close()

Теперь этот сокет использовать нельзя.  
  
В случае, если другая сторона сторона закроет сокет, функция recv вернёт пустой объект bytes.  
  
Если данных приходится ждать слишком долго, можно перед использованием функции recv задать (однократно) таймаут с помощью функции settimeout.

conn, addr = sock.accept() *# старая строка получения сокета*  
conn.settimeout(60) *# установка таймаута*  
data = conn.recv(16384) *# получение данных, про это рассказано выше*

Теперь, если за 60 секунд не придут никакие данные, функция recv вернёт пустой объект bytes, как и при закрытом соединении.

if not data:  
 print("No data")  
 conn.close()

Для примера и закрепления всего прочитанного привожу простенький HTTP-сервер, возвращающий текущие дату и время.  
  
Запустите его и наберите в браузере адрес **http://localhost:8080/time.html**

Показать код

**TCP-клиент**

HTTP-сервер с браузером это, конечно, хорошо, но вы же тут все хотите онлайн-игры делать ;D Поэтому придётся научиться делать программу клиентом. Сокет создаётся точно так же:

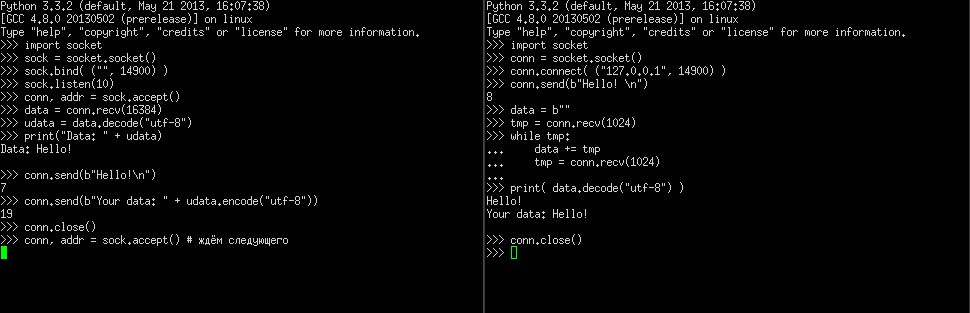
conn = socket.socket()

А вот дальше появляется отличие. Вместо прослушивания порта мы подключаемся к другому хосту с помощью функции connect, которая принимает этот самый хост (IP-адрес или можно сразу обычный адрес буквами написать) и порт.

conn.connect( ("127.0.0.1", 14900) )

А дальше всё как обычно: для установки таймаута используется settimeout, для обмена данными send и recv, для закрытия close.

conn.send(b"Hello! \n")  
data = b""  
tmp = conn.recv(1024)  
while tmp:  
 data += tmp  
 tmp = conn.recv(1024)  
print( data.decode("utf-8") )  
conn.close()

  
  
Те, кто собрался делать крутые онлайн-игры в Blender Game Engine, столкнутся с тем, что функции *accept* и *recv* ждут соединения и данных, и в результате игра виснет на время ожидания, что плохо. Блокировку с *recv* поможет снять функция *setblocking(0)*. Тогда в случае отсутствия данных функция не будет ждать, а выкинет исключение socket.error, которое можно будет поймать в блоке try-except, после чего спокойно завершить скрипт, не вешая игру.

*# при открытии соединения:*  
conn = socket.socket()  
conn.connect( ("yandex.ru", 80) )  
conn.setblocking(0)  
  
*# в скрипте, читающем данные:*  
try: data = conn.recv(1024)  
except socket.error: *# данных нет*  
 pass *# тут ставим код выхода*  
else: *# данные есть*  
 print(data)  
 *# если в блоке except вы выходите,*  
 *# ставить else и отступ не нужно*  
*# скрипт, читающий данные, запускаем на каждом кадре*

Аналогично с функцией *accept*.

*# при создании сервера:*  
conn = socket.socket()  
conn.bind( ("", 8989) )  
conn.listen(100)  
conn.setblocking(0)  
  
*# в скрипте, который получает клиентов:*  
try: client, addr = conn.accept()  
except socket.error: *# данных нет*  
 pass *# тут ставим код выхода*  
else: *# данные есть*  
 client.setblocking(0) *# снимаем блокировку и тут тоже*  
 parse(client, addr)  
 *# если в блоке except вы выходите,*  
 *# ставить else и отступ не нужно*  
*# скрипт, получающий клиентов, запускаем на каждом кадре*

Но, конечно, для сервера обрабатывать, например, под сотню клиентов, у каждого просить recv и прочее это издевательство, и тут лучше использовать потоки или какие-нибудь гринлеты с gevent, а полученные данные хранить в очереди (Queue), которая будет читаться основным потоком, или быть гуру и заюзать Twisted, и всё такое прочее мудрёное, но брать на себя ответственность объяснять такую страшную и тяжелую вещь я не хочу, так как сам в этом ещё не силён; лучше вот почитайте [хороший цикл статей про многопоточность в питоне](http://asvetlov.blogspot.ru/2010/11/1.html).  
  
Для обработки текстовых данных ознакомьтесь с [функцями для работы со строками](http://andreymal.org/python_str/), с помощью них уже можно будет составить простой текстовый протокол и, закодироав координаты кубика с помощью str, соединив через join, отправив закодированное по сокету и разбив присланное на другом компьютере, двигать кубик по сети. (Но лучше, конечно, какой-нибудь pickle использовать, а для серьёзной игры свой бинарный протокол, чтобы трафик впустую не тратился.)