

# Table of Contents

|                    |         |
|--------------------|---------|
| Introduction       | 1.1     |
| ソケットプログラム          | 1.2     |
| socket             | 1.3     |
| サーバ側               | 1.3.1   |
| IPアドレスとポートの設定      | 1.3.2   |
| ソケット作成・bind・listen | 1.3.3   |
| どんな通信を行うか          | 1.3.3.1 |
| どのIPアドレスとポートを使うか   | 1.3.3.2 |
| どれぐらい通信するか         | 1.3.3.3 |
| ソケット作成まとめ          | 1.3.3.4 |
| 処理待ち               | 1.3.4   |
| サーバ処理              | 1.3.5   |
| データの受信             | 1.3.6   |
| データの送信             | 1.3.7   |
| 接続を閉じる             | 1.3.8   |
| サーバまとめ             | 1.3.9   |
| クライアント             | 1.3.10  |
| IPアドレスとポートの設定      | 1.3.11  |
| ソケットを作成            | 1.3.12  |
| サーバに接続             | 1.3.13  |
| 送信・受信              | 1.3.14  |
| 実践                 | 1.3.15  |
| socketserver       | 1.4     |
| サーバ定義              | 1.4.1   |
| リクエストハンドル          | 1.4.2   |
| socketserverまとめ    | 1.4.3   |
| socketserverを動かそう  | 1.4.4   |
| 実践2                | 1.4.5   |

# ソケットプログラム

ソケットプログラムとは、サーバとクライアントの通信をプログラムしたものである。通常、ソケットとは、TCP/IPアプリケーションを作成するための抽象化されたインターフェースとされている。簡単に言うとソケットは、サーバとクライアント同士を通信するためのケーブルのようなものだと想像すればよい。

基本的にソケットプログラムでは、クライアントとサーバの2つが存在する。ソケットプログラムの基本動作としては、クライアントからの要求に対して加工データや必要なデータをクライアントに送信するという処理を行う。

## socket

Pythonにおけるソケットプログラムでは、**socket**と呼ばれるモジュールが使用される。**socket**は、サーバ側とクライアント側の両方で使用される。

### サーバ側

それでは、**socket**モジュールを使ってサーバを構築していこう。  
主にサーバ側は、以下の手順で作成していく。

1. IPアドレスとポート番号の設定
2. ソケットの作成
3. バインド
4. 接続可能回数の設定
5. 処理待ち(`accept`)
6. データの受信
7. データの送信
8. 処理を閉じる

上における処理をまとめると以下のようなソースコードになる

```
# -*- coding:utf-8 -*-
import socket
#サーバのIPとポート番号を決める
host = "127.0.0.1"
port = 6000

#ソケットを作成する
serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversock.bind((host, port))
serversock.listen(10)

#クライアントからの要求を待つ
print('Waiting for connections...')
clientsock, client_address = serversock.accept()

#クライアントからの通信後に処理をする部分
while True:
    #クライアントからのsendを受け取る
    rcvmsg = clientsock.recv(4096)
    #データをPythonで読めるように変換
    rcvmsg=rcvmsg.decode("utf-8")
    print ('Received -> %s' % (rcvmsg))
    #データがなければ処理をwhileを抜けて処理を終了する
    if rcvmsg == '':
        break
    print ('Recepi message...')
```

```
s_msg = "Get answer"
#データをバイト列に変換
s_msg=s_msg.encode("utf-8")
#クライアントに返事を返す
clientsock.send(s_msg)
print ("send message")
#クライアントとの処理を終了する部分
clientsock.close()
```

以下では、このソースコードを分解して何をしているかを説明していく。

## IPアドレスとポートの設定

```
#サーバのIPとポート番号を決める
host = "127.0.0.1"
port = 6000
```

サーバを立てる際には、IPアドレスと使用するポート番号を設定する必要がある。

IPアドレスには自分のPCが使用しているIPアドレスを設定する。なお、例ではlocalアドレスである127.0.0.1を設定している。

ポート番号では、現在自分が使っていないポート番号を使用する必要がある。ポート番号は、現在使用しているポート番号以外なら何でも入れられる。もし、使用ポートを使うとよくわからないエラーが出現することになる。

**注意事項として、IPアドレスは必ず文字型、ポート番号は整数型で設定する必要がある。**

## ソケット作成・bind・listen

サーバでは、クライアントとの通信を行うためのケーブル(ソケット)を作成する必要がある。このソケット作成では、以下のようなことを設定していく。

1. サーバがどのような通信を行うか。
2. どのIPとポートを使うか
3. どれぐらい通信するか

## どんな通信を行うか

まず、最初にこのサーバがどのような通信を行うかを設定していく。どのような通信を行うかというのは、IPv4で通信するのかIPv6で通信するのか、またTCPで通信かUDPで通信するのかということである。このような設定は、socket関数を使用する。socket関数は、socketモジュールでsocketにおけるどんな通信を行うかを設定する部分である。前節のソースでは以下に当たる。

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

socket関数の第一引数にはソケットのアドレスファミリーが設定される。アドレスファミリーとは、ソケット関数で使用されるアドレス構造の形式を決定するものである。簡単にいうと、通信に使用するのは、IPv4のパケット構造ですかそれともIPv6のパケット構造ですかというのを設定する部分である。前節のソースコードでは、第一引数がsocket.AF\_INETとなっているため、IPv4通信を意味する記述となっている。アドレスファミリーには様々なもののが存在するがここでは代表的なものだけを以下の表にまとめた。

| 定数           | 意味        |
|--------------|-----------|
| AF_INET      | IPv4      |
| AF_INET6     | IPv6      |
| AF_BLUETOOTH | BLUETOOTH |

次に第二引数では、プロトコルファミリーを設定する。プロトコルファミリーとは、どのようなプロトコルを使用するかを記述する部分である。簡単に言うとTCP通信にするかUDP通信にするかを設定するということである。前節の例では、`socket.SOCK_STREAM`なので、TCP通信を設定している。プロトコルファミリーにも多くのものが存在するが以下では代表的なもの表にまとめた。

| 定数          | 意味    |
|-------------|-------|
| SOCK_STREAM | TCP通信 |
| SOCK_DGRAM  | UDP通信 |

`serversock`は、ソケットディスクリプトである。これは、ソケット自体である。

## どのIPアドレスとポートを使うか

サーバソケットを作成する時は、指定したIPアドレスとポート番号をシステムに対して使うよと宣言し、使用済みにする必要がある。この操作を実行するのが`bind`である。

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

`bind`関数では、第一引数にIPアドレスを設定し、第二引数にポート番号を設定する。この関数では、IPアドレスとポート番号をタプル型にしなければならないことに注意する。タプル型とは複数の要素を固定の1つの要素にする型である。タプル型にするには、要素を()で囲めば良い。

## どれぐらい通信するか

サーバソケットの通信では、その接続をどれぐらいするのかを設定する必要がある。それを行うのが`listen`関数である。

```
serversock.listen(10)
```

## ソケット作成まとめ

```
#ソケットを作成する
serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversock.bind((host, port))
serversock.listen(10)
```

以上より上のコードは、TCP通信のIPv4で10回通信できるソケットを作成するという意味である。

## 処理待ち

サーバでは、クライアントからの受信を待たなければならない。もちろんPythonのソケットプログラミングでもこの動作を行う必要がある。Pythonでの処理待ちでは、`accept`関数を使用する。

```
clientsock, client_address = serversock.accept()
```

`accept`関数では、戻り値として、クライアントと通信を行うソケットと接続先のIPアドレスとポート番号が返信される。これからのクライアントとの通信にはクライアントソケットを使用していく。例では、クライアントソケットは、`clientsock`となっている。

## サーバ処理

サーバ処理では、クライアントから送信されたデータを加工する方法について記述する。サーバでの動作としては、2つしかない。1つはクライアントからのデータを受信することである。2つ目はクライアントからのデータを送信することである。サーバでは、この2つの動作の間に受信したデータを加工する処理を記述し、そのデータを返すということを実行している。この2つの詳細について次章から記述していく。

## データの受信

socketでのデータの受信では、**recv**関数を使用する。recv関数では、引数として受け取るためのデータサイズを設定する。データサイズはバッファと呼ばれている。データの大きさは、4096が良いとされている。

```
#クライアントからのデータを受信する
rcvmsg = clientsock.recv(4096)
#データをPythonで扱えるようにする
rcvmsg=rcvmsg.decode("utf-8")
```

recv関数の戻り値にはクライアントからのデータが返答される。しかし、Python3系ではこの受信した文字は扱うことができない。なぜなら、送信された文字はバイナリ型となっている。なので、受信した文字をutf-8に変換する必要がある。今回は、受信するためにバイト列をUTF-8にするのでdecode関数を使用する。decode関数は引数に変換する文字列を与え、戻り値に変換された文字列が返信される。

| 関数名    | 変換方法      | 使用用途        |
|--------|-----------|-------------|
| decode | バイト列->文字列 | 受信した文字を変換する |
| encode | 文字列->バイト列 | 送信文字を変換する   |

後述するが、送信時には、文字列をencode関数でバイト列に変換してから処理をする必要がある。

データの受信で使う関数には、様々な種類が存在する。基本的にはrecvを使ってもらえばよいが、以下の関数を用途に合わせて使いこなして欲しい。

| 関数名      | 動作  |
|----------|---|
| recv     | byte型でデータを受信する                                |
| recvfrom | タプル型(data,address)でデータを受信する                   |
| recvmsg  | タプル型(data,ancdata,msg_flags,address)でデータを受信する |

## データの送信

サーバでは、受け取った情報から特定の処理やテキストを返す必要がある。例えば、Webサーバでは、クライアントからのHTML要求に対してHTMLテキストを返すという処理を行う。このクライアントに対して処理を送信することは、**send**関数によって表現される。

```
s_msg = "Get answer"
#データをバイト列に変換する
s_msg=s_msg.encode("utf-8")
#クライアントに返事を返す
clientsock.send(s_msg)
```

send関数を使う前に返信する値を文字列からバイト列にする必要がある。これは、encode関数によって変換できる。send関数では、引数にこの変換した値を設定して送信する。

例では、Get answerという文字列をバイト列に変換してsend関数の引数に渡し、データを送信している。

データの送信で使う関数は、受信と同様に様々な種類が存在する。基本的にはsendを使えばよいが、以下の関数を用途に合わせて使いこなして欲しい。

| 関数名     | 動作                   |
|---------|----------------------|
| send    | byte型でデータを送信する       |
| sendall | 全データの送信及び送信エラーチェック   |
| sendto  | 送信したバイト数を戻り値として取得できる |

## 接続を閉じる

サーバでは、処理の終えたクライアントとの接続を閉じなければならない。これは、close関数によって実行される。

```
clientsocket.close()
```

## サーバまとめ

Pythonにおけるサーバプログラムでは、以下の構成でまとまっている。

```
import socket

-----
+ IPアドレスとポートの設定
-----

-----
+ ソケット作成(socket)
+ bind
+ listen
-----

-+ クライアントからの要求をまつ(accept)
-----


while True:
    -----
    + データを受信(recv)
    -----
    #受信データがない時
    if 受信データ==" ":
        break

    -----
    + データを送信(send)
    -----


+ クライアントを閉じる(close)
-----
```

この構成で記述したPythonのサーバプログラムが以下になる。このプログラムは、記述してあるが自分で記述することを心がけて欲しい。

```
# -*- coding:utf-8 -*-
import socket
#サーバのIPとポート番号を決める
host = "127.0.0.1"
port = 6000

#ソケットを作成する
serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```

serversock.bind((host,port))
serversock.listen(10)

#クライアントからの要求を待つ
print('Waiting for connections...')
clientsock, client_address = serversock.accept()

#クライアントからの通信後に処理をする部分
while True:
    #クライアントからのsendを受け取る
    rcvmsg = clientsock.recv(4096)
    #データをPythonで読めるように変換
    rcvmsg=rcvmsg.decode("utf-8")
    print ('Received -> %s' % (rcvmsg))
    #データがなければ処理をwhileを抜けて処理を終了する
    if rcvmsg == '':
        break
    print ('Recepi message...')
    s_msg = "Get answer"
    #データをバイト列に変換
    s_msg=s_msg.encode("utf-8")
    #クライアントに返事を返す
    clientsock.send(s_msg)
    print ("send message")
#クライアントとの処理を終了する部分
clientsock.close()

```

## クライアント

クライアント側では以下の手順を行う。

1. IPアドレスとポート番号の設定
2. ソケットを作成
3. サーバに接続する
4. サーバにデータを送信
5. サーバからデータを受信

上記の流れから以下のようなソースコードになる。

```

import socket
#IPとポート番号を設定
host = "127.0.0.1"
port = 6000

#ソケットを作成
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#サーバと接続する
client.connect((host, port))

#データをサーバに送信する
messages="Hello"
messages=messages.encode("utf-8")
client.send(messages)

#サーバからのデータを受信する
response = client.recv(4096)
response=response.decode("utf-8")
print(response)

```

## IPアドレスとポートの設定

クライアントで設定するIPアドレスとポート番号は、サーバ側で設定したのと同じものを設定する必要がある。

## ソケットを作成

ソケットの作成では、サーバで設定したアドレスファミリーとプロトコルファミリーを設定する必要がある。ここで作成されるのは、クライアントソケットであることに注意する。

## サーバに接続

サーバに接続する際にPythonでは、**connect関数**を使用する。

```
client.connect((host, port))
```

**connect**関数の引数には、タプルでまとめたIPアドレスとポート番号を設定する。

## 送信・受信

クライアントでは、サーバと同じで**send**関数と**recv**関数を使用する。ここで注意するのは、送信・受信時には**encode**関数と**decode**関数を行う必要がある。

## 実践

実際にサーバとクライアントを作成してみましょう。今回は、クライアントから数値を入力し送信し、その値をサーバ側で二乗した値をクライアントに返答する。

[目指す処理結果]

```
数字を入力してください
67
Answer:4489
```

注意：ソケットプログラムでは、**encode**と**decode**する前に**文字列**にする必要がある。なので、数値を扱うときは、**str**と**int**変換を気にしないといけない。

実践の答え例は、Practice1のディレクトリにあります。

## socketserver

前々節で**socket**モジュールでサーバを作成する方法を説明した。実際に体験してもらって感じたと思うが**socket**モジュールでサーバを作成するのは多くの設定を自分でやらないといけないため非常に面倒である。しかし、安心して欲しい。Pythonにはこれらの面倒な処理を無視して、簡単にサーバを構築することができる。そのモジュールの名前は、**socketserver**である。

**socketserver**は、ネットワークサービスを作成することができるフレームワークである。Python2系では、**SocketServer**であり、3系では、**socketserver**となっている。このモジュールは標準モジュールであるためインストールする必要はない。

**socketserver**では簡単にTCPやUDPのサーバを作成することができる。なお、**socketserver**では**socket**モジュールの関数を使用することができるので**send**や**recv**などが突然出現しても驚かないで欲しい。では、実際に**socketserver**で作成されたTCPサーバのソースコードを例にあげる。

```
import socketserver

#IPアドレスとポート番号を設定する
Host="127.0.0.1"
```

```

port=6000

#StreamRequestHandlerを継承して新しいクラスを定義する
class Myhandler(socketserver.StreamRequestHandler):
    def handle(self):
        while True:
            #クライアントからのデータを受け取る
            data=self.request.recv(1024).decode("utf-8")
            print(data)
            if len(data)==0:
                break
            request_data="GETS".encode("utf-8")
            #クライアントにデータを送信する
            self.request.send(request_data)
        #ハンドルを閉じる
        self.request.close()

#-----+
#TCPサーバを作成する
server=socketserver.TCPServer((Host,port),Myhandler)
print("listen", server.socket.getsockname())
#サーバを永久に稼働させる
server.serve_forever()

```

現在の状態では、何か難しいことをやっているように見えるかもしれないが1つずつ説明するので心配しないで欲しい。

まず、このプログラム自体は2つの部分に分けられることに注目して欲しい。具体的には、#-----+で分けた上と下の部分である。下の部分では、実際にサーバの種類の定義とサーバの全体的な動作を記述している。そして、上の部分ではクライアントの要求での動作を定義している。以下では、#----+より下の処理をサーバ定義、上の部分をリクエストハンドルとして話を進めることにする。

## サーバ定義

socketserverでは、そのサーバが何をするサーバかを最初に定義する必要がある。これは、サーバクラスによって定義することができる。socketserverでは、BaseServer, TCPServer, UDPServer, UnixStreamServer, UnixDatagramServerの5つが存在する。BaseServerは、すべてのサーバクラスの元となるものである。しかし、これ自体が単体で使用されることはない。

TCPServerは、TCP/IPのソケットを使って通信を行うサーバクラスである。また、UDPServerはUDP通信の利用を可能にさせるサーバクラスである。

UnixStreamServer, UnixDatagramServerは、Unixで利用可能なTCPServerとUDPServerからはせいしたものである。

主にサーバクラスでは、TCPServerとUDPServerが使用される。

| サーバオブジェクト          | 意味              |
|--------------------|-----------------|
| BaseServer         | サーバクラスの元クラス     |
| TCPServer          | TCPサーバのクラス      |
| UDPServer          | UDPサーバのクラス      |
| UnixStreamServer   | Unix用のTCPサーバクラス |
| UnixDatagramServer | Unix用のUDPサーバクラス |

以下では、TCPサーバクラスを定義している。

```

#IPアドレスとポート番号を設定する
Host="127.0.0.1"
port=6000

```

```
#TCPサーバを作成する
server=socketserver.TCPServer((Host,port),Myhandler)
```

上記でもわかるが、サーバクラスでは、使用するアドレスとポート番号と後述するリクエストハンドルを引数として渡すことになる。なお、アドレスとポート番号はタプルでひとまとめにして渡す必要がある。ここでは、プログラムに対してこんな動作（クライアントのデータに対してどう返信するかなど）をするサーバを作りますと宣言していると言える。また、サーバクラスでは、サーバオブジェクトを返り値として返却する（例のserver）。このオブジェクトでサーバの動作を設定することができます。主サーバオブジェクトでよく利用されるのは、serve\_forever関数である。この関数は、定義したサーバをずっと稼働させることを宣言するものである。これにより、常時稼働のサーバを作成することができる。Ctr-cで終了することができる。

```
#サーバを永久に稼働させる
server.serve_forever()
```

他にも多くの関数が存在するが詳しくはドキュメントを参考にして欲しい。ここまでで、サーバの定義ができるようになった。次章では、リクエストハンドルについて説明する。

## リクエストハンドル

リクエストハンドルは、クライアントからの処理に対してどのように処理をするかを記述する部分である。ここで、定義したリクエストハンドルがサーバクラスの第二引数に設定される。socketserverで継承元と使用されるリクエストハンドルクラスには、socketserver.BaseRequestHandler,socketserver.StreamRequestHandler,socketserver.DatagramRequestHandlerの3つのクラスがある。このうち1つを継承した独自リクエストクラスを定義することでリクエストハンドル作成ができる。ここで、継承とは、あるクラスの性質を新しく定義したクラスに受け継がせることである。これにより必要な設定を大幅に減らすことができる。主にリクエストハンドルの継承元としては、socketserver.StreamRequestHandlerの使用が多い。

実際に今回は、socketserver.StreamRequestHandlerを継承した独自リクエストハンドルであるMyhandlerクラスを定義した例を示す。

```
#StreamRequestHandlerを継承して新しいクラスを定義する
class Myhandler(socketserver.StreamRequestHandler):
    def handle(self):
        while True:
            #クライアントからのデータを受け取る
            data=self.request.recv(1024).decode("utf-8")
            print(data)
            if len(data)==0:
                break
            request_data="GETS".encode("utf-8")
            #クライアントにデータを送信する
            self.request.send(request_data)
        #ハンドルを閉じる
        self.request.close()
```

さて、実際にクライアントからのデータの処理は、Myhandlerクラス内のhandle関数で定義される。ここの中にクライアントからの受信データを処理させるクラスを記述することになる。ここで、注意して欲しいのが、クライアントからのリクエストを処理する場合には、関数名の前にself.requestが必要であるということである。実際の処理自体では、socketモジュールのときと同様にself.request.recv関数によりクライアントからの処理を受信し、self.request.send関数でGETという文字列をクライアントに送信し、self.request.close関数で、処理を終了している。ここで、思い出して欲しいことはデータの送受信はbyte列であるためencode,decode関数を使用しなければならないということである。この処理自体は、socketserverでも変わらないので注意してもらいたい。socketserverでの主な関数を表にまとめた。

| リクエスト関数                          | 意味                       |
|----------------------------------|--------------------------|
| self.request.recv(buffer_size)   | クライアントからのデータを受信          |
| self.request.send(send_data)     | クライアントにデータを送信            |
| self.request.send_all(send_data) | クライアントにデータを送信(sendの上位関数) |
| self.request.close()             | サーバの処理を終了                |

復習として再度decodeとencode関数を表としてまとめておく。

| 関数名    | 変換方法      | 使用用途   |
|--------|-----------|--------|
| decode | バイト列->文字列 | データ受信時 |
| encode | 文字列->バイト列 | データ送信時 |

handle関数での処理には、データの送受信だけではなく関数を外部から呼び出すといった複雑な処理をすることも可能である。その場合は、クラス定義の外に関数の定義を行い、handle関数内で定義した関数を呼び出す処理を記述することで処理を実現できる。詳しくは、後述の練習問題2を参考にして欲しい。

## socketserverまとめ

以上より、socketserverでは、どんなサーバを定義するかのサーバクラスとクライアントからのデータにどう動作するかを決定するリクエストハンドルを設定するという操作により完結することがわかる。そのため、socketserverは抽象的に設定すると以下のような構成となる。

```
import socketserver

-----
+IPアドレスとポート番号を設定
-----


-----
+リクエストハンドル
+リクエストクラスを定義
class Myhandler(継承元のサーバクラス):
    +handle関数
    def handle(self):
        while True:
            -----
            +データの受信(self.request.recv)
            +データ加工
            +データの送信(self.request.send)
            -----
            +処理を終了(self.request.close)
            -----


-----
+サーバを定義
server=socketserver.TCPServer((Host,port),Myhandler)
+サーバ関数(server.serv_forever)
-----
```

## socketserverを動かそう

Socket2ディレクトリに以下のようなTCPサーバプログラムとクライアントプログラムをサンプルとしておいた。  
**TCPサーバプログラム**

```

import socketserver

#IPアドレスとポート番号を設定する
Host="127.0.0.1"
port=6000

#StreamRequestHandlerを継承して新しいクラスを定義する
#ここでhandleの中身はユーザが勝手に設定してよい
class Myhandler(socketserver.StreamRequestHandler):
    def handle(self):
        while True:
            #以下からのself.requestはクライアントからの要求に対する属性
            #クライアントからのデータを受け取る
            data=self.request.recv(1024).decode("utf-8")
            print(data)
            #受信データがなければwhileをぬける
            if len(data)==0:
                break
            request_data="GETS".encode("utf-8")
            #クライアントにデータを送信する
            self.request.send(request_data)
        #ハンドルを閉じる
        self.request.close()

#実際のmainはここからははじまる
#TCPサーバを作成する
#引数にはIPとポート番号,自分が定義したハンドルクラスを設定
server=socketserver.TCPServer((Host,port),Myhandler)
#getsocknameは現在自分が設定したソケットの情報が表示される
print("listen", server.socket.getsockname())
#サーバを永久に稼働させる
server.serve_forever()

```

socketserverサーバに対して、クライアントプログラム自体は、socketモジュールで作成したものと同じでよい。作成の仕方は全く同じでよい。この点もsocketserverを使う利点であると言える。

### クライアントプログラム

```

import socket

host = "127.0.0.1"
port = 6000

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client.connect((host, port))
messages="Hello"
messages=messages.encode("utf-8")
client.send(messages)

response = client.recv(4096)
response=response.decode("utf-8")
print(response)

```

では、実際にターミナルを2つ用意して動作するかを見ていこう。次にSocket2ディレクトリに移動し、Server2.pyを最初に起動していただきたい。すると以下のようない表示が表示されたと思う。

```
listen ('127.0.0.1', 6000)
```

この表示があれば、サーバが起動したことを示している。その後、client.pyを起動していただきたい。するとサーバ側に以下の処理が表示されたと思う。

```
listen ('127.0.0.1', 6000)
```

```
Hello
```

動作が確認できれば実際に練習問題を解いて理解を深めることを進める。

## 実践2

実際にsocketserverを使って、クライアントの要求を計算するサーバを作成してみましょう。

クライアントからの入力では、演算子と2つの値を送信することにする。送信されるデータはカンマ(,)区切りで連結した文字とする。例としては、**演算子,1つ目の値,2つ目の値**としたデータを送信する。なので、サーバ側の処理では、そのデータから計算を行い、データを返すようにする。実際に期待される動作としては以下のようなものである。

```
計算結果を返すサービスです  
実行する演算子を入力してください  
足し算--->add  
引き算--->sub  
掛け算--->mul  
割り算--->div  
add  
1つ目の数を入力してください  
7  
2つ目の数を入力してください  
4  
[Result]  
11
```

ただし、入力にはinput関数を使い、サーバクラスにはTCPServer, socketserver.StreamRequestHandlerを継承元としてリクエストハンドルクラスを作成すること。

模範解答は、Practice2内に存在する。

## ソケットプログラム

ソケットプログラムとは、サーバとクライアントの通信をプログラムしたものである。通常、ソケットとは、TCP/IPアプリケーションを作成するための抽象化されたインターフェースとされている。簡単に言うとソケットは、サーバとクライアント同士を通信するためのケーブルのようなものだと想像すればよい。

基本的にソケットプログラムでは、クライアントとサーバの2つが存在する。ソケットプログラムの基本動作としては、クライアントからの要求に対して加工データや必要なデータをクライアントに送信するという処理を行う。

## socket

Pythonにおけるソケットプログラムでは、**socket**と呼ばれるモジュールが使用される。socketは、サーバ側とクライアント側の両方で使用される。

## サーバ側

それでは、socketモジュールを使ってサーバを構築していこう。  
主にサーバ側は、以下の手順で作成していく。

1. IPアドレスとポート番号の設定
2. ソケットの作成
3. バインド
4. 接続可能回数の設定
5. 処理待ち(accept)
6. データの受信
7. データの送信
8. 処理を閉じる

上における処理をまとめると以下のようなソースコードになる…

## IPアドレスとポートの設定

クライアントで設定するIPアドレスとポート番号は、サーバ側で設定したのと同じものを設定する必要がある。

## ソケット作成・bind・listen

サーバでは、クライアントとの通信を行うためのケーブル(ソケット)を作成する必要がある。このソケット作成では、以下のようなことを設定していく。

1. サーバがどのような通信を行うか。
2. どのIPとポートを使うか
3. どれぐらい通信するか

## どんな通信を行うか

まず、最初にこのサーバがどのような通信を行うかを設定していく。どのような通信を行うかというのは、IPv4で通信するのかIPv6で通信するのか、またTCPで通信かUDPで通信するのかということである。このような設定は、**socket関数**を使用する。**socket関数**は、socketモジュールでsocketにおけるどんな通信を行うかを設定する部分である。前節のソースでは以下に当たる。

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

socket関数の第一引数にはソケットのアドレスファミリーが設定される。アドレスファミリーとは、ソケット関数で使用されるアドレス構造の形式を決定するものである。簡単にいうと、通信に使用するのは、IPv4のパケット構造ですかそれともIPv6のパケット構造ですかというのを設定する部分である。前節のソースコードでは、第一引数がsocket.AF\_INETとなっているため、IPv4通信を意味する記述となっている。アドレスファミリーには様々なもののが存在するがここでは代表的なものだけを以下の表にまとめた。

| 定数           | 意味        |
|--------------|-----------|
| AF_INET      | IPv4      |
| AF_INET6     | IPv6      |
| AF_BLUETOOTH | BLUETOOTH |

次に第二引数では、プロトコルファミリーを設定する。プロトコルファミリーとは、どのようなプロトコルを使用するかを記述する部分である。簡単に言うとTCP通信にするかUDP通信にするかを設定するということである。前節の例では、socket.SOCK\_STREAMなので、TCP通信を設定している。プロトコルファミリーにも多くのものが存在するが以下では代表的なもの表にまとめた。

| 定数          | 意味    |
|-------------|-------|
| SOCK_STREAM | TCP通信 |
| SOCK_DGRAM  | UDP通信 |

serversockは、ソケットディスクリプトである。これは、ソケット自体である。

## どのIPアドレスとポートを使うか

サーバソケットを作成する時は、指定したIPアドレスとポート番号をシステムに対して使うよと宣言し、使用済みにする必要がある。この操作を実行するのが**bind**である。

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

bind関数では、第一引数にIPアドレスを設定し、第二引数にポート番号を設定する。この関数では、IPアドレスとポート番号をタプル型にしなければならないことに注意する。タプル型とは複数の要素を固定の1つの要素にする型である。タプル型にするには、要素を()で囲めば良い。

## どれぐらい通信するか

サーバソケットの通信では、その接続をどれくらいするのかを設定する必要がある。それを行うのが**listen**関数である。

```
serversock.listen(10)
```

## ソケット作成まとめ

```
#ソケットを作成する
serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversock.bind((host,port))
serversock.listen(10)
```

以上より上のコードは、TCP通信のIPv4で10回通信できるソケットを作成するという意味である。

## 処理待ち

サーバでは、クライアントからの受信を待たなければならない。もちろんPythonのソケットプログラミングでもこの動作を行う必要がある。Pythonでの処理待ちでは、**accept関数**を使用する。

```
clientsock, client_address = serversock.accept()
```

accept関数では、戻り値として、クライアントと通信を行うソケットと接続先のIPアドレスとポート番号が返信される。これからクライアントとの通信にはクライアントソケットを使用していく。例では、クライアントソケットは、**clientsock**となっている。

## サーバ処理

サーバ処理では、クライアントから送信されたデータを加工する方法について記述する。サーバでの動作としては、2つしかない。1つはクライアントからのデータを受信することである。2つ目はクライアントからのデータを送信することである。サーバでは、この2つの動作の間に受信したデータを加工する処理を記述し、そのデータを返すということを実行している。この2つの詳細について次章から記述していく。

## データの受信

socketでのデータの受信では、**recv**関数を使用する。recv関数では、引数として受け取るためのデータサイズを設定する。データサイズはバッファと呼ばれている。データの大きさは、4096が良いとされている。

```
#クライアントからのデータを受信する
rcvmsg = clientsock.recv(4096)
#データをPythonで扱えるようにする
rcvmsg=rcvmsg.decode("utf-8")
```

recv関数の戻り値にはクライアントからのデータが返答される。しかし、Python3系ではこの受信した文字は扱うことができない。なぜなら、送信された文字はバイナリ型となっている。なので、受信した文字をutf-8に変換する必要がある。今回は、受信するためにバイト列をUTF-8にするのでdecode関数を使用する。decode関数は引数に変換する文字列を与え、戻り値に変換された文字列が返信される。

| 関数名    | 変換方法      | 使用用途        |
|--------|-----------|-------------|
| decode | バイト列->文字列 | 受信した文字を変換する |
| encode | 文字列->バイト列 | 送信文字を変換する   |

後述するが、送信時には、文字列をencode関数でバイト列に変換してから処理をする必要がある。

データの受信で使う関数には、様々な種類が存在する。基本的にはrecvを使ってもらえばよいが、以下の関数を用途に合わせて使いこなして欲しい。

| 関数名      | 動作  |
|----------|---|
| recv     | byte型でデータを受信する                                |
| recvfrom | タプル型(data,address)でデータを受信する                   |
| recvmsg  | タプル型(data,ancdata,msg_flags,address)でデータを受信する |

## データの送信

サーバでは、受け取った情報から特定の処理やテキストを返す必要がある。例えば、Webサーバでは、クライアントからのHTML要求に対してHTMLテキストを返すという処理を行う。このクライアントに対して処理を送信することは、**send**関数によって表現される。

```
s_msg = "Get answer"  
#データをバイト列に変換する  
s_msg=s_msg.encode("utf-8")  
#クライアントに返事を返す  
clientsock.send(s_msg)
```

**send**関数を使う前に返信する値を文字列からバイト列にする必要がある。これは、**encode**関数によって変換できる。**send**関数では、引数にこの変換した値を設定して送信する。

例では、**Get answer**という文字列をバイト列に変換して**send**関数の引数に渡し、データを送信している。

データの送信で使う関数は、受信と同様に様々な種類が存在する。基本的には**send**を使えばよいが、以下の関数を用途に合わせて使いこなして欲しい。

| 関数名     | 動作                   |
|---------|----------------------|
| send    | byte型でデータを送信する       |
| sendall | 全データの送信及び送信エラーチェック   |
| sendto  | 送信したバイト数を戻り値として取得できる |

## 接続を閉じる

サーバでは、処理の終えたクライアントとの接続を閉じなければならない。これは、`close`関数によって実行される。

```
clientsocket.close()
```

## サーバまとめ

Pythonにおけるサーバプログラムでは、以下の構成でまとまっている。

```
import socket

-----
+ IPアドレスとポートの設定
-----

-----
+ ソケット作成(socket)
+ bind
+ listen
-----

-----
+ クライアントからの要求をまつ(accept)
-----

while True:
    -----
    + データを受信(recv)
    -----
    #受信データがない時
    if 受信データ==" ":
        break

    -----
    + データを送信(send)
    -----


-----
+ クライアントを閉じる(close)
-----
```

この構成で記述したPythonのサーバプログラムが以下になる。このプログラムは、記述してあるが自分で記述することを心がけて欲しい。``

## クライアント

クライアント側では以下の手順を行う。

1. IPアドレスとポート番号の設定
2. ソケットを作成
3. サーバに接続する
4. サーバにデータを送信
5. サーバからデータを受信

上記の流れから以下のようなソースコードになる。

```
import socket
#IPとポート番号を設定
host = "127.0.0.1"
port = 6000

#ソケットを作成
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#サーバと接続する
client.connect((host, port))

#データをサーバに送信する
messages="Hello"
messages=messages.encode("utf-8")
client.send(messages)

#サーバからのデータを受信する
response = client.recv(4096)
response=response.decode("utf-8")
print(response)
```

## ソケットを作成

ソケットの作成では、サーバで設定したアドレスファミリーとプロトコルファミリーを設定する必要がある。ここで作成されるのは、クライアントソケットであることに注意する。

## サーバに接続

サーバに接続する際にPythonでは、 **connect関数**を使用する。

```
client.connect((host, port))
```

connect関数の引数には、 タプルでまとめたIPアドレスとポート番号を設定する。

## 送信・受信

クライアントでは、サーバと同じでsend関数とrecv関数を使用する。ここで注意するのは、送信・受信時にはencode関数とdecode関数を行う必要がある。

## 実践

実際にサーバとクライアントを作成してみましょう。今回は、クライアントから数値を入力し送信し、その値をサーバ側で二乗した値をクライアントに返答する。

[目指す処理結果]

数字を入力してください

67

Answer:4489

注意：ソケットプログラムでは、`encode`と`decode`する前に文字列にする必要がある。なので、数値を扱うときは、`str`と`int`変換を気にしないといけない。

実践の答え例は、Practice1のディレクトリにあります。

## socketserver

前々節でsocketモジュールでサーバを作成する方法を説明した。実際に体験してもらって感じたと思うがsocketモジュールでサーバを作成するのは多くの設定自分でやらないといけないため非常に面倒である。しかし、安心して欲しい。Pythonにはこれらの面倒な処理を無視して、簡単にサーバを構築することができる。そのモジュールの名前は、socketserverである。

socketserverは、ネットワークサービスを作成することができるフレームワークである。Python2系では、SocketServerであり、3系では、socketserverとなっている。このモジュールは標準モジュールであるためインストールする必要はない。

socketserverでは簡単にTCPやUDPのサーバを作成することができる。なお、socketserverではsocketモジュールの関数を使用することができるのでsendやrecvなどが突然出現しても驚かないで欲しい。では、実際にsocketserverで作成されたTCPサーバのソースコードを例にあげる。

```
import socketserver

#IPアドレスとポート番号を設定する
Host="127.0.0.1"
port=6000

#StreamRequestHandlerを継承して新しいクラスを定義する
class Myhandler(socketserver.StreamRequestHandler):
    def handle(self):
        while True:
            #クライアントからのデータを受け取る
            data=self.request.recv(1024).decode("utf-8")
            print(data)
            if len(data)==0:
                break
            request_data="GETS".encode("utf-8")
            #クライアントにデータを送信する
            self.request.send(request_data)
        #ハンドルを閉じる
        self.request.close()

#-----+
#TCPサーバを作成する
server=socketserver.TCPServer((Host,port),Myhandler)
print("listen", server.socket.getsockname())
#サーバを永久に稼働させる
server.serve_forever()
```

現在の状態では、何か難しいことをやっているように見えるかもしれないが1つずつ説明するので心配しないで欲しい。

まず、このプログラム自体は2つの部分に分けられることに注目して欲しい。具体的には、#-----+で分けた上と下の部分である。下の部分では、実際にサーバの種類の定義とサーバの全体的な動作を記述している。そして、上の部分ではクライアントの要求での動作を定義している。以下では、#----+より下の処理をサーバ定義、上の部分をリクエストハンドルとして話を進めることにする。

## サーバ定義

socketserverでは、そのサーバが何をするサーバかを最初に定義する必要がある。これは、サーバクラスによって定義することができる。socketserverでは、BaseServer, TCPServer, UDPServer, UnixStreamServer, UnixDatagramServerの5つが存在する。BaseServerは、すべてのサーバクラスの元となるものである。しかし、これ自体が単体で使用されることはない。TCPserverは、TCP/IPのソケットを使って通信を行うサーバクラスである。また、UDPServerはUDP通信の利用を可能にさせるサーバクラスである。UnixStreamServer, UnixDatagramServerは、Unixで利用可能なTCPServerとUDPServerからはせいしたものである。主にサーバクラスでは、TCPServerとUDPServerが使用される。

| サーバオブジェクト          | 意味              |
|--------------------|-----------------|
| BaseServer         | サーバクラスの元クラス     |
| TCPServer          | TCPサーバのクラス      |
| UDPServer          | UDPサーバのクラス      |
| UnixStreamServer   | Unix用のTCPサーバクラス |
| UnixDatagramServer | Unix用のUDPサーバクラス |

以下では、TCPサーバクラスを定義している。

```
#IPアドレスとポート番号を設定する
Host="127.0.0.1"
port=6000

#TCPサーバを作成する
server=socketserver.TCPServer((Host,port),Myhandler)
```

上記でもわかるが、サーバクラスでは、使用的アドレスとポート番号と後述するリクエストハンドルを引数として渡すことになる。なお、アドレスとポート番号はタプルでひとまとめにして渡す必要がある。ここでは、プログラムに対してこんな動作（クライアントのデータに対してどう返信するかなど）をするサーバを作りますと宣言していると言える。また、サーバクラスでは、サーバオブジェクトを返り値として返却する（例のserver）。このオブジェクトでサーバの動作を設定することができます。主サーバオブジェクトでよく利用されるのは、serve\_forever関数である。この関数は、定義したサーバをずっと稼働させることを宣言するものである。これにより、常時稼働のサーバを作成することができる。Ctr-cで終了することができる。

```
#サーバを永久に稼働させる
server.serve_forever()
```

他にも多くの関数が存在するが詳しくはドキュメントを参考にして欲しい。ここまでで、サーバの定義ができるようになった。次章では、リクエストハンドルについて説明する。

## リクエストハンドル

リクエストハンドルは、 クライアントからの処理に対してどのように処理をするかを記述する部分である。ここで、 定義したリクエストハンドルがサーバクラスの第二引数に設定される。socketserverで継承元と使用されるリクエストハンドルクラスには、 socketserver.BaseRequestHandler, socketserver.StreamRequestHandler, socketserver.DatagramRequestHandlerの3つのクラスがある。このうち1つを継承した独自リクエストクラスを定義することでリクエストハンドル作成ができる。ここで、 継承とは、 あるクラスの性質を新しく定義したクラスに受け継がせることである。これにより必要な設定を大幅に減らすことができる。主にリクエストハンドルの継承元としては、 socketserver.StreamRequestHandlerの使用が多い。

実際に今回は、 socketserver.StreamRequestHandlerを継承した独自リクエストハンドルであるMyhandlerクラスを定義した例を示す。

```
#StreamRequestHandlerを継承して新しいクラスを定義する
class Myhandler(socketserver.StreamRequestHandler):
    def handle(self):
        while True:
            #クライアントからのデータを受け取る
            data=self.request.recv(1024).decode("utf-8")
            print(data)
            if len(data)==0:
                break
            request_data="GETS".encode("utf-8")
            #クライアントにデータを送信する
            self.request.send(request_data)
        #ハンドルを閉じる
        self.request.close()
```

さて、 実際にクライアントからのデータの処理は、 Myhandlerクラス内のhandle関数で定義される。ここの中にクライアントからの受信データを処理させるクラスを記述することになる。ここで、 注意して欲しいのが、 クライアントからのリクエストを処理する場合には、 関数名の前にself.requestが必要であるということである。

実際の処理自体では、 socketモジュールのときと同様にself.request.recv関数によりクライアントからの処理を受信し、 self.request.send関数でGETという文字列をクライアントに送信し、 self.request.close関数で、 処理を終了している。ここで、 思い出して欲しいことはデータの送受信はbyte列であるためencode, decode関数を使用しなければならないということである。この処理自体は、 socketserverでも変わらないので注意してもらいたい。socketserverでの主な関数を表にまとめた。

| リクエスト関数                          | 意味                       |
|----------------------------------|--------------------------|
| self.request.recv(buffer_size)   | クライアントからのデータを受信          |
| self.request.send(send_data)     | クライアントにデータを送信            |
| self.request.send_all(send_data) | クライアントにデータを送信(sendの上位関数) |
| self.request.close()             | サーバの処理を終了                |

復習として再度decodeとencode関数を表としてまとめておく。

| 関数名    | 変換方法      | 使用用途   |
|--------|-----------|--------|
| decode | バイト列->文字列 | データ受信時 |
| encode | 文字列->バイト列 | データ送信時 |

handle関数での処理には、 データの送受信だけではなく関数を外部から呼び出すといった複雑な処理をすることも可能である。その場合は、 クラス定義の外に関数の定義を行い、 handle関数内で定義した関数を呼び出す処理を記述することで処理を実現できる。詳しくは、 後述の練習問題2を参考にして欲しい。



## socketserverまとめ

以上より、socketserverでは、どんなサーバを定義するかのサーバクラスとクライアントからのデータにどう動作するかを決定するリクエストハンドルを設定するという操作により完結することがわかる。そのため、socketserverは抽象的に設定すると以下のような構成となる。

```
import socketserver

-----
+IPアドレスとポート番号を設定
-----


-----
+リクエストハンドル
+リクエストクラスを定義
class Myhandler(継承元のサーバクラス):
    +handle関数
    def handle(self):
        while True:
            -----
            +データの受信(self.request.recv)
            +データ加工
            +データの送信(self.request.send)
            -----


            -----
            +処理を終了(self.request.close)
            -----


-----
+サーバを定義
server=socketserver.TCPServer((Host,port),Myhandler)
+サーバ関数(server.serv_forever)
-----
```

## socketserverを動かそう

Socke2ディレクトリに以下のようなTCPサーバプログラムとクライアントプログラムをサンプルとしておいた。  
**TCPサーバプログラム**

```
import socketserver

#IPアドレスとポート番号を設定する
Host="127.0.0.1"
port=6000

#StreamRequestHandlerを継承して新しいクラスを定義する
#ここでhandleの中身はユーザが勝手に設定してよい
class Myhandler(socketserver.StreamRequestHandler):
    def handle(self):
        while True:
            #以下からのself.requestはクライアントからの要求に対する属性
            #クライアントからのデータを受け取る
            data=self.request.recv(1024).decode("utf-8")
            print(data)
            #受信データがなければwhileをぬける
            if len(data)==0:
                break
            request_data="GETS".encode("utf-8")
            #クライアントにデータを送信する
            self.request.send(request_data)
        #ハンドルを閉じる
        self.request.close()

#実際のmainはここからははじまる
#TCPサーバを作成する
#引数にはIPとポート番号,自分が定義したハンドルクラスを設定
server=socketserver.TCPServer((Host,port),Myhandler)
#getsocknameは現在自分が設定したソケットの情報が表示される
print("listen", server.socket.getsockname())
#サーバを永久に稼働させる
server.serve_forever()
```

socketserverサーバに対して、クライアントプログラム自体は、socketモジュールで作成したものと同じでよい。作成の仕方は全く同じでよい。この点もsocketserverを使う利点であると言える。

### クライアントプログラム

```
import socket

host = "127.0.0.1"
port = 6000

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client.connect((host, port))
messages="Hello"
messages=messages.encode("utf-8")
client.send(messages)

response = client.recv(4096)
response=response.decode("utf-8")
print(response)
```

では、実際にターミナルを2つ用意して動作するかを見ていこう。次にSocke2ディレクトリに移動し、Server2.pyを最初に起動していただきたい。すると以下のようないい出力が表示されたと思う。

```
listen ('127.0.0.1', 6000)
```

この表示がでれば、サーバが起動したことを示している。その後、client.pyを起動していただきたい。するとサーバ側に以下の処理が表示されたと思う。

```
listen ('127.0.0.1', 6000)
Hello
```

動作が確認できれば実際に練習問題を解いて理解を深めることを進める。

## 実践2

実際にsocketserverを使って、クライアントの要求を計算するサーバを作成してみましょう。

クライアントからの入力では、演算子と2つの値を送信することにする。送信されるデータはカンマ(,)区切りで連結した文字とする。例としては、**演算子,1つ目の値,2つ目の値**としたデータを送信する。なので、サーバ側の処理では、そのデータから計算を行い、データを返すようにする。実際に期待される動作としては以下のようなものである。

```
計算結果を返すサービスです  
実行する演算子を入力してください  
足し算--->add  
引き算--->sub  
掛け算--->mul  
割り算--->div  
add  
1つ目の数を入力してください  
7  
2つ目の数を入力してください  
4  
[Result]  
11
```

ただし、入力にはinput関数を使い、サーバクラスにはTCPServer, socketserver.StreamRequestHandlerを継承元としてリクエストハンドルクラスを作成すること。

模範解答は、Practice2内に存在する。