

PYTHON

# PROTOCOLO DE COMUNICAÇÃO E PYTHON

HUMBERTO D. DE SOUSA



**LISTA DE FIGURAS**

Figura 7.1 – Servidor.....	8
Figura 7.2 – Cliente.....	10
Figura 7.3 – Executando o Cliente antes do Servidor. ....	12
Figura 7.4 – Executando o Servidor. ....	12
Figura 7.5 – Executando o Cliente. ....	12
Figura 7.6 – Resposta do Servidor.....	12
Figura 7.7 – Acesso ao servidor FTP ibiblio via browser.....	20

**LISTA DE CÓDIGOS-FONTE**

Código-fonte 7.1 – Retornando IP.....	6
Código-fonte 7.2 – Identificando Portas. ....	7
Código-fonte 7.3 – Código para servidor.....	8
Código-fonte 7.4 – Código para cliente. ....	11
Código-fonte 7.5 – Código para Servidor_Chat.....	13
Código-fonte 7.6 – Código para Cliente_Chat.....	14
Código-fonte 7.7 – Código para finalizar o Cliente. ....	14
Código-fonte 7.8 – Código para servidor UDP. ....	16
Código-fonte 7.9 – Código para cliente UDP.....	17
Código-fonte 7.10 – Primeira conexão FTP. ....	19
Código-fonte 7.11 – Efetuando login no servidor FTP.....	21
Código-fonte 7.12 – Mudando de diretórios no servidor FTP.....	22
Código-fonte 7.13 – Listando os arquivos do diretório corrente. ....	22
Código-fonte 7.14 – Baixando arquivos ASCII. ....	23
Código-fonte 7.15 – Baixando arquivos binários. ....	24
Código-fonte 7.16 – Exemplo de um FTP interativo.....	25

## SUMÁRIO

7 PROTOCOLO DE COMUNICAÇÃO E PYTHON .....	5
7.1 Tudo começa com a biblioteca "socket" .....	5
7.1.1 Preparação do servidor .....	7
7.1.2 Preparação do cliente.....	10
7.1.3 Chat Python.....	13
7.1.4 Protocolo UDP.....	15
7.2 O mundo não é feito só de sockets .....	18
7.2.1 Um exemplo FTP mais interativo .....	25
REFERÊNCIAS.....	29

## 7 PROTOCOLO DE COMUNICAÇÃO E PYTHON

Neste conteúdo, veremos como o Python pode se interligar com diversas áreas. Até o momento no seu curso, foram apresentados os conceitos básicos da linguagem e um pouco de aplicações externas, como, por exemplo, a utilização dos pacotes externos, como fizemos com o Pygeocoder. A missão, a partir de agora, é demonstrar as várias áreas de atuação em que poderemos utilizar Python.

Agora entenderemos como ele pode interagir com alguns protocolos, saberemos mais adiante como ele será utilizado com IoT, um pouco mais à frente, também verificaremos a ligação do Python com Inteligência Artificial e finalizaremos o conteúdo deste ano em Python comunicando-se com Cybersecurity junto a ferramentas de pentest. Então, agora é abrir o seu IDE e vamos *codar*.

### 7.1 Tudo começa com a biblioteca “socket”

A biblioteca *socket* é integrante do conjunto de bibliotecas-padrão que já fazem parte da PVM do Python, ou seja, diferentemente da Pygeocoder, ela não precisa ser instalada por meio do comando “pip”, apresentado anteriormente. Para utilizar essa biblioteca, basta realizar o *import* na primeira linha do seu código, da seguinte forma:

```
import socket
```

Você deve estar se perguntando: por que preciso importar essa biblioteca? Qual a sua finalidade? Pois bem, essa é a biblioteca que será responsável por criar ferramentas e estabelecer comunicação entre dois processos, ou mais, que podem estar em uma rede de computadores. Por isso, vamos relembrar os principais protocolos envolvidos em uma transmissão de dados entre computadores:

- IP: que representa o endereço do qual partiu ou para o qual irá determinado pacote de dados, ou seja, o endereçamento lógico;
- TCP: protocolo de transporte que possui maior confiabilidade na entrega dos pacotes, devido ao processo de “handshake” que ele realiza; e

- UDP: protocolo de transporte que possui maior velocidade, porém menor confiabilidade de entrega (quando comparado ao TCP), uma vez que ele não realiza “handshake”.

Claro que essas definições apenas pretendem relembrar os conceitos básicos desses protocolos.

Vejamos um primeiro exemplo para recuperarmos o endereço IP por meio do DNS:

- Crie um novo diretório chamado: **Capitulo7\_Socket**, dentro dele, crie um arquivo do tipo “Python File”, chamado: **Socket.py**, e digite o seguinte código:

```
import socket
resp="S"
while(resp=="S"):
    url=input("Digite uma url: ")
    ip=socket.gethostbyname(url)
    print("O IP referente à url informada é: ", ip)
    resp=input("Digite <s> para continuar: ").upper()
```

Código-fonte 7.1 – Retornando IP  
Fonte: Elaborado pelo autor (2018)

Nesse exemplo, estamos:

- Importando a biblioteca *socket*.
- Criando uma variável para controlar o nosso laço, que virá na sequência.
- Criando um laço para que, enquanto você digita a letra “s”, o código continue lhe perguntando uma url e exibindo o seu IP. O laço somente será encerrado caso digite algo diferente de “s” ou se digitar uma url inexistente.
- Dentro do laço, temos uma variável chamada “url” que simplesmente irá armazenar o domínio que está tentando localizar o IP, por isso, digite algo como: *www.globo.com*.
- Logo depois, colocamos a variável “ip” para receber o conteúdo do método *gethostbyname()*, que será o IP baseado no valor da variável “url” que está dentro dos parênteses, ou seja, *gethostbyname()* é um método pertencente à classe *socket*, que recebe uma URL e retorna o IP.

- Por fim, exibimos o IP na tela para o usuário.

Simples, não é mesmo? Mas até o momento vimos o `socket` somente agindo junto ao protocolo IP, que, por sua vez, ainda necessita de uma porta para que pacotes possam ser direcionados de maneira íntegra e correta.

Podemos identificar as portas da nossa máquina que estão sendo utilizadas para serviços, como domínio, http e ftp, por exemplo. Crie um novo arquivo “Python File” com o nome “Portas.py” e digite o código a seguir:

```
import socket

print(socket.getservbyname("domain"))
print(socket.getservbyname("http"))
print(socket.getservbyname("ftp"))
```

Código-fonte 7.2 – Identificando Portas  
Fonte: Elaborado pelo autor (2018)

Com a execução desse código, teremos o retorno das portas que estamos disponibilizando para: domínio (por padrão 53), usado para resolver a conversão entre DNS e IP; HTTP (por padrão 80), usado para navegar nas páginas WEB; e FTP (por padrão 21), usado para transferência de arquivos.

Existem “n” portas que possuem números padrões definidos pela IANA e que permitem o acesso a serviços diversos. Para maiores informações sobre as portas, recomendo uma olhada no link: <http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>.

Agora, que já sabemos como recuperar um IP e como identificar as portas, já é possível estabelecer um diálogo entre dois processos que podem estar sendo executados em máquinas separadas dentro de uma mesma rede, ou não; os processos podem estar rodando em um mesmo computador.

### 7.1.1 Preparação do servidor

Vamos preparar duas ferramentas, uma será utilizada para ficar no servidor, isso significa que será a ferramenta que ficará escutando e atendendo aos pedidos dos clientes. A segunda ferramenta deverá ficar nos clientes, ou seja, é a ferramenta que irá fazer as solicitações para o servidor.



Figura 7.1 – Servidor.  
Fonte: www.penso.com.br (2018)

Vamos começar criando um arquivo “Python File”, chamado **Servidor.py**, que a princípio deverá conter os seguintes comandos:

```
from socket import *
servidor="127.0.0.1"
porta=43210
obj_socket = socket(AF_INET, SOCK_STREAM)
obj_socket.bind((servidor,porta))
obj_socket.listen(2)
print("Aguardando cliente...")
while True:
    con, cliente = obj_socket.accept()
    print("Conectado com: ", cliente)
    while True:
        msg_recebida = str(con.recv(1024))
        print("Recebemos: ", msg_recebida)
        msg_enviada = b'Olah cliente'
        con.send(msg_enviada)
        break
    con.close()
```

Código-fonte 7.3 – Código para servidor  
Fonte: elaborado pelo autor (2017)

Vamos a uma explicação sobre o que está ocorrendo nesse código:

- Primeiramente, importamos todas (\*) as funções da biblioteca “socket”, isso será preciso para que possamos criar objetos do tipo “socket”.



- Criamos duas variáveis, “servidor” e “porta”, para armazenarmos esses dois dados que serão utilizados posteriormente; em “servidor”, poderíamos utilizar também a palavra “localhost” para identificarmos que o servidor é a própria máquina que está executando o código, o que também é representado pelo endereço “127.0.0.1”, que foi aqui utilizado. Em “porta”, escolhemos uma porta representada por um número inteiro entre 0 e 65535, em que normalmente as portas entre 0 e 1023 são portas utilizadas, por padrão, para atribuições de serviços conhecidos por meio do sistema operacional.
- Na quarta linha, criamos o nosso objeto `socket` “obj\_socket”, por meio da função “`socket()`”, que exige dois parâmetros: o primeiro definirá a família responsável por identificar os pacotes. Para o nosso exemplo, definimos como `AF_INET`, o que significa que iremos utilizar a identificação do emissor/receptor do(s) pacote(s) via DNS ou número IP (poderíamos utilizar a constante `AF_UNIX`, o que mudaria a forma de identificar a origem/destino do(s) pacote(s)). Já o segundo parâmetro refere-se ao transporte desse pacote, que pode ser `SOCK_STREAM`, que representa o protocolo TCP (o que nós optamos) ou `SOCK_DGRAM`, que representa o protocolo de transporte UDP.
- Na próxima linha, fazemos a associação no nosso objeto `socket` com o nosso servidor e porta.
- Na linha em que utilizamos a função “`listen()`”, estamos definindo o máximo de clientes que o nosso servidor irá atender simultaneamente, para o nosso caso, definimos que serão, no máximo, dois (2) clientes.
- Montamos, na sequência, dois laços infinitos (enquanto for verdadeiro): no primeiro, aguardamos a chamada de um cliente (por meio da função `accept()`), assim que tivermos, receberemos uma tupla e iremos direcionar a identificação do cliente para a variável “cliente” e a identificação da conexão para a variável “con” e, então, iremos exibir a identificação do nosso cliente;
- No segundo laço, aguardando uma solicitação que pode ser transmitida em pacotes de 1024 bytes, exibimos a mensagem recebida e geramos

uma mensagem para enviar no formato de “**bytes**” (por isso, a mensagem começa com “b” e, em seguida, a *string* que se deseja), enviamos por meio do método “send()” e encerramos esse segundo laço *while*.

- Finalmente, fechamos a conexão e voltamos a aguardar uma nova conexão.

Todos os dados transmitidos via *socket* devem estar no formato *byte*. O *socket* não envia nem recebe dados *strings*, por exemplo.

Veremos agora o lado do cliente, o que mudaria em nosso código.

### 7.1.2 Preparação do cliente



Figura 7.2 – Cliente  
Fonte: Google (2018)

Vamos preparar um código que permita a comunicação de um cliente qualquer, que faça parte da rede, com o nosso servidor. Para isso, crie um arquivo “Python File”, nomeado como **Cliente.py**, e monte o código a seguir:

```
from socket import *
```

```
servidor="127.0.0.1"
porta=43210

msg = bytes(input("Digite algo: "), 'utf-8')
obj_socket=socket(AF_INET, SOCK_STREAM)
obj_socket.connect((servidor,porta))
obj_socket.send(msg)
resposta=obj_socket.recv(1024)
print("Recebemos: ", resposta)
obj_socket.close()
```

Código-fonte 7.4 – Código para cliente  
Fonte: Elaborado pelo autor (2018)

Detalhando as linhas do código apresentado:

- Importamos as funções da biblioteca `socket` e criamos duas variáveis, uma para representar o servidor, caso o servidor não fosse a máquina local, poderíamos definir o IP do servidor externo, se fosse o caso. Cuidado ao definir a porta, deve ser a mesma em que inicializamos o servidor.
- Na quarta linha, definimos o conteúdo da variável “msg”, por meio de uma entrada do usuário (`input`). Observe que utilizamos a função “**bytes()**” para converter o conteúdo do *input*, ou seja, a string para o formato *bytes*, mais uma vez, lembrando: o *socket* transmite somente dados do tipo *byte*. A função “`bytes()`” possui o segundo parâmetro que faz referência ao padrão de caracteres “utf-8”.
- As três próximas linhas são referentes à criação do objeto *socket* (por meio da função “`socket()`”), à conexão com o servidor, por meio da função “`connect()`”, e, finalmente, enviando uma mensagem para o servidor, utilizando o método “`send()`”.
- A variável `resposta` recebe o dado enviado pelo servidor, limitando o tamanho para 1024 *bytes*.
- E finalizamos o código, exibindo a resposta e fechando conexão.

Agora, surge o momento da execução. Caso você tente executar o “Cliente.py”, primeiramente, após digitar o dado a ser transmitido para o servidor, irá receber uma mensagem de erro, conforme aponta a imagem a seguir, pois o servidor não está com a porta disponível, uma vez que você não o executou ainda.



```
C:\Python\python.exe C:/MeusProjetos_Python/Capitulo7_1_Socket/Cliente.py
Digite algo: abc
Traceback (most recent call last):
  File "C:/MeusProjetos_Python/Capitulo7_1_Socket/Cliente.py", line 8, in <module>
    obj_socket.connect((servidor,porta))
ConnectionRefusedError: [WinError 10061] Nenhuma conexão pôde ser feita porque a máquina de destino as recusou ativamente

Process finished with exit code 1
```

Figura 7.3 – Executando o Cliente antes do Servidor  
Fonte: Desenvolvido pelo próprio autor (2018)

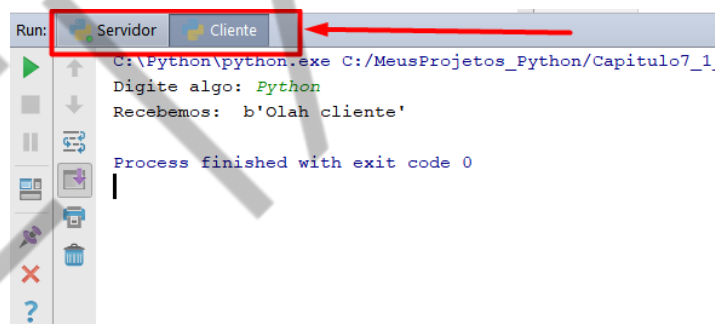
Dessa forma, execute primeiramente a versão do nosso código que está no arquivo “Servidor.py”, seu console ficará da seguinte forma:



```
Run: Servidor2
C:\Python\python.exe C:/MeusProjetos_Python/Capitulo7_1_Socket/Servidor2
Aguardando cliente....
```

Figura 7.4 – Executando o Servidor  
Fonte: Desenvolvido pelo próprio autor (2018)

Após colocar o servidor em execução, execute o nosso arquivo “Cliente.py”, ele irá solicitar o dado a ser transmitido. Digite o dado que deseja transmitir e receba a resposta do servidor, conforme apresentado na figura Executando o Cliente :

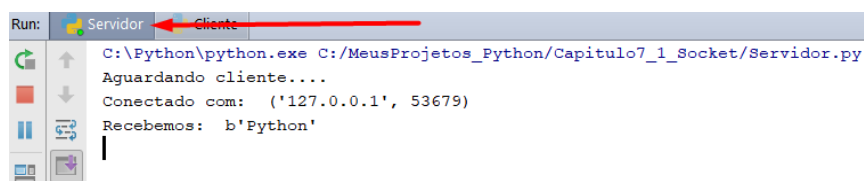


```
Run: Servidor Cliente
C:\Python\python.exe C:/MeusProjetos_Python/Capitulo7_1_
Digite algo: Python
Recebemos: b'Olah cliente'

Process finished with exit code 0
```

Figura 7.5 – Executando o Cliente  
Fonte: Desenvolvido pelo próprio autor (2018)

De acordo com o destaque da imagem, você poderá ver o que chegou ao servidor e como continua o seu status, clicando na guia “Servidor”, conforme apresentado na figura Resposta do Servidor.



```
Run: Servidor Cliente
C:\Python\python.exe C:/MeusProjetos_Python/Capitulo7_1_Socket/Servidor.py
Aguardando cliente....
Conectado com: ('127.0.0.1', 53679)
Recebemos: b'Python'
```

Figura 7.6 – Resposta do Servidor  
Fonte: Desenvolvido pelo próprio autor (2018)

Repare que você pode seguir executando o cliente e passando mensagens para ele, que irá receber sua mensagem e sempre retornará a mensagem “Olah Cliente”. Observe, também, na guia da execução do servidor, que mostra o IP do cliente e a porta, como não definimos uma porta para o cliente, o próprio `socket` gerou e disponibilizou, por meio de uma porta aleatória.

O desafio agora é: você consegue montar um “chat” entre o cliente e o servidor? Ou seja, você pode enviar um dado para o servidor, faça o mesmo com o servidor, e deixe o cliente dentro de um laço infinito para que possam conversar. Tente fazer isso, antes de continuar para o próximo tópico, no qual terá a resposta.

### 7.1.3 Chat Python

Vamos demonstrar, agora, como ficará o código do cliente e do servidor, a fim de montarmos o nosso chat. Vamos começar pelo servidor:

```
from socket import *
servidor="127.0.0.1"
porta=43210
obj_socket = socket(AF_INET, SOCK_STREAM)
obj_socket.bind((servidor,porta))
obj_socket.listen(2)
print("Aguardando cliente...")
while True:
    con, cliente = obj_socket.accept()
    print("Conectado com: ", cliente)
    while True:
        msg_recebida = str(con.recv(1024))
        print("Recebemos: ", str(msg_recebida)[2:-1])
        msg_enviada = bytes(input("Sua resposta: "), 'utf-8')
        con.send(msg_enviada)
        break
    con.close()
```

Código-fonte 7.5 – Código para Servidor\_Chat  
Fonte: Elaborado pelo autor (2018)

Em relação ao novo código do servidor apresentado, alteramos somente o conteúdo do segundo `while`, no qual estamos convertendo a mensagem que recebemos para *string* (`str(msg_recebida)`) e, em seguida, fazendo um *slice* (`[2:-1]`) dela, em que pegamos do segundo caractere até o último caractere da *string*, com isso, quando recebermos, por exemplo: **b'olah servidor'**, será exibido apenas: **olah servidor**. Alteramos também a linha referente ao conteúdo da variável

msg\_enviada, que antes era uma mensagem única e agora é definida por meio do input(), no qual o usuário irá digitar o que desejar.

Em relação ao lado do cliente, vejamos como ficou o código:

```
from socket import *

servidor="127.0.0.1"
porta=43210

while True:
    obj_socket = socket(AF_INET, SOCK_STREAM)
    obj_socket.connect((servidor, porta))
    msg = bytes(input("Sua mensagem: "), 'utf-8')
    obj_socket.send(msg)
    resposta=obj_socket.recv(1024)
    print("Resposta do Servidor: ", str(resposta)[2:-1])
obj_socket.close()
```

Código-fonte 7.6 – Código para Cliente\_Chat  
Fonte: Elaborado pelo autor (2018)

Do lado do cliente, foi incluído o laço *while*, para que o cliente possa mandar mais que uma mensagem ao servidor, deixamos de fora do *while* somente a definição do servidor (ip e porta) e o encerramento da conexão. A resposta do servidor também foi convertida para *string* e foi realizado o *slice* da *string* para considerar somente a mensagem válida.

Execute o servidor, em seguida, o cliente e veja que poderá trocar mensagens entre os dois processos. Se estiver em um ambiente com uma rede de computadores, poderá verificar o funcionamento desses códigos em equipamentos distintos. Claro que muitas aprimorações podem ser realizadas, como, por exemplo, encerrar a conversa do lado do cliente, para isso, bastaria sair do laço infinito:

```
from socket import *

servidor="127.0.0.1"
porta=43210

while True:
    obj_socket = socket(AF_INET, SOCK_STREAM)
    obj_socket.connect((servidor, porta))
    msg = bytes(input("Sua mensagem: "), 'utf-8')
    obj_socket.send(msg)
    resposta=obj_socket.recv(1024)
    print("Resposta do Servidor: ", str(resposta)[2:-1])
    if str(msg)[2:-1].upper()=="FIM":
        break
obj_socket.close()
```

Código-fonte 7.7 – Código para finalizar o Cliente  
Fonte: Elaborado pelo autor (2018)

Adicionamos somente as linhas que estão em vermelho, quando digitarmos “fim” receberemos a mensagem do servidor e, então, a aplicação do lado do cliente será encerrada. Já do lado do servidor, ele continuará escutando novos clientes.

Assim, poderíamos também pensar ainda em muitas melhoras, mas a ideia principal do capítulo é demonstrar como podemos estabelecer comunicação entre dois equipamentos que estejam conectados em um mesmo ambiente de rede, além, é claro, de apresentar a funcionalidade para transmissão de dados entre computadores. Todos os conceitos apresentados serão extremamente úteis para o desenvolvimento de ferramentas em Python que irão permitir controlar e testar informações que estarão trafegando na sua rede.

Como sugestão para estudos futuros, caso você precise rodar uma aplicação cliente-servidor em um ambiente de alta concorrência, ou seja, com muitos clientes conectando-se simultaneamente ao servidor, é possível que você tenha que optar por trabalhar com “threads” (processos dentro do sistema operacional). Nesse caso, você tornaria cada conexão com o cliente uma “thread” e, com isso, o gerenciamento passaria a ser escalável e propício para a alta concorrência ou ainda utilizar a biblioteca “socketserver” junto com “threads”, assim suas requisições para o servidor estariam ainda mais sob controle. Não abordaremos esses modos por fazerem parte de uma programação mais avançada, fugindo do escopo do conteúdo de Python deste momento, que é o de apresentar as possibilidades de aplicações do Python dentro de um ambiente de infraestrutura e segurança.

#### **7.1.4 Protocolo UDP**

Como você já viu anteriormente, o protocolo UDP não é confiável em relação à garantia de que os dados cheguem até o destino, mas, como contraponto, sabemos que esse protocolo entrega de maneira muito mais rápida, quando comparado ao TCP. As aplicações desse protocolo estão mais voltadas para ambientes que utilizem “streaming” de vídeos e/ou áudios ou até mesmo ambientes de jogos online, que têm a velocidade como algo primordial para a jogabilidade. Veremos um pequeno exemplo, utilizando o protocolo UDP para que possamos



completar a manipulação dos dois principais protocolos de transporte dentro do Python.

Vamos iniciar, montando o código do servidor. Para isso, crie o arquivo do tipo “Python File”, chamado **Servidor\_UDP.py**, e monte o seguinte código:

```
from socket import *
servidor="127.0.0.1"
porta=43210
obj_socket = socket(AF_INET, SOCK_DGRAM)
obj_socket.bind((servidor,porta))
print("Servidor pronto....")
while True:
    dados, origem = obj_socket.recvfrom(65535)
    print("Origem.....: ", origem)
    print("Dados recebidos.: ", dados.decode())
    resposta=input("Digite a resposta: ")
    obj_socket.sendto(resposta.encode(), origem)
obj_socket.close()
```

Código-fonte 7.8 – Código para servidor UDP

Fonte: Elaborado pelo autor (2018)

Vamos às descrições do código informado:

- Começamos importando e estabelecendo o endereço do servidor e a porta que será utilizada para a comunicação.
- Na terceira linha, alteramos o segundo parâmetro para DGRAM, o que determina que utilizaremos o protocolo de transporte UDP.
- Realizamos o **bind** do servidor com a porta e endereço especificados.
- Não realizamos o `listen()`, uma vez que não há necessidade de confiabilidade na troca de dados, por isso, já entramos no laço.
- Dentro do laço, aguardamos os dados do cliente, com o tamanho limitado a 65535 (tamanho máximo), prática bem comum, devido à aplicação cotidiana desse protocolo. Perceba que o método agora utilizado é o `recvfrom()`, que não possui ligação com objeto de conexão.
- Os dois `prints()` a seguir servem para exibir a origem e, em seguida, os dados. Perceba que agora não utilizamos mais a conversão para *string* e o *slice*, em vez disso, usamos a função “`decode()`”, que exibe os dados *bytes* no formato *string*. Essa forma também poderia ser utilizada junto ao protocolo TCP.



- Recebemos a resposta que o servidor deseja passar e encaminhamos para o cliente por meio do método “sendto()”, que é formado por dois parâmetros: o primeiro é a mensagem propriamente dita, na qual, dessa vez, utilizamos o método “encode()” para converter de *string* para *byte*; e o segundo parâmetro é a origem para a qual desejamos enviar a mensagem.
- Por fim, finalizamos a conexão do *socket*.

Crie, agora, um novo arquivo “Python File”, chamado **Cliente\_UDP.py**, e digite o código a seguir:

```
from socket import *

servidor="127.0.0.1"
porta=43210
obj_socket = socket(AF_INET, SOCK_DGRAM)
obj_socket.connect((servidor, porta))
saida=""
while saida!="X":
    msg = input("Sua mensagem: ")
    obj_socket.sendto(msg.encode(), (servidor,porta))
    dados, origem = obj_socket.recvfrom(65535)
    print("Resposta do Servidor: ", dados.decode())
    saida=input("Digite <X> para sair: ").upper()
obj_socket.close()
```

Código-fonte 7.9 – Código para cliente UDP  
Fonte: Elaborado pelo autor (2018)

As alterações realizadas nesse código estão destacadas na cor vermelha. Observe que alteramos o segundo parâmetro da função “socket()” para UDP. O método “sendto()” também foi utilizado, pois, diferentemente do método “send()” utilizado no TCP, o “sendto()” possui dois parâmetros: a mensagem e o destino da mensagem. A captura do servidor também foi alterada para o método “recvfrom()”.

Muitas são as possibilidades a partir do conteúdo apresentado, até mesmo criar um “bot”, dentro de um ambiente controlado, no qual o cliente faz uma pergunta, que a aplicação do servidor verifica e, com base em uma análise, envia a resposta para o cliente. Por exemplo, o cliente pergunta: “Qual o e-mail do administrador?”, “Qual o ramal do departamento de pessoal?”, “Quem não efetuou login ainda hoje?”.

O bot pode fazer uma análise sobre as palavras recebidas, então, realizar algum processamento e responder de acordo com a análise/processamento dos

dados que foram recebidos, ajudando, de maneira automática, todos os clientes que estiverem na rede. Essa ideia pode ser ainda mais aprimorada e, por exemplo, fazer com que o servidor esteja conectado a um banco de dados que armazenará perguntas prováveis e respostas já determinadas, ou ainda estar conectado a alguma inteligência artificial, como Watson da IBM, por exemplo, direcionando as respostas para aqueles que fizerem parte da sua rede.

## 7.2 O mundo não é feito só de sockets

Além da biblioteca “sockets”, que, como já vimos, permite a troca de pacotes entre equipamentos que estão conectados em uma mesma rede por meio dos protocolos TCP e UDP, o Python possui muitas outras bibliotecas que permitem manipular outros protocolos, como, por exemplo, SMTP para e-mails, HTTP para envio ou recebimento de dados de uma página WEB, entre outros. Claro que não iremos apresentar todos, isso demandaria uma disciplina inteira.

Mas, a fim de complementarmos um pouco mais e consolidarmos o poder do Python no meio dos protocolos, iremos abordar um pouco agora sobre o FTP (File Transfer Protocol), que tem como principal finalidade trocar arquivos entre os dispositivos que estiverem conectados em uma mesma rede. Uma primeira observação muito importante: utilize-o com moderação, pois esse protocolo é inseguro e pode trazer vulnerabilidades para o seu ambiente, daí a importância em conhecê-lo.

Primeiro, para a manutenção em ambientes legados, em que pode parecer que está tudo funcional e seguro; e, em segundo lugar, pelo simples fato de ser o ponto de partida para aqueles que quiserem, posteriormente, se aprofundar no assunto e partir para protocolos mais atuais, responsáveis pela transmissão de arquivos e que funcionam sobre uma plataforma mais segura.

Para começar, é importante apontar que, por meio do FTP, poderemos realizar downloads, uploads, listagem de arquivos e manutenção também dos diretórios. Outra característica importante para apontarmos neste momento é que o principal ponto falho de segurança é a forma de envio do usuário e senha que

tentarão realizar o login no servidor FTP, pois não recebem qualquer tipo de criptografia e permitem que qualquer monitor de tráfego visualize esses dados.

Caso precise montar um novo ambiente mais seguro para a transferência de arquivos, procure pesquisar e desenvolver a sua plataforma sobre o protocolo HTTP protegido por SSL (*Secure Sockets Layer*), que consiste em uma ferramenta de criptografia que serviu como base para o surgimento do HTTPS (*Hyper Text Transfer Protocol Secure*).

A biblioteca “ftplib” é responsável por permitir a manipulação desse protocolo, logo, teremos que importá-lo dentro do nosso código. Para começarmos a parte prática, crie um “Phyton File”, chamado **Protocolo\_FTP.py**, e digite as seguintes linhas:

```
from ftplib import *
ftp_ativo=False
ftp = FTP('ftp.ibiblio.org')
print(ftp.getwelcome())
ftp.quit()
```

Código-fonte 7.10 – Primeira conexão FTP  
Fonte: Elaborado pelo autor (2018)

Você já pode executar o seu arquivo, deverá ver uma mensagem do servidor FTP e a conexão será finalizada na sequência. Caso o código esteja idêntico e apresente qualquer erro, verifique se o seu computador possui um firewall que esteja bloqueando a conexão com um servidor FTP ou ainda se a rede em que está conectado (no caso de empresa ou qualquer outra instituição) permite uma transmissão FTP. Vamos analisar, com um pouco mais de detalhe, o código proposto anteriormente:

- Na primeira linha, fizemos a importação de todas as funções do módulo ftplib, como, por exemplo, a função FTP(), que permite criar um objeto que representará a conexão estabelecida.
- Na segunda linha, criamos uma variável *booleana*, ou seja, armazenará somente os valores “True” ou “False” (cuidado, pois no Python, os valores “True” e “False” devem ser digitados com a **primeira** letra maiúscula, diferente dos padrões adotados em outras linguagens). Essa variável está definida como *False*, e a utilizaremos posteriormente, para definirmos se a conexão será *ativa* ou *passiva*. A conexão *ativa* era mais utilizada quando

a Internet ainda não era tão difundida, pois permitia que o servidor criasse uma conexão diretamente com o cliente, entretanto, os clientes passaram a ter firewalls e outros tipos de proteções, o que passou a impedir esse tipo de solicitação. Então, a utilização da conexão *passiva* passou a ser mais comum, representando que o servidor irá liberar uma porta e solicitará o estabelecimento da conexão para o cliente, por isso, a variável está com o valor *False*, que representa a conexão *passiva*.

- Na terceira linha, criamos um objeto para representar uma conexão FTP, para isso, utilizamos o método “FTP()”, que precisa apenas do endereço do servidor FTP, no exemplo, utilizamos o endereço “ftp.ibiblio.org”. Caso queira, pode acessá-lo diretamente em seu browser, por meio do endereço “ftp://ftp.ibiblio.org/”, irá surgir uma imagem como a que será apresentada logo a seguir.
- Na quarta linha, chamamos o método “getwelcome()” do objeto “ftp”, que irá apresentar uma mensagem-padrão retornada pelo servidor. Essa mensagem pode, por exemplo, ser útil para informar ao cliente que algum arquivo está indisponível, ou que o servidor está passando por uma atualização, ou simplesmente ser responsável por um “olá”.
- Na última linha, encerramos a conexão.



Figura 7.7 – Acesso ao servidor FTP ibiblio via browser  
Fonte: Desenvolvido pelo próprio autor (2018)

Este servidor “ibiblio” é público (uma biblioteca pública para compartilhamento de arquivos digitais) e permite acesso aos seus arquivos de maneira livre. Perceba que, por intermédio do browser, você pode navegar entre os diretórios, provavelmente não irá encontrar nada de muito interessante entre os diretórios, e também baixar os arquivos, algo que iremos fazer por meio do Python para que você possa automatizar essa ação utilizando uma ferramenta.

Pesquise no Google a seguinte expressão: **inurl:"ftp://ftp."**

A expressão pesquisada é formada por uma dork (inurl) do Google Hacking, que nos permite pesquisar palavras ou termos diretamente nas urls disponíveis para pesquisa. Com esse exemplo, traremos vários endereços de FTP que podem ou não estar acessíveis, como, por exemplo: <ftp://ftp.ibge.gov.br/>, em que você poderá encontrar diversos PDFs que podem ser úteis para *Machine Learning* e/ou pesquisas de âmbito geral.

Continuando o nosso código, iremos acrescentar as seguintes linhas:

```
from ftplib import *
ftp_ativo=False
ftp = FTP('ftp.ibiblio.org')
print(ftp.getwelcome())
usuario=input("Digite o usuario: ")
senha=input("Digite a senha: ")
ftp.login(usuario, senha)
print("Diretório atual de trabalho: ", ftp.pwd())
ftp.quit()
```

Código-fonte 7.11 – Efetuando login no servidor FTP  
Fonte: Elaborado pelo autor (2018)

O que está destacado na cor vermelha representa o que foi adicionado no nosso código. Primeiro, podemos observar que criamos duas variáveis para representarem um usuário e senha, respectivamente (poderíamos, na senha, utilizar a biblioteca “getpass” conforme apresentado em capítulos anteriores). Na sequência, utilizamos o método “login()”, que é responsável por estabelecer a conexão com o servidor. Caso você não informe nenhum usuário e nenhuma senha, entrará no sistema como “**usuário anônimo**”, esse tipo de usuário pode ou não ser permitido pelo servidor. Quando executar, não digite nada em usuário e senha e verá que conseguirá acesso como anônimo. Em seguida, tente executar novamente e tente digitar um usuário qualquer e uma senha qualquer e verá que não terá acesso. Isso significa que, para esse servidor, ou você entra como anônimo, e, com certeza, terá acesso restrito ao conteúdo, ou entra por meio de um usuário previamente

cadastrado no servidor, assim, provavelmente, terá um acesso maior dentro do ambiente.

Seguindo a leitura do código apresentado, na linha do “print()” que foi adicionado, utilizamos o método **pwd()** que retorna o endereço atual de trabalho, ou seja, retorna o diretório em que você se encontra atualmente. Você pode se deslocar entre os diretórios por meio da função **cwd()**, conforme apresentamos a seguir:

```
from ftplib import *
ftp_ativo=False
ftp = FTP('ftp.ibiblio.org')
print(ftp.getwelcome())
usuario=input("Digite o usuario: ")
senha=input("Digite a senha: ")
ftp.login(usuario, senha)
print("Diretório atual de trabalho: ", ftp.pwd())
ftp.cwd('pub')
print("Diretório corrente: ", ftp.pwd())
ftp.quit()
```

Código-fonte 7.12 – Mudando de diretórios no servidor FTP  
Fonte: Elaborado pelo autor (2018)

Como podemos observar, mudamos para o diretório “pub” e, logo depois, podemos observar, por meio da mensagem do “print()”, que realmente estamos em outro diretório. Procure acessar o FTP do IBGE e navegar entre os diretórios.

Podemos também listar os arquivos que encontrarmos nos diretórios, antes de encerrarmos a conexão. Vejamos a seguir:

```
from ftplib import *
ftp_ativo=False
ftp = FTP('ftp.ibiblio.org')
print(ftp.getwelcome())
usuario=input("Digite o usuario: ")
senha=input("Digite a senha: ")
ftp.login(usuario, senha)
print("Diretório atual de trabalho: ", ftp.pwd())
ftp.cwd('pub')
print("Diretório corrente: ", ftp.pwd())
print(ftp.retrlines('LIST'))
ftp.quit()
```

Código-fonte 7.13 – Listando os arquivos do diretório corrente  
Fonte: Elaborado pelo autor (2018)

Como podemos perceber na única linha destacada em vermelho, foi só utilizar o método **dir()**, que nos foi retornada a lista de arquivos encontrados no diretório atual, inclusive, com data de criação e tamanho. Bem prático, não é mesmo? Você

pode montar um laço permitindo que o usuário digite as pastas que deseja navegar e visualizar os conteúdos, procure praticar.

E para baixarmos algum arquivo? Primeiro, precisamos entender que tudo o que for transmitido pelo protocolo é no formato “byte”, assim como vimos nos protocolos TCP e UDP, e, com certeza, teremos que realizar a conversão para *strings*, a fim de que o Python possa interpretá-los e manipulá-los de maneira mais natural. Entretanto, nem todos os arquivos estarão no formato ASCII, como, por exemplo, um arquivo zipado. Por isso, deverão ser transmitidos e baixados em formato de “byte” mesmo. Vejamos, primeiro, o caso de um arquivo ASCII, o arquivo “README”, que está dentro do diretório-raiz. Crie um novo arquivo “Python File”, com o nome de “**FTP\_ASCII.py**”, e digite o seguinte código:

```
import os
from ftplib import *
def escreverLinha(data):
    arq.write(data)
    arq.write(os.linesep)
ftp_ativo=False
ftp = FTP('ftp.ibiblio.org')
print(ftp.getwelcome())
ftp.login()
arquivo='LEIAME'
ftp.set_pasv(ftp_ativo)
with open (arquivo, 'w') as arq:
    ftp.retrlines('RETR README', escreverLinha)
ftp.quit()
```

Código-fonte 7.14 – Baixando arquivos ASCII  
Fonte: Elaborado pelo autor (2018)

Por meio da análise das primeiras linhas desse código, podemos perceber que criamos uma função chamada “escreverLinha()”, a qual explicaremos posteriormente. Conectamo-nos ao servidor “ftp.ibiblio.org”, a mensagem de boas-vindas, e realizamos o login como “anônimo”. Em seguida, definimos o modo de conexão, para o nosso caso “passiva”, conforme explicado anteriormente.

Na sequência, abrimos um “with” para criarmos um arquivo, no qual detalharemos as linhas a seguir:

- Na linha do bloco “with”, definimos que iremos abrir/criar um arquivo chamado “LEIAME” em modo de escrita, e que, dentro desse bloco, será identificado como “arq”.



- Na linha interna ao bloco “with”, iremos utilizar o método **retrlines()**, que é formado por dois parâmetros: no primeiro, devemos especificar o retorno (RETR) e o nome do arquivo que será retornado ('RETR README'); o segundo parâmetro representa uma função que criamos no início do código, a “escreverLinha()”, que receberá o conteúdo da linha (`arq.write(data)`) no “arq” e utilizará o separador de linha, identificado por meio da leitura do sistema operacional, escrevendo o mesmo no “arq” também (`arq.write(os.linesep)`), ou seja, cada linha será escrita no “arq”, uma a uma.
- Por fim, encerramos a conexão.

Quando você executar o arquivo, será gerado, no mesmo diretório do FTP\_ASCII.py, um arquivo chamado LEIAME. Nele, teremos o mesmo conteúdo do arquivo README, que está no ftp.ibiblio.org.

Agora, vamos realizar o código responsável por baixar um arquivo binário, como uma imagem, pdf, vídeo ou qualquer outro formato que não seja ASCII. Crie um novo arquivo do tipo “Python File”, chamado **FTP\_Binario.py**, e digite o código a seguir:

```
from ftplib import *
ftp = FTP('ftp.ibiblio.org')
print(ftp.getwelcome())
ftp.login()
ftp.cwd('/pub/linux/logos/pictures')
with open('pai_do_linux.jpg', 'wb') as arq:
    ftp.retrbinary('RETR linux-father-of-linux.jpg', arq.write)
ftp.quit()
```

Código-fonte 7.15 – Baixando arquivos binários  
Fonte: Elaborado pelo autor (2018)

Como não existe a preocupação na separação de linhas (como nos arquivos ASCII), podemos perceber de pronto que não há a necessidade da função “escreverLinhas()”, que criamos anteriormente. Para baixar um arquivo binário, basta, basicamente: criar a conexão (FTP()), estabelecer o login (login()) e definir o diretório (cwd()). Caso o arquivo não esteja no diretório-raiz e, dentro do “with”, definir o arquivo de saída e também especificar que será binário por meio da flat “wb”, ou seja, o arquivo (pai\_do\_linux.jpg) será criado em modo escrita e também escrito em binário. Dentro do bloco “with”, utilizamos o método “**retrbinary()**”, que



permite especificar o arquivo que será lido ('RETR linus-father-of-linux.jpg') e o arquivo que será escrito (arq.write), depois disso, apenas encerramos a conexão FTP.

Mais uma vez, ao executar esse arquivo, você verá que foi gerado outro, "pai\_do\_linux.jpg", no mesmo diretório do "FTP\_Binario.py", e que pode ser aberto normalmente. Pronto, agora você já pode desenvolver uma ferramenta para baixar arquivos, binários ou não, de um servidor FTP.

### 7.2.1 Um exemplo FTP mais interativo

Vamos montar um exemplo um pouco mais interativo, para que a ferramenta fique mais flexível para o usuário e possa também contar com a sua interação. Para isso, vamos criar um novo arquivo "Python File", chamado "FTP\_Interativo.py", e aproveitaremos parte do conhecimento obtido até o momento para montarmos o código a seguir:

```
import os
from ftplib import *
ftp_ativo=False
ftp = FTP(input("Digite o FTP que se deseja conectar: "))
print(ftp.getwelcome())
usuario=input("Digite o usuario: ")
senha=input("Digite a senha: ")
ftp.login(usuario, senha)
print("\nConexão bem sucedida.\nDiretório atual de trabalho: ", ftp.pwd(),"\n\n")
menu="1"
while menu=="1" or menu=="2" or menu=="3":
    menu=input("Escolha a opção desejada: "
              "\n<1> - para Listar arquivos"
              "\n<2> - para definir um diretório"
              "\n<3> - para baixar um arquivo: ")
    if menu=="1":
        print(ftp.dir())
    elif menu=="2":
        ftp.cwd(input("Digite o diretório que deseja entrar: "))
        print("\nDiretório corrente é: ", ftp.pwd())
    elif menu=="3":
        tipo=input("Digite <B> para arquivo binário ou "
                  "\nqualquer outra letra para arquivo ASCII: ").upper()
        if tipo=="B":
            with open(input("Digite o nome do arquivo destino: "), 'wb') as arq:
                ftp.retrbinary('RETR ' + input("Arquivo de origem: "), arq.write)
        else:
            with open(input("Digite o nome do arquivo destino: "), 'w') as arq:
                def escreverLinha(data):
                    arq.write(data)
                    arq.write(os.linesep)
                ftp.retrlines('RETR ' + input("Arquivo de origem: "), escreverLinha)
            print("Arquivo baixado com sucesso!\n\n")
ftp.quit()
```

Código-fonte 7.16 – Exemplo de um FTP interativo

Fonte: Elaborado pelo autor (2018)

Com base nesse código, realizamos uma interação maior com o usuário, o que permite utilizar o código em diversas situações. Vamos acompanhar linha a linha:

- **Linhas 1 e 2:** realizamos os imports das bibliotecas que serão necessárias para o decorrer do código.
- **Linha 3:** configuramos nossa variável para uma conexão passiva.
- **Linha 4:** definimos o objeto “ftp”, que representará nossa conexão conforme o que o usuário digitar. Quando executar digite, por exemplo, “ftp.ibge.gov.br”.
- **Linhas 5, 6, 7, 8 e 9:** exibimos a mensagem de boas-vindas, solicitamos usuário e senha (pode deixar vazios os dois, para o exemplo do “ibge”), estabelecemos o login, por fim, mostramos uma mensagem de que a conexão foi estabelecida e apresentamos o diretório atual.
- **Linhas 10, 11, 12, 13, 14 e 15:** criamos uma variável “menu” para que o usuário escolha uma das opções desejadas, enquanto o usuário digitar algo igual a 1, 2 ou 3, o laço será repetido e o usuário perguntado novamente sobre qual ação deseja realizar.
- **Linhas 16 e 17:** se o usuário digitar o número “1”, iremos exibir uma listagem do diretório atual, exibindo arquivos e diretórios; após a exibição, irá voltar para o menu.
- **Linhas 18, 19 e 20:** se o usuário digitar o número “2”, iremos mudar de diretório de acordo com o que o usuário digitar. Seguindo o nosso exemplo, digite: “seculox”, em seguida, será exibido o diretório atual e voltará ao menu, utilize novamente a opção “1” para exibir o conteúdo do diretório “seculox”.
- **Linhas 21, 22 e 23:** se o usuário digitar o número “3”, iremos perguntar o que ele deseja baixar. Se for um arquivo binário, ele deverá digitar a letra “b”. Qualquer outro caractere irá considerar o que ele deseja baixar como ASCII. Seguindo o nosso exemplo, digite “B”.

- **Linhas 24, 25 e 26:** nesse caso, se o usuário digitou “B”, iremos pedir o nome do arquivo destino. Diante do nosso exemplo, digite “zipado.zip” (o nome do arquivo deve ser completo, inclusive, com a extensão), em seguida, ele pedirá o arquivo de origem; seguindo o nosso exemplo, escolheremos o arquivo “representacao\_politica.zip” (o nome do arquivo deve ser completo, inclusive, com a extensão). Cuidado que o nome do arquivo deve ser exatamente igual ao que está aparecendo na listagem. Aguarde até surgir a mensagem “Arquivo baixado com sucesso” e, então, aparecerá o menu novamente. Escolha a opção “3” para baixar um arquivo e, na próxima pergunta, digite qualquer letra diferente de “B”, pois iremos baixar um arquivo “ASCII”.
- **Linhas 27 e 28:** se o usuário digitar qualquer letra diferente de “B”, representa que deseja baixar um arquivo “ASCII”. Ele solicitará o nome do arquivo de destino; seguindo o nosso exemplo, digite: “textobaixado.txt” (inclusive com a extensão).
- **Linhas 29, 30 e 31:** criamos a função para realizar a leitura das linhas e a identificação do sistema operacional em relação ao separador de linha utilizado dentro do arquivo.
- **Linha 32:** irá solicitar o nome do arquivo de origem. Seguindo o nosso exemplo, digite o arquivo “leia\_me.txt”, exatamente igual. E então, o “arq” será escrito com base no “leia\_me.txt”.
- **Linha 33:** exibimos a mensagem de que o arquivo foi baixado (CUIDADO com a tabulação dessa linha) e voltaremos para o menu.
- **Linha 34:** somente será executada se o usuário digitar algo diferente de “1”, “2” ou “3”, o que encerrará a conexão.

Se você seguiu os nossos passos, durante a explicação das linhas, deverá ter, no mesmo diretório do arquivo “FTP\_Interativo.py”, os arquivos: “zipado.zip” e “textobaixado.txt”. Nele, poderá exibí-los ou descompactá-los, no caso do arquivo compactado. Assim, o seu usuário poderá interagir em qualquer servidor FTP, navegar entre os diretórios e baixar arquivos ASCII ou binários.

Agora, imagine se pudermos criar uma interface a fim de sairmos do console, o que acha? Legal, não é mesmo? Vamos desenvolver nossas interfaces, criar ferramentas mais interativas e conectar nossas ferramentas a dispositivos IoT. Aguarde... cenas do próximo capítulo!

EMSE

## REFERÊNCIAS

FORBELLONE, André Luiz Villar. **Lógica de programação**. 3. ed. São Paulo: Prentice Hall, 2005.

JET BRAINS. **PyCharm Community, version 2017.2.4**: IDE para programação. Disponível em: <<https://www.jetbrains.com/pycharm/download/#section=windows>>. Último acesso: nov. 2017.

KUROSE, James F. **Redes de computadores e a Internet**: uma abordagem top-down. 6. ed. São Paulo: Pearson Education do Brasil, 2013.

MAXIMIANO, Antonio Cesar Amaru. **Empreendedorismo**. São Paulo: Pearson Prentice Hall Brasil, 2012.

PIVA, Dilermando Jr. **Algoritmos e programação de computadores**. São Paulo: Elsevier, 2012.

PUGA, Sandra. **Lógica de programação e estruturas de dados**. São Paulo: Prentice Hall, 2003.

RAMEZ, Elmasri; SHAMKANT B. Navathe. **Sistemas de banco de dados**. 6. ed. São Paulo: Pearson Addison Wesley, 2011.

RHODES, Brandon. **Programação de redes com Python**. São Paulo: Novatec, 2015.

STALLINGS, W. **Arquitetura e organização de computadores**. 8. ed. São Paulo: Prentice Hall Brasil, 2010.