pync

Brendon Worthington

Oct 20, 2022

CONTENTS

CHAPTER

ONE

USAGE

1.1 Client/Server Model

pync can act as a client:

```
pync [options] dest port[s]
```

or a server:

```
pync -l [options] [dest] port
```

Once a connection has been established, any data read from stdin gets sent to the connection and any data received from the connection gets written to stdout:

```
stdin  --data-> connection
stdout <-data-- connection
```

To illustrate a very basic client/server model, you can connect two **pync** instances together to send messages back and forth.

1. On one console, create a server to listen on a specific port:

Unix

```
pync -l 8000
```

Windows

```
py -m pync -l 8000
```

Python

```
# server.py
from pync import pync
pync('-l 8000')
```

pync is now listening for a connection on port 8000.

2. On a separate console, connect to the server on the port being listened on:

Unix

```
pync localhost 8000
```

Windows

```
py -m pync localhost 8000
```

Python

```
# client.py
from pync import pync
pync('localhost 8000')
```

There should now be a connection between the two consoles and anything typed in one console should display in the other and vice-versa.

When finished, hit Ctrl+C from either console to close the connection.

1.1.1 What's next?

While sending messages back and forth isn't all that interesting, this core concept of redirecting input and output, to and from the network, opens up a range of other possibilities:

- *Data Transfer*
- *Talking to Servers*
- *Port Scanning*
- *Remote Command Execution*
- *Remote Code Execution*

  SEE ALSO

- *-l: Listen mode, for inbound connects*
- Client-server model

1.2 Data Transfer

> **Warning:**
>
> * Please DO NOT transfer any sensitive data using the following methods.

To build on the previous client/server example, we can transfer file data from one machine to another.

1.2.1 Downloading Files

1. Use the -l option to create a server to host the file:

Unix

```
pync -l localhost 8000 < file.in
```

Windows

```
py -m pync -l localhost 8000 < file.in
```

Python

```
# server.py
from pync import pync

# Be sure to open files in binary mode
# for the pync function.
with open('file.in', 'rb') as f:
    pync('-l localhost 8000', stdin=f)
```

2. On a separate console, connect to the server to download the file:

Unix

```
pync localhost 8000 > file.out
```

Windows

```
py -m pync localhost 8000 > file.out
```

Python

```
# client.py
from pync import pync

# Be sure to open files in binary mode
# for the pync function.
with open('file.out', 'wb') as f:
    pync('localhost 8000', stdout=f)
```

During the file transfer, there won't be any progress indication. The connection will close automatically after the file has been transferred.

1.2.2 Uploading Files

You can also upload files by swapping the client/server roles.

1. Create a server to download the file data:

Unix

```
pync -l localhost 8000 > file.out
```

Windows

```
py -m pync -l localhost 8000 > file.out
```

Python

```
# server.py
from pync import pync

# Be sure to open files in binary mode
# for the pync function.
with open('file.out', 'wb') as f:
    pync('-l localhost 8000', stdout=f)
```

2. On a separate console, connect to the server to upload the file:

Unix

```
pync localhost 8000 < file.in
```

Windows

```
py -m pync localhost 8000 < file.in
```

Python

```
# client.py
from pync import pync

# Be sure to open files in binary mode
# for the pync function.
with open('file.in', 'rb') as f:
    pync('localhost 8000', stdin=f)
```

SEE ALSO

- *-l: Listen mode, for inbound connects*

1.3 Talking to Servers

Sometimes it can be useful to interact with servers by hand for troubleshooting or to verify a servers response to certain commands.

1.3.1 Talking to a web server

You can send a GET request to a web server to receive the home page.

Unix

```
printf "GET / HTTP/1.0\r\n\r\n" | pync host.example.com 80
```

Windows

Create a text file (http_get.txt) containing the following:

```
GET / HTTP/1.0

```

That's a GET request line followed by a blank line.
The blank line tells the web server that you're done sending the request and are now ready to receive a response.

Once you've created the http_get.txt file, you can then pipe it into **pync**'s stdin stream to receive the web page:

```
py -m pync -C host.example.com 80 < http_get.txt
```

The -C flag tells **pync** to replace all LF (\n) characters with a CRLF sequence (\r\n).

Python

```
# http_get.py
import io
from pync import pync
# BytesIO turns our request into a file-like
# object for the pync function.
request = io.BytesIO(b'GET / HTTP/1.0\r\n\r\n')
pync('host.example.com 80', stdin=request)
```

1.3.2 Talking to a mail server

You could also submit emails to Simple Mail Transfer Protocol (SMTP) servers.
Suppose you have a text file (email_template.txt):

```
HELO host.example.com
MAIL FROM: <user@host.example.com>
RCPT TO: <user2@host.example.com>
DATA
From: A tester <user@host.example.com>
To: <user2@host.example.com>
Date: date
Subject: a test message

Body of email.
.
QUIT
```

You could then send this template to the server like so:

Unix

```
pync -C smtp.example.com 25 < email_template.txt
```

Windows

```
py -m pync -C smtp.example.com 25 < email_template.txt
```

Python

```
# smtp.py
from pync import pync
with open('email_template.txt', 'rb') as f:
    pync('-C smtp.example.com 25', stdin=f)
```

SMTP typically requires lines to be terminated with a carriage return (CR) line feed (LF) sequence (\r\n).
The -C flag tells **pync** to replace all LF characters (\n) with CRLF characters instead (\r\n).

1.4 Port Scanning

Warning:

* Please be CAREFUL and RESPONSIBLE with this functionality.
* Please DO NOT scan machines without the owners permission first.

Sometimes it's useful to know which ports are open and what services a target machine is running.

1.4.1 Scanning Multiple Ports

By passing a range and/or list of ports, we can connect to multiple ports one after another.

Combining this with the -v and -z options to turn on verbose and zero I/O mode, we can create a simple port scanner:

Unix

```
pync -vz host.example.com 20-30 80 443
```

Windows

```
py -m pync -vz host.example.com 20-30 80 443
```

Python

```
# scan.py
from pync import pync
pync('-vz host.example.com 20-30 80 443')
```

As you can see, you can provide a single port, a list of ports or a range of ports to scan. In this case, we scan port 20 to 30 (20,21,22...30), port 80 (http) and port 443 (https).

For example, if ports 22 and 25 are open, you should see output similar to this:

```
...
Connection to host.example.com 22 port [tcp/ssh] succeeded!
Connection to host.example.com 25 port [tcp/smtp] succeeded!
```

1.4.2 Banner Grabbing

It might also be useful to know which server software is running, and which versions.

This can be done by setting a small timeout with the -w flag, or maybe by issuing a well known command to the server:

Unix

```
echo "QUIT" | pync host.example.com 20-30
```

Windows

```
echo "QUIT" | py -m pync host.example.com 20-30
```

Python

```
# banner.py
import io
from pync import pync

command = io.BytesIO(b'QUIT')
pync('host.example.com 20-30', stdin=command)
```

For example, if SSH was running on port 22, you might see output similar to this:

```
...
SSH-1.99-OpenSSH_3.6.1p2
Protocol mismatch.
```

SEE ALSO

- *-v: Verbose*
- *-z: Zero-I/O mode*

1.5 Remote Command Execution

Warning:

* Please be CAREFUL and RESPONSIBLE with this functionality.
* Please DO NOT use this functionality for evil purposes.

The **-e** option allows you to execute a process and have that process' stdin/stdout/stderr be connected to the network socket.

Any data coming in from the network will go to the process' stdin and any data coming from the process' stdout/stderr will go out to the network.

For example, we can create an interactive shell to execute commands on a remote machine.

1.5.1 A Simple Reverse Shell

1. Create a local server that will listen for the reverse shell connection:

Unix

```
pync -vl localhost 8000
```

Windows

```
py -m pync -vl localhost 8000
```

Python

```
from pync import pync
pync('-vl localhost 8000')
```

2. On another console, connect back to the server and execute the shell:

Unix

```
pync -ve "/bin/sh -i" localhost 8000
```

Windows

```
py -m pync -ve "cmd /q" localhost 8000
```

Python

```
# reverse_shell.py
import platform
from pync import pync

command = '/bin/sh -i'
if platform.system() == 'Windows':
    command = 'cmd /q'

pync('-ve "{}" localhost 8000'.format(command))
```

There should now be a prompt on the server console that allows you to remotely execute commands on the client machine.

1.5.2 A Simple Bind Shell

1. Create a server on port 8000 that executes the shell upon connection:

Unix

```
pync -vle "/bin/sh -i" localhost 8000
```

Windows

```
py -m pync -vle "cmd /q" localhost 8000
```

Python

```
# bind_shell.py
import platform
from pync import pync

command = '/bin/sh -i'
if platform.system() == 'Windows':
    command = 'cmd /q'

pync('-vle "{}" localhost 8000'.format(command))
```

2. On another console, connect to the server to interact with the shell:

Unix

```
pync -v localhost 8000
```

Windows

```
py -m pync -v localhost 8000
```

Python

```
from pync import pync
pync('-v localhost 8000')
```

There should now be a prompt on the client console that allows you to remotely execute commands on the server machine.

SEE ALSO

- *[-e]xecuting [-c]ommands*
- *-l: Listen mode, for inbound connects*
- *-v: Verbose*

1.6 Remote Code Execution

Warning:

Please BE CAREFUL with this functionality as it could expose your system to attackers.
Also, please DO NOT use this functionality for evil purposes.

The **-y** option allows you to execute a string of python code in a separate process and have the process' stdin/stdout/stderr be connected to the network socket.

Any data coming in from the network will go to the process' stdin and any data coming from the process' stdout/stderr will go out to the network.

For example, we can create an interactive python interpreter shell to execute python code on a remote machine.

1.6.1 A Reverse Python Shell

1. Create a local server that will listen for the reverse shell connection:

Unix

```
pync -vl localhost 8000
```

Windows

```
py -m pync -vl localhost 8000
```

Python

```
from pync import pync
pync('-vl localhost 8000')
```

2. On another console, connect back to the server and execute the shell:

Unix

```
pync -vy "import code; code.interact()" localhost 8000
```

Windows

```
py -m pync -vy "import code; code.interact()" localhost 8000
```

Python

```
# reverse_pyshell.py
from pync import pync
pync('-vy "import code; code.interact()" localhost 8000')
```

There should now be a prompt on the server console that allows you to remotely execute python code on the client machine.

1.6.2 A Bind Python Shell

1. Create a server on port 8000 that executes the shell upon connection:

Unix

```
pync -vly "import code; code.interact()" localhost 8000
```

Windows

```
py -m pync -vly "import code; code.interact()" localhost 8000
```

Python

```
# bind_pyshell.py
from pync import pync
pync('-vly "import code; code.interact()" localhost 8000')
```

2. On another console, connect to the server to interact with the shell:

Unix

```
pync -v localhost 8000
```

Windows

```
py -m pync -v localhost 8000
```

Python

```
from pync import pync
pync('-v localhost 8000')
```

There should now be a prompt on the client console that allows you to remotely execute python code on the server machine.

1.6.3 A Python Exec Server

Python's builtin exec function lets you execute a string of python code in a separate namespace.

By reading data from stdin (the network), you can essentially allow arbitrary code to be executed remotely.

1. Create a server that stays open, receiving python code to execute:

Unix

```
pync -vlky "import sys; exec(sys.stdin.read(), {})" localhost 8000
```

Windows

```
py -m pync -vlky "import sys; exec(sys.stdin.read(), {})" localhost 8000
```

Python

```
# pyexec_server.py
from pync import pync
pync('-vlky "import sys; exec(sys.stdin.read(), {})" localhost 8000')
```

We use the **-k** option here to keep the server open between connections, serving one connection after another.

2. Connect to the exec server and send a string of python code to execute:

Unix

```
echo "import sys; sys.stdout.write('Hello\n')" | pync -vq -1 localhost 8000
```

Windows

```
echo "import sys; sys.stdout.write('Hello\n')" | py -m pync -vq -1 localhost 8000
```

Python

```
import io
from pync import pync

pycode = io.BytesIO(b"import sys; sys.stdout.write('Hello\n')")
pync('-vq -1 localhost 8000', stdin=pycode)
```

After executing the above, you should be able to see the message "Hello" printed on the client machine.

Passing a negative number to the **-q** option tells the pync client to keep running after EOF on stdin (after sending the code to execute). Otherwise the client would quit immediately, not giving the server any time to respond.

You should be able to repeat step 2 (sending code to the exec server) for as long as the server is running.

Experiment by sending different lines of code!

When finished, hit Ctrl+C on the server console to stop the server.

SEE ALSO

- *Executing P[-Yy]thon Code*
- *-l: Listen mode, for inbound connects*
- *-v: Verbose*

1.7 pync For Python Developers

There are two main objects of interest when using **pync** in your own Python scripts: the **pync** function and the Netcat class.

1.7.1 Running the pync function

Running the **pync** function is similar to running **pync** from the command-line. It will run a given string of arguments and return an integer exit status value once finished.

The **pync** function also takes a few more keyword arguments: stdin, stdout and stderr.
By default, these arguments point to the console for input and output.

You can pass your own file-like objects to these keyword arguments to control where the data gets read from and written to.

Send a GET request to an HTTP server

```
# pync_http_get.py
import io
import sys

from pync import pync

# io.BytesIO turns the GET request bytes string into a file-like
# object for the pync function.
request = io.BytesIO(b'GET / HTTP/1.0\r\n\r\n')

# pync reads and writes bytes, so be sure to open files in
# binary mode.
with open('http.response', 'wb') as response:
    status = pync('-q -1 host.example.com 80', stdin=request, stdout=response)

sys.exit(status)
```

This example sends a GET request string to a web server and saves the response to a file.

1.7.2 Creating a Netcat instance

Under the hood, the **pync** function creates a custom Netcat class and handles any exceptions that may occur, printing them to stderr.

If you would like more control over exception handling or maybe you'd like to customize your own Netcat, you can use the Netcat class.

Send a GET request to an HTTP server

```
# netcat_http_get.py
import io
from pync import Netcat

# io.BytesIO turns the GET request byte string into a file-like
# object for the Netcat class.
request = io.BytesIO(b'GET / HTTP/1.0\r\n\r\n')

# Netcat reads and writes bytes so be sure to open files in
# binary mode.
response = open('http.response', 'wb')
nc = Netcat('host.example.com', 80,
    q=-1,
    stdin=request,
    stdout=response,
)

try:
    nc.readwrite()
finally:
    response.close()
    nc.close()
```

As before when using the **pync** function, this sends a GET request to a web server and saves the response to a file.

CHAPTER

TWO

OPTIONS

2.1 [-e]xecuting [-c]ommands

> **Warning:**
>
> Please BE CAREFUL with this functionality as it could expose your system to attackers.
> Also, please DO NOT use this functionality for evil purposes.

pync can execute a process and connect the process' stdin/stdout/stderr to the network socket.

Any data that comes in from the network will go to the process' stdin, and any data that comes out from the process' stdout/stderr will be sent out to the network.

There are two options that can provide this functionality, the **-e** option and the **-c** option.

2.1.1 Running a Command With -e

The **-e** option takes the full pathname of a command to execute, along with any arguments.

1. Create a local server that sends "Hello" to the first client that connects:

Unix

```
pync -vle "/bin/echo Hello" localhost 8000
```

Windows

```
py -m pync -vle "echo Hello" localhost 8000
```

Python

```
import platform
from pync import pync

cmd = '/bin/echo Hello'
if platform.system() == 'Windows':
    cmd = 'echo Hello'

pync('-vle "{}" localhost 8000')
```

2. Connect to the Hello server to see the message:

Unix

```
pync -v localhost 8000
```

Windows

```
py -m pync -v localhost 8000
```

Python

```
from pync import pync
pync('-v localhost 8000')
```

2.1.2 Running a Command With -c

The **-c** option is the same as the **-e** option but allows extra shell features such as pipelines and evironment variable expansion.

1. Create a local server that sends the current working directory to the first client that connects:

Unix

```
pync -vle "/bin/echo `pwd`" localhost 8000
```

Windows

```
py -m pync -vle "echo %cd%" localhost 8000
```

Python

```
import platform
from pync import pync

cmd = '/bin/echo `pwd`'
if platform.system() == 'Windows':
    cmd = 'echo %cd%'

pync('-vle "{}" localhost 8000')
```

2. Connect to the server to receive the server's current working directory:

Unix

```
pync -v localhost 8000
```

Windows

```
py -m pync -v localhost 8000
```

Python

```
from pync import pync
pync('-v localhost 8000')
```

SEE ALSO

- *Remote Command Execution*

2.2 -h: show available options and exit.

The **-h** option simply shows **pync**'s help message.

This will display a short description and a brief summary of the available options:

Unix

```
pync -h
```

Windows

```
py -m pync -h
```

Python

```
from pync import pync
pync('-h')
```

2.3 -k: Keep inbound sockets open for multiple connects

By default, **pync**'s TCP server will accept one client before closing the server's socket.

By using the -k option, you can keep the server open to serve multiple clients one after another.

2.3.1 Creating a Date/Time Server

1. Combining -k with the -l and -e options, we can create a simple date/time server that stays open between connections:

Unix

```
pync -kle date localhost 8000
```

Windows

```
py -m pync -kle "time /t && date /t" localhost 8000
```

Python

```
# datetime_server.py
import platform
from pync import pync

command = 'date'
if platform.system() == 'Windows':
    command = 'time /t && date /t'

pync('-kle {} localhost 8000'.format(command))
```

2. To test this, connect to the server on a separate console:

Unix

```
pync localhost 8000
```

Windows

```
py -m pync localhost 8000
```

Python

```
from pync import pync
pync('localhost 8000')
```

By setting the **-k** option on the server, you should be able to keep connecting to it to get the current time and date.

When you're finished, hit Ctrl+C on the server console to close the server.

SEE ALSO

- *[-e]xecuting [-c]ommands*
- *-l: Listen mode, for inbound connects*

2.4 -l: Listen mode, for inbound connects

To create a TCP server, you can use the -l option to listen for incoming connections:

Unix

```
pync -l localhost 8000
```

Windows

```
py -m pync -l localhost 8000
```

Python

```
from pync import pync
pync('-l localhost 8000')
```

If instead you want a UDP server, combine the -l and -u options:

Unix

```
pync -lu localhost 8000
```

Windows

```
py -m pync -lu localhost 8000
```

Python

```
from pync import pync
pync('-lu localhost 8000')
```

SEE ALSO

- *-k: Keep inbound sockets open for multiple connects*
- *-u: UDP mode*

2.5 -q: quit after EOF on stdin and delay of seconds

By default, *-q* is set to 0 to tell **pync** to quit immediately after reaching EOF (End Of File) on stdin.
If you want to have it wait after EOF, set *-q* to the number of seconds you want to wait.
Or if you would like to wait forever until the connection closes, set *-q* to a negative number.

2.5.1 An Example

1. Create a test TCP server:

Unix

```
pync -l localhost 8000
```

Windows

```
py -m pync -l localhost 8000
```

Python

```
from pync import pync
pync('-l localhost 8000')
```

2. Connect to the server with *-q* set to 5 seconds. This will pipe the message "Hello, World!" into **pync**'s stdin, then after EOF has been reached on the message, it will wait 5 seconds before closing:

Unix

```
echo "Hello, World!" | pync -q 5 localhost 8000
```

Windows

```
echo Hello, World! | py -m pync -q 5 localhost 8000
```

Python

```
import io
from pync import pync

# io.BytesIO turns our message into a file-like
# object for the pync function.
message = io.BytesIO(b'Hello, World!')
pync('-q 5 localhost 8000', stdin=message)
```

SEE ALSO

- *-l: Listen mode, for inbound connects*

2.6 -u: UDP mode

By default, **pync** uses TCP (Transmission Control Protocol) for transport.
Using the *-u* option, you can set the transport to UDP (User Datagram Protocol) instead.

2.6.1 A Client/Server Example

1. Create a test UDP server:

Unix

```
pync -lu localhost 8000
```

Windows

```
py -m pync -lu localhost 8000
```

Python

```
# server.py
from pync import pync
pync('-lu localhost 8000')
```

2. On a separate console, connect to the server:

Unix

```
pync -u localhost 8000
```

Windows

```
py -m pync -u localhost 8000
```

Python

```
# client.py
from pync import pync
pync('-u localhost 8000')
```

SEE ALSO

- *-l: Listen mode, for inbound connects*

2.7 -v: Verbose

The *-v* option will print status messages to stderr.
This can be useful to see whether a machine has connected to your server or whether a connection attempt was successful or not.

2.7.1 Example

1. Create a test TCP server:

Unix

```
pync -lv localhost 8000
```

Windows

```
py -m pync -lv localhost 8000
```

Python

```
from pync import pync
pync('-lv localhost 8000')
```

2. On a separate console, connect to the server:

Unix

```
pync -v localhost 8000
```

Windows

```
py -m pync -v localhost 8000
```

Python

```
from pync import pync
pync('-v localhost 8000')
```

You should now be able to see messages printed on each console indicating that the connection was successful:

Server

```
Listening on [localhost] (family 2, port 8000)
Connection from [127.0.0.1] port 8000 [tcp/*] accepted (family 2, sport 44650)
```

Client

```
Connection to 127.0.0.1 8000 port [tcp/*] succeeded!
```

You can also create a simple port scanner by combining *-v* with the *-z* option.
See ../examples/port-scanning for more.

SEE ALSO

- *-l: Listen mode, for inbound connects*
- *-z: Zero-I/O mode*
- ../examples/port-scanning

2.8 Executing P[-Yy]thon Code

Warning:

* Please be CAREFUL and RESPONSIBLE with this functionality.
* Please DO NOT use this functionality for evil purposes.

pync can execute python code in a separate process and connect the process' stdin/stdout/stderr to the network socket.

Any data that comes in from the network will go to the process' stdin, and any data that comes out from the process' stdout/stderr will be sent out to the network.

There are two options that can provide this functionality, the lowercase **-y** option and the uppercase **-Y** option.

2.8.1 Executing Python Code With -y

The lowercase **-y** option takes a string of python code to execute. This option is best used when you have a simple one-liner to execute.

For example, you can create a simple echo server by reading data from stdin (the network) and writing that same data back to stdout (the network):

Unix

```
pync -vly "import sys; sys.stdout.write(sys.stdin.read())" localhost 8000
```

Windows

```
py -m pync -vly "import sys; sys.stdout.write(sys.stdin.read())" localhost 8000
```

Python

```
from pync import pync
pync('-vly "import sys; sys.stdout.write(sys.stdin.read())" localhost 8000')
```

To test this server, connect to it and send it a message:

Unix

```
echo Hello | pync -vq -1 localhost 8000
```

Windows

```
echo Hello | py -m pync -vq -1 localhost 8000
```

Python

```
import io
from pync import pync

hello = io.BytesIO(b'Hello\n')
pync('-vq -1 localhost 8000', stdin=hello)
```

After receiving the message, the echo server should send it back to the client which then would display on the client console.

Here, we pass a negative number to the **-q** option to ensure pync doesn't quit immediately after EOF on stdin (after sending the "Hello" message). Otherwise, there's a chance the client would quit before receiving the message back from the echo server.

Note: You could just as well use the builtin print and input functions for this but because print and input (raw_input on python 2) are different on python 2 and python 3 I just decided using the sys module would be better since it works on both versions of python.

2.8.2 Executing Python Files With -Y

The uppercase **-Y** option takes the full pathname of a python file to execute.

SEE ALSO

- *-q: quit after EOF on stdin and delay of seconds*
- *-v: Verbose*
- *Remote Code Execution*

2.9 -z: Zero-I/O mode

CHAPTER

THREE

API REFERENCE

3.1 pync

pync.**pync**(*args*, *stdin=None*, *stdout=None*, *stderr=None*, *Netcat=<class 'pync.netcat.Netcat'>*)

Create and run a Netcat instance. This is similar to running **pync** from the command-line.

Parameters

- **args** (*str*) – A string containing command-line arguments.
- **stdin** (*file, optional*) – A file-like object to read outgoing network data from.
- **stdout** (*file, optional*) – A file-like object to write incoming network data to.
- **stderr** (*file, optional*) – A file-like object to write error/verbose/debug messages to.

Returns

Error status code depending on success (0) or failure (>0).

Return type

int

Examples

Listing 1: Create a local TCP server on port 8000.

```
from pync import pync
pync('-l localhost 8000')
```

Listing 2: Connect to a local TCP server on port 8000.

```
from pync import pync
pync('localhost 8000')
```

Listing 3: Create a local TCP server to host a file on port 8000.

```
from pync import pync
with open('file.in', 'rb') as f:
    pync('-l localhost 8000', stdin=f)
```

Listing 4: Connect to a local TCP server to download a file on port 8000.

```
from pync import pync
with open('file.out', 'wb') as f:
    pync('localhost 8000', stdout=f)
```

3.2 Netcat

class pync.**Netcat**(*dest=''*, *port=None*, *l=False*, *u=False*, *p=None*, *stdin=None*, *stdout=None*, *stderr=None*, ***kwargs*)

Factory class that returns the correct Netcat object based on the arguments given.

Parameters

- **dest** (*str, optional*) – The IP address or hostname to connect or bind to depending on the "l" parameter.
- **port** (*int, list(int)*) – The port number to connect or bind to depending on the "l" parameter.
- **l** (*bool, optional*) – Set to True to create a server and listen for incoming connections.
- **u** (*bool, optional*) – Set to True to use UDP for transport instead of the default TCP.
- **p** (*int, optional*) – The source port number to bind to.
- **kwargs** – All other keyword arguments get passed to the underlying Netcat class.

You can use this class as a context manager using the "with" statement:

```
with Netcat(...) as nc:
    nc.readwrite()
```

If you use it without the "with" statement, please make sure to use the close method after use:

```
nc = Netcat(...)
nc.readwrite()
nc.close()
```

Examples

Listing 5: Use the "l" option to create a *pync.NetcatTCPServer* object.

```
from pync import Netcat
with Netcat(dest='localhost', port=8000, l=True) as nc:
    nc.readwrite()
```

Listing 6: By default, without the "l" option, Netcat will return a *pync.NetcatTCPClient* object.

```
from pync import Netcat
with Netcat(dest='localhost', port=8000) as nc:
    nc.readwrite()
```

Listing 7: Create a *pync.NetcatUDPServer* with the "u" and "l" options.

```
from pync import Netcat
with Netcat(dest='localhost', port=8000, l=True, u=True) as nc:
    nc.readwrite()
```

Listing 8: And a *pync.NetcatUDPClient* using only the "u" option.

```
from pync import Netcat
with Netcat(dest='localhost', port=8000, u=True) as nc:
    nc.readwrite()
```

Listing 9: Any other keyword arguments get passed to the underlying Netcat class.

```
from pync import Netcat
# Use the "k" option to keep the server open between connections.
with Netcat(dest='localhost', port=8000, l=True, k=True) as nc:
    nc.readwrite()
```

Listing 10: Pass a list of ports to connect to one after the other.

```
# Simple port scan example.
from pync import Netcat
# Use the "z" option to turn Zero i_o on (connect then close).
# Use the "v" option to turn verbose output on to see connection success or failure.
ports = [8000, 8003, 8002]
with Netcat(dest='localhost', port=ports, z=True, v=True) as nc:
    nc.readwrite()
```

ArgumentParser

alias of `NetcatArgumentParser`

TCPClient

alias of *NetcatTCPClient*

TCPServer

alias of *NetcatTCPServer*

UDPClient

alias of *NetcatUDPClient*

UDPServer

alias of *NetcatUDPServer*

classmethod from_args(*args*, *stdin=None*, *stdout=None*, *stderr=None*)

Create a Netcat object from command-line arguments instead of keyword arguments.

Parameters

- **args** (*str*) – A string containing the command-line arguments to create the Netcat instance with.
- **stdin** (*file, optional*) – A file-like object to read outgoing network data from.

- **stdout** (*file, optional*) – A file-like object to write incoming network data to.
- **stderr** (*file, optional*) – A file-like object to write verbose/debug/error messages to.

Example

```
from pync import Netcat
with Netcat.from_args('-l localhost 8000') as nc:
    nc.readwrite()
```

3.3 Clients

class pync.**NetcatClient**(*dest*, *port*, *_4=None*, *_6=None*, *b=None*, *c=None*, *D=None*, *e=None*, *I=None*, *n=None*, *O=None*, *P=None*, *p=None*, *r=None*, *s=None*, *w=None*, *X=None*, *x=None*, *y=None*, *Y=None*, *z=None*, ***kwargs*)

A Netcat client is iterable. You can pass one or more ports and iterate through each *pync.NetcatConnection*.

Parameters

- **dest** (*str*) – The destination hostname or IP address to connect to.
- **port** (*int, list(int)*) – The port number(s) to connect to.
- **e** (*str, optional*) – Execute a command upon connection.
- **z** (*bool, optional*) – Set to True to turn Zero i_o on (connect then close). Useful for simple port scanning.

You can use sub-classes of this class as a context manager using the "with" statement:

```
with NetcatClient(...) as nc:
    nc.readwrite()
```

If you choose not to use the "with" statement, please make sure to use the close() method after use:

```
nc = NetcatClient(...)
nc.readwrite()
nc.close()
```

Example

Listing 11: We can connect to multiple ports one after another by passing a list of ports.

```
with NetcatClient('localhost', [8000, 8001]) as nc:
    for connection in nc:
        connection.readwrite()
```

Listing 12: Using the "z" and "v" options, we can perform a simple port scan.

```
with NetcatClient('localhost', [8000, 8002], z=True, v=True) as nc:
    nc.readwrite()
```

iter_connections()

Override in subclass Iterate through and yield each connection. Close each connection before moving on to the next.

next_connection()

Override in subclass Return the next NetcatConnection.

class pync.**NetcatTCPClient**(*dest*, *port*, *_4=None*, *_6=None*, *b=None*, *c=None*, *D=None*, *e=None*, *I=None*, *n=None*, *O=None*, *P=None*, *p=None*, *r=None*, *s=None*, *w=None*, *X=None*, *x=None*, *y=None*, *Y=None*, *z=None*, ***kwargs*)

Bases: *NetcatClient*

A *pync.NetcatClient* for the Transmission Control Protocol.

Connection

alias of *NetcatTCPConnection*

class pync.**NetcatUDPClient**(*dest*, *port*, *_4=None*, *_6=None*, *b=None*, *c=None*, *D=None*, *e=None*, *I=None*, *n=None*, *O=None*, *P=None*, *p=None*, *r=None*, *s=None*, *w=None*, *X=None*, *x=None*, *y=None*, *Y=None*, *z=None*, ***kwargs*)

Bases: *NetcatClient*

A *pync.NetcatClient* for the User Datagram Protocol.

Connection

alias of *NetcatUDPConnection*

3.4 Servers

class pync.**NetcatServer**(*port*, *dest=''*, *_4=None*, *_6=None*, *b=None*, *c=None*, *D=None*, *e=None*, *I=None*, *k=None*, *n=None*, *O=None*, *y=None*, *Y=None*, ***kwargs*)

A Netcat server is iterable. You can iterate through each incoming connection.

Parameters

- **port** (*int*) – The port number to bind the server to.
- **dest** (*str, optional*) – The hostname or IP address to bind the server to.
- **e** (*str, optional*) – Execute a command upon connection.
- **k** (*bool, optional*) – Set to True to keep the server open between connections.
- **kwargs** – Any other keyword arguments get passed to each connection.

You can use sub-classes of this class as a context manager using the "with" statement:

```
with NetcatServer(...) as nc:
    nc.readwrite()
```

If you don't use the "with" statement, please make sure to use the close() method after use:

```
nc = NetcatServer(...)
nc.readwrite()
nc.close()
```

Example

Listing 13: Use the "k" option to keep the server open and iterate through each *pync.NetcatConnection*.

```
with NetcatServer(8000, dest='localhost', k=True) as nc:
    for connection in nc:
        connection.readwrite()
```

close()

Close the server.

iter_connections()

Override in subclass Iterate through and yield each connection. Close each connection before moving on to the next.

next_connection()

Override in subclass Return the next NetcatConnection.

class pync.**NetcatTCPServer**(*port*, *dest=''*, *_4=None*, *_6=None*, *b=None*, *c=None*, *D=None*, *e=None*, *I=None*, *k=None*, *n=None*, *O=None*, *y=None*, *Y=None*, ***kwargs*)

Bases: *NetcatServer*

A *pync.NetcatServer* for the Transmission Control Protocol.

Connection

alias of *NetcatTCPConnection*

next_connection()

Override in subclass Return the next NetcatConnection.

class pync.**NetcatUDPServer**(*port*, *dest=''*, *_4=None*, *_6=None*, *b=None*, *c=None*, *D=None*, *e=None*, *I=None*, *k=None*, *n=None*, *O=None*, *y=None*, *Y=None*, ***kwargs*)

Bases: *NetcatServer*

A *pync.NetcatServer* for the User Datagram Protocol.

Connection

alias of *NetcatUDPConnection*

3.5 Connections

class pync.**NetcatConnection**(*net*, *C=None*, *d=None*, *i=None*, *q=None*, *w=None*, ***kwargs*)

Wraps a socket object to provide Netcat-like functionality.

Parameters

q (*int, optional*) – Quit the readwrite loop after EOF on stdin and delay of secs.

You can use sub-classes of this class as a context manager using the "with" statement:

```
with NetcatConnection(...) as nc:
    nc.readwrite()
```

If you choose not to use the "with" statement, please make sure to use the close() method after use:

```
nc = NetcatConnection(...)
nc.readwrite()
nc.close()
```

close()

Override to add any cleanup code.

classmethod connect(*dest*, *port*, ***kwargs*)

Factory method to connect to a server and return a NetcatConnection instance. This method should be implemented by a sub-class.

Parameters

- **dest** (*str*) – The destination hostname or IP address to connect to.
- **port** (*int*) – The port number to connect to.
- **kwargs** – Any other keyword arguments get passed to __init__.

Returns

Returns a subclass of *pync.NetcatConnection* once a connection has been established.

Return type

pync.NetcatConnection

Example

```
with NetcatConnection.connect('localhost', 8000) as conn:
    conn.readwrite()
```

classmethod listen(*dest*, *port*, ***kwargs*)

Factory method to listen for a connection and return a NetcatConnection instance. This method should be implemented by a sub-class.

Parameters

- **dest** (*str*) – The hostname or IP address to bind to.
- **port** (*int*) – The port number to bind to.
- **kwargs** – Any other keyword arguments get passed to __init__.

Returns

Returns a subclass of *pync.NetcatConnection* once a connection has been established.

Return type

pync.NetcatConnection

Example

```
with NetcatConnection.listen('localhost', 8000) as conn:
    conn.readwrite()
```

readwrite()

The main loop to read and write i_o. Read from stdin and send to network. Receive from network and write to stdout. Write verbose/debug/error messages to stderr.

This loop is based on the netcat-openbsd 1.105-7 ubuntu version.

Example

```
with NetcatConnection(sock) as nc:
    nc.readwrite()
```

class pync.**NetcatTCPConnection**(*net*, *C=None*, *d=None*, *i=None*, *q=None*, *w=None*, ***kwargs*)

Bases: *NetcatConnection*

Wraps a TCP socket to provide Netcat-like functionality.

classmethod connect(*dest*, *port*, ***kwargs*)

Factory method to connect to a TCP server and return a *pync.NetcatTCPConnection* object.

Parameters

- **dest** (*str*) – The destination hostname or IP address to connect to.
- **port** (*int*) – The port number to connect to.
- **kwargs** – Any other keyword arguments get passed to __init__.

Return type
pync.NetcatTCPConnection

Example

```
from pync import NetcatTCPConnection
with NetcatTCPConnection.connect('localhost', 8000) as conn:
    conn.readwrite()
```

classmethod listen(*dest*, *port*, ***kwargs*)

Factory method to listen for an incoming TCP connection and return a *pync.NetcatTCPConnection* object.

Parameters

- **dest** (*str*) – The destination hostname or IP address to bind to.
- **port** (*int*) – The port number to bind to.
- **kwargs** – Any other keyword arguments get passed to __init__.

Return type
pync.NetcatTCPConnection

Example

```
from pync import NetcatTCPConnection
with NetcatTCPConnection.listen('localhost', 8000) as conn:
    conn.readwrite()
```

class pync.**NetcatUDPConnection**(*net*, *C=None*, *d=None*, *i=None*, *q=None*, *w=None*, ***kwargs*)

Bases: *NetcatConnection*

Wraps a UDP socket object to provide Netcat-like functionality.

classmethod connect(*dest*, *port*, ***kwargs*)

TODO

classmethod listen(*dest*, *port*, ***kwargs*)

TODO

CHAPTER

FOUR

NAME

pync - arbitrary TCP and UDP connections and listens (Netcat for Python).

CHAPTER

FIVE

SYNOPSIS

Unix

```
pync [-46bCDdhklnruvz] [-c string] [-e filename] [-I length]
     [-i interval] [-O length] [-P proxy_username] [-p source_port]
     [-q seconds] [-s source] [-T toskeyword] [-w timeout]
     [-X proxy_protocol] [-x proxy_address[:port]]
     [-Y pyfile] [-y pycode] [dest] [port]
```

Windows

```
py -m pync [-46bCDdhklnruvz] [-c string] [-e filename] [-I length]
           [-i interval] [-O length] [-P proxy_username] [-p source_port]
           [-q seconds] [-s source] [-T toskeyword] [-w timeout]
           [-X proxy_protocol] [-x proxy_address[:port]]
           [-Y pyfile] [-y pycode] [dest] [port]
```

Python

```
from pync import pync
args = '''[-46bCDdhklnruvz] [-c string] [-e filename] [-I length]
          [-i interval] [-O length] [-P proxy_username] [-p source_port]
          [-q seconds] [-s source] [-T toskeyword] [-w timeout]
          [-X proxy_protocol] [-x proxy_address[:port]]
          [-Y pyfile] [-y pycode] [dest] [port]'''
pync(args, stdin, stdout, stderr)
```

CHAPTER

SIX

DESCRIPTION

Inspired by the Black Hat Python book, the goal of **pync** is to create an easy to use library that provides Netcat-like functionality for Python developers.

Common uses include:

- common/tcp-proxy
- shell-script based HTTP clients and servers
- network daemon testing
- a SOCKS or HTTP ProxyCommand for ssh(1)

CHAPTER

SEVEN

INSTALLATION

pync should work on any system with Python installed (version 2.7 or higher).

Use Python's pip command to install **pync** straight from GitHub:

Unix

```
python -m pip install https://github.com/brenw0rth/pync/archive/main.zip
```

Windows

```
py -m pip install https://github.com/brenw0rth/pync/archive/main.zip
```

CHAPTER

EIGHT

USAGE

CHAPTER

NINE

OPTIONS

Option	Description
-4	Use IPv4 addresses only
-6	Use IPv6 addresses only
-b	Allow broadcast
-C	Send CRLF as line-ending
-c string	specify shell commands to exec after connect (use with caution).
-D	Enable the debug socket option
-d	Detach from stdin
-e filename	specify filename to exec after connect (use with caution).
-h, –help	show available options and exit.
-I length	TCP receive buffer length
-i secs	Delay interval for lines sent, ports scanned
-k	Keep inbound sockets open for multiple connects
-l	Listen mode, for inbound connects
-n	Suppress name/port resolutions
-O length	TCP send buffer length
-P proxy_username	Username for proxy authentication
-p source_port	Specify local port for remote connects
-q seconds	quit after EOF on stdin and delay of seconds
-r	Randomize remote ports
-s source	Local source address
-T toskeyword	Set IP Type of Service
-u	UDP mode [default: TCP]
-v	Verbose
-w secs	Timeout for connects and final net reads
-X proxy_protocol	Proxy protocol: “4”, “5” (SOCKS) or “connect”
-x proxy_address[:port]	Specify proxy address and port
-Y pyfile	specify python file to exec after connect (use with caution).
-y pycode	specify python code to exec after connect (use with caution).
-z	Zero-I/O mode [used for scanning]
dest	The destination host name or ip to connect or bind to
port	The port number to connect or bind to

CHAPTER

TEN

API REFERENCE

CHAPTER

ELEVEN

EXAMPLES

Example	Description
chat.py	Simple chat protocol with a custom username
upload.py	Simple file upload (use with caution).
download.py	Simple file download (use with caution).
proxy.py	Simple TCP proxy server
pyshell.py	Reverse or bind python interpreter shell (use with caution).
scan.py	Simple TCP connect port scanner
shell.py	Reverse or bind remote system shell (use with caution).

CHAPTER

TWELVE

SEE ALSO

CHAPTER

THIRTEEN

CAVEATS

UDP port scans will always succeed (i.e report the port as open), rendering the -uz combination of flags relatively useless.

INDEX