### Personal Background

Previously to this course I had no prior programming experience, this project was the first time that I had used Python.

I have worked on projects before which have involved very small amounts of HTML and PHP, so some of the ideas were familiar E.g. Loops and statements. However, after being properly taught this, I realised that at the time I didn’t fully understand what I was doing.

### The Initial Task

For this project, there were four key requirements (1 - 4), with a further six which were optional but nice to have (5 - 10).

1. Hostname.
2. Operating System Version.
3. CPU Architecture.
4. Size / Space information for the root filesystem.
5. List of Users.
6. Networking Information (Interfaces, Routes and Open Network Connections).
7. Currently Active Users.
8. System Performance / Load.
9. Running Processes.
10. Any other useful information.

I was only able to complete tasks (1, 2, 3, 4, 6, 7, 8, 9,10), with (5) attempted but incomplete, for the LocalClientScript.py. For my Client / Server scripts I managed to achieve (1,2,3,4,7) with (5,6,8,9,10) attempted but not 100% functional.

### Focus Points

During this project I thoroughly enjoyed getting stuck in, although frustrating at times the feeling of satisfaction when I was able to figure something out, made it all worthwhile. I started the project as soon as we were given it, and tried to chip away at it little and often. In this report, I have included ‘Snippets’ of code for ease of reading, these are cut down with docstrings, comments and whitespace removed.

## Print Substitution:

As expected I was able to complete tasks (1, 2, and 3) pretty quickly with the use of platform.uname. But the output was horrible and hard to decipher, so I looked up the module and reformatted it using print substitution and calling individual components. I started the project on Python 2, where print didn’t have the parentheses so this was useful. I later switched to Python 3.6, where this is no longer needed.

print("Operating system: %s" % (platforminfo[0]))

Became the same as:

print("Operating system:", platforminfo[0])

## File System Usage:

Initially I found it hard to find a concise method for finding the file system usage information. I found a function built into Python as part of the os module called os.walk. In the help file for this function it had an example, I spent a long time trying to figure out how it was working and how I could manipulate it for my uses.

I eventually worked it out, but in the interim I had come across the psutils module. I was then quickly able to implement a solution to enumerate the disk usage in three values Total Space, Used Space and Free Space.

This completed task (4), however since all I had was the root filesystem information of a single pre-specified directory, I wanted something more adaptable. So I found another function psutil.disk\_partitions, this gathers information about mounted partitions. Included in this was the root mountpoint or drive letter, I was able to manipulate this and put it into a for loop. The loop iterates through each device listed, gathering and printing the usage information.

This was great and worked well, but in windows CD drives are also given a drive letter. And if the drive was for example ‘empty’, then the program would error and crash. So I had to add in exception handling, once I added this feature it worked perfectly and will display usage info for anything mounted.

print("Mounted drives and Usage:", "\n ")  
**for** i **in** part:  
 print(i.device + "\\")  
 **try**:  
 usage = psutil.disk\_usage(i.device + "\\")  
 print("Total Space: ", (SizeConverter(usage[0])))  
 print("Used Space: ", (SizeConverter(usage[1])))  
 print("Free Space: ", (SizeConverter(usage[2])), "\n ")  
 **except** Exception **as** e:  
 print(e)  
 print("Drive unable to be scanned. Usually empty CDROM or Floppy drive.", "\n")  
 **pass**

## OS Walk:

Since I had invested so much time and effort into understanding the os.walk function, I decided I was going to implement it as an extra task (10). I thought from a network administrators point of view, that the ability to scan a specified file path to find usage information would be useful E.g. a users ‘Home Folder’.

So I defined a function that called FileScan, this iterates through all directory’s (including those within) from a specified start point. It prints file path, the contents and total size of files in a directory and a total of the space used.

**def FileScan(**path):  
 totalsize = 0  
 **while True**:  
 **for** filepath, directorys, files **in** os.walk**(**path):  
 print("Filepath: ", filepath)  
 filesize = sum([getsize(join(filepath, name)) **for** name **in** files])  
 print("Total Size of Files:", SizeConverter(filesize))  
 print("Contents:")  
 **for** i **in** files:  
 print(i)  
 totalsize += filesize  
 print("Directory Scanned:", path)  
 print("Total Size of Directory:", SizeConverter(totalsize))  
 **break  
 return**

This function is called as part of an if statement, which asks the user if they would like to use this feature. If ‘Yes’ it will then ask for the user to specify the directory they would like to scan.

**if** scan == "y":  
 print("Please type the filepath you would like to scan, Not Case Sensitive. ")  
 print("E.g. C:\\\\Users\Admin\Downloads")  
 dir = input()  
 dir = dir.lower()  
 FileScan(dir)

## Byte Size Conversion:

From all of the functions that enumerate the drive space used, these outputted in an understandably large number of Bytes. For small files this isn’t too much of a problem, but for larger files it quickly becomes hard to work out. So I defined a function called SizeConverter, this looks to see what ‘Unit’ range the number of Bytes falls in E.g. Kb, Mb, Etc. It will then divide it by the right number of times for that ‘Unit’ E.g. Kb / 1024, Mb / 1024 / 1024, Etc. It will then return that value to two decimal places.

**def SizeConverter**(bytesize):  
 **if** bytesize <= 1024:  
 divided = bytesize  
 **return** str(divided) + " B"  
  
 **elif** bytesize >= 1025 **and** bytesize <= 1048576:  
 divided = (bytesize / 1024)  
 **return** "%.2f" % divided + " Kb"  
  
 **elif** bytesize >= 1048577 **and** bytesize <= 1073741842:  
 divided1 = (bytesize / 1024)  
 divided = (divided1 / 1024)  
 **return** "%.2f" % divided + " Mb"

You can see this function is called and used frequently throughout the script, I found it hard to work out initially but once it was finished, it proved very useful.

## Client / Server:

Initially when I started this project, I never thought that I would get to the client / server stage. So I created my script to be ran entirely on the client machine. But as I progressed further and further through the project, I felt I knew enough to give it a shot, so I started experimenting with sockets trying to get it working.

Eventually I managed to create the certificate files using OpenSSL, and get the SSL socket connection working and passing data. But then I realised that because of the way I had written my script, my functions were no longer going to work. I had created my function to print to screen, pre-formatted from within itself, so no data was being returned at the end of the function (see os.walk code snippet). This meant that I had no values to send through the socket, so I had to completely rewrite my entire script to return values that could be sent.

**def FileScan**(path):  
 totalsize = 0  
 filelist = []  
 sizes = []  
  
 **while True**:  
 **for** filepath, directorys, files **in** os.walk(path):  
   
 fn = "FileScan"  
 host = Host  
 filesize = sum([getsize(join(filepath, name)) **for** name **in** files])  
 **for** i **in** files:  
 filelist.append(i)  
 sizes.append(SizeConverter(filesize))  
 numfiles = (len(files))  
 totalsize += filesize  
  
 JSONData = json.dumps((fn, host, path, filelist, numfiles, sizes, SizeConverter(totalsize), filepath))  
 **break  
 return** JSONData

Then the next problem, I had written one of my functions to do a number of things. So from this function I would need it to return multiple values, that meant putting all of them into a container. Since I wanted to retain the different values own data structures, it made sense to drop all the values into a JSON string, then just return and send that. So after lots of research and coffee, I managed to get the right values I wanted from the function. They were then dumped into JSON, sent over the socket and loaded the other side.

It then became apparent that I had another problem, now that I had the raw data on the server side, I had to parse it. This meant writing a completely different function, that would split the values back out and format them in the right order. After eventually getting the function working, I started looking at doing the same for the other features of my script. I then realised that on the server side, I had no method of identifying the data being received. So for a different feature with a different output, the initial function would try to parse it and error. I also had no way of differentiating one clients information from another.

So I had to tag the data types before they were sent, then look for the values and filter accordingly on the server side. The way I implemented this was simply, to put function name (fn) and host ip address (host) at the start of any data being sent. The server then can define who sent the data and how to parse it correctly. I have done this on the server side with a series of if statements, within my listening loop. So I define server side functions before the loop, then I can call them as needed from the if statements to parse the data. This allows me to keep the server thread running, and without interruption filter data constantly.

**def Listening**():  
  
 **while True**:  
 (readable, writable, exceptional) = select.select(sockets, [], sockets)  
  
 **for** s **in** readable:  
 **if** s **is** master\_socket:  
 (client, \_) = master\_socket.accept()  
 ssl\_socket = ssl.wrap\_socket(client, server\_side=**True**, certfile="server.crt", keyfile="server.key")  
 ssl\_socket.setblocking(0)  
 sockets.append(ssl\_socket)  
 # print("Open Server Connections")  
 # print(sockets)  
  
 **else**:  
 incoming\_data = ssl\_socket.read()  
  
 **if not** incoming\_data:  
 ssl\_socket.shutdown(socket.SHUT\_RDWR)  
 ssl\_socket.close()  
 sockets.remove(ssl\_socket)  
 **else**:  
 decoded\_data = incoming\_data.decode("UTF-8")  
 python\_data = json.loads(decoded\_data)  
 outgoing\_data = "Server Received Data"  
 ssl\_socket.write(outgoing\_data.encode("UTF-8"))

The server script currently is capable of multiple client connections, I have tested it with multiple clients and it receives and processes information with no problems.

### Self-Critique / Potential Developments

If I had more time to continue this project, there are a number of things that I would have liked to implement. For example I started working on a simple block of code to identify the Operating System, from which platform specific commands could then be ran. After testing I confirmed the identifier statement works cross platform, so all that I have left to do is implement the commands. That said, the psutil module works across most major platforms, so the script should work on Windows and Linux, providing that the python version and psutil modules are installed. I ran out of time to fully test whether it did work.

In hindsight, I should have created my script as client / server from the start. That way I would have realised sooner that the way that I was writing my functions was going to cause problems, and I could have rectified it without any major disruption. I also would have realised that the way my tool works could have been optimised better. What I mean by this is that, rather than running the client script which sends all information to the server in bursts. I could have created a simple server side menu structure that would have asked the user what they want to run. From this input a set of instructions could be sent to the client to run the required functions, and then once ran the client could send the data back to the server. I would have liked to look into making a simple GUI, either using python modules, or some kind of simple web interface. This could then be hosted on the company internal web server, or a standalone wamp server, admins could then gather information they need about devices from anywhere with a connection.

I also would have liked to have made an SQL database, this would allow all the information gathered to be logged. It would also facilitate the ability to automate the information gathering. I could have included a section to query client machines on a timed basis. This setup is also far more scalable, for a larger company this monitoring could be happening in the background say once a day in quiet hours. Data can then be analysed in working hours without putting the extra overheads of information gathering on the network. Or potentially you could go the other way and poll at shorter intervals for a more accurate capture.

The client server scripts I have created at the moment are only partially functional (1,2,3,4,7- Working), as I had to re-write pretty much everything, I ran out of time to finish off implementing the other functionality. Given more time I am confident that I could get all the functions I had on the LocalClientScript.py fully integrated. I experienced issued with the sheer amount of data that I was sending across the socket, Json.dumps can only handle a limited size of data. My next step was to refine the information being sent on the client side, then to try and segment the data into smaller chunks then reconstruct server side. For example, my directory scanning function works on client / server, however if the directory has a large amount of information in it, then it will error.

I would have liked to test the server for larger numbers of connections, over a VBox VLAN, from this I would have liked to capture traffic using Wireshark and check that the SSL sockets were working, although I am confident they are.

# Pre-requisites

This tool has been built for use with the Python 3.6 Interpreter, with the psutil module installed.

### User Guides

## Local Client Script

This script is without any form of GUI, so all of the output is printed to screen. The script once ran, will pause at certain points waiting for user input. Initially it will pause straight away asking;

print("Capture System Information Y/N")

This is to confirm that you’re ready to take a capture of the host system information. If you select “Y” then this will output to screen all the system information. If you select “N” then the script will not run and you will get a message, any other value will be rejected and you’ll be asked to enter a “value in range”. The script will then ask whether you would like to use the file system scan function.

"Would you like to scan for filesystem directory information? Y/N"

As before, if you select “N” then the script will not run and you will get a message, any other value will be rejected and you’ll be asked to enter a “value in range”. If you select “Y”, then you will be asked;

Please type the filepath you would like to scan, Not Case Sensitive.

This function can take some time to run depending on the complexity of the file structure

E.g. C:\\Users\Admin\Downloads

Here you can enter any valid system path, the initial script output will display all system mount points which should make it easier. Simply enter a path in the same format as the example above, should you make a mistake and enter an invalid path, the script will return a value like;

#### Directory Scanned: this is an invalid path

#### Total Size of Directory: 0 B

If you enter a correct path, the output will look like;

#### Filepath: c:\\users\admin\downloads

#### Contents:

(

Directory contents listed here.

)

#### Total number of files: 10

#### Total Size of Files: 3.44 Gb

#### Directory Scanned: c:\\users\admin\downloads

#### Total Size of Directory: 3.44 Gb

This function is recursive so it will walk through all directorys under the specified path, displaying the current scanned path, contents, number of files in path and total size of path.

The last two lines displayed are the original path selected to scan, and the total size of ALL files under that path.

## Client / Server Scripts

The client server is far easier to use, first run the Server-New.py script., this will create the listening loop on localhost port 30000.

Then run the Client1.py and client2.py scripts as required, Client1.py is currently set to use host 127.0.0.1 and Client2.py is using localhost.

The server will then print function output to screen like;

----------------------------------------------------------------

PartUsage Function Information Received

From host: 127.0.0.1 at 16 : 53 on 27 / 1 / 2017

C:\

931.02 Gb

81.08 Gb

849.94 Gb

D:\

Drive unable to be scanned. Usually empty CDROM or Floppy drive

E:\

465.73 Gb

150.18 Gb

315.55 Gb

----------------------------------------------------------------

### Version Control

GitHub was used for version control from the start of the project, culminating in 2 branches. Master which is the revisions before implementing Client / Server, and Client / Server which speaks for itself.

To view search GitHub for ‘Python Shizniz’.